The Fortress Language Specification

Version 1.0 α

Eric Allen
David Chase
Joe Hallett
Victor Luchangco
Jan-Willem Maessen
Sukyoung Ryu
Guy L. Steele Jr.
Sam Tobin-Hochstadt

Additional contributors:
Joao Dias
Carl Eastlund
Christine Flood
Yossi Lev
Cheryl McCosh

September 19, 2006

Contents

Part I

Preliminaries

Chapter 1

Introduction

The Fortress Programming Language is a general-purpose, statically typed, component-based programming language designed for producing robust high-performance software with high programmability.

In many ways, Fortress is intended to be a "growable language", i.e., a language that can be gracefully extended and applied in new and unanticipated contexts. Fortress supports state-of-the-art compiler optimization techniques, scaling to unprecedented levels of parallelism and of addressable memory. Fortress has an extensible component system, allowing separate program components to be independently developed, deployed, and linked in a modular and robust fashion. Fortress also supports modular and extensible parsing, allowing new notations and static analyses to be added to the language.

The name "Fortress" is derived from the intent to produce a "secure Fortran", i.e., a language for high-performance computation that provides abstraction and type safety on par with modern programming language principles. Despite this etymology, the language is a new language with little relation to Fortran other than its intended domain of application. No attempt has been made to support backward compatibility with existing versions of Fortran; indeed, many new language features were invented during the design of Fortress. Many aspects of Fortress were inspired by other object-oriented and functional programming languages, including The Java™ Programming Language [5], NextGen [6], Scala [21], Eiffel [16], Self [1], Standard ML [18], Objective Caml [14], Haskell [23], and Scheme [13]. The result is a language that employs cutting-edge features from the programming-language research community to achieve an unprecedented combination of performance and programmability.

1.1 Fortress in a Nutshell

Two basic concepts in Fortress are that of *object* and of *trait*. An object consists of *fields* and *methods*. The fields of an object are specified in its definition. An object definition may also include method definitions.

Traits are named program constructs that declare sets of methods. They were introduced in the Self programming language, and their semantic properties (and advantages over conventional class inheritance) were analyzed by Schärli, Ducasse, Nierstrasz, and Black [8]. In Fortress, a method declared by a trait may be either *abstract* or *concrete*: abstract methods have only *headers*; concrete methods also have *definitions*. A trait may *extend* other traits: it *inherits* the methods provided by the traits it extends. A trait provides the methods that it inherits as well as those explicitly declared in its declaration.

Every object extends a set of traits (its "supertraits"). An object inherits the concrete methods of its supertraits and must include a definition for every method declared but not defined by its supertraits.

`object` SolarSystem `extends` { StarSystem, OrbitingObject }

```
    sun = Sol
    planets = { Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto }
    position = Polar(25000 lightYears, 0 radians)
    ω : ℝ64 AngularVelocity = 2π radians / 226 million years in seconds

    variation(ω_Δ) =
      ω += ω_Δ
end
```

In this example, the object SolarSystem extends the traits StarSystem and OrbitingObject. The fields ω and *position* are defined with appropriate quantities. The field *sun* is defined to be another object named Sol, and the field *planets* is defined to be a set of objects. The method *variation* is defined to take a single parameter ω_Δ, and update the ω field of the object. As this example illustrates, Fortress provides static checking of physical units and dimensions on quantities.

Note that the identifiers used in this example are not restricted to ASCII character sequences. Fortress allows the use of Unicode characters in program identifiers, as well as subscripts and superscripts. (See Appendix E for a discussion of Unicode and support for entering programs in ASCII.) Fortress also allows multiplication to be expressed by simple juxtaposition, as can be seen in the definitions of ω and *position*. Fortress also allows for operator overloading, as well as a facility for extending the syntax with domain-specific languages.

Although Fortress is statically and nominally typed, types are not specified for all fields, nor for all method parameters and return values. Instead, wherever possible, *type inference* is used to reconstruct types. In the examples throughout this specification, we often omit the types when they are clear from context. Additionally, types can be parametric with respect to other types and values (most notably natural numbers).

These design decisions are motivated in part by our goal of making the scientist/programmer's life as easy as possible without compromising good software engineering. In particular, they allow us to write Fortress programs that preserve the look of standard mathematical notation.

In addition to objects and traits, Fortress allows the programmer to define top-level functions. Functions are first-class values: They can be passed to and returned from functions, and assigned as values to fields and variables. Functions and methods can be overloaded, with calls to overloading methods resolved by multiple dynamic dispatch similarly to the manner described in [17]. Keyword parameters and variable size argument lists are also supported.

Fortress programs are organized into *components*, which export and import APIs and can be linked together. APIs describe the "shape" of a component, specifying the types in traits, objects and functions provided by a component. All external references within a component (i.e., references to traits, objects and functions implemented by other components) are to APIs imported by the component. We discuss components and APIs in detail in Chapter 22.

To address the needs of modern high-performance computation, Fortress also supports a rich set of operations for defining parallel execution and distribution of large data structures. This support is built into the core of the language. For example, `for` loops in Fortress are parallel by default.

1.2 Organization

This language specification is organized as follows. In Part II, the Fortress language features for application programmers are explained, including objects, types, and functions. Relevant parts of the concrete syntax are provided with many examples. The full concrete syntax of Fortress is described in Appendix G. In Part III, APIs and documentation of some of the Fortress standard libraries for application programmers are presented. Part IV describes advanced Fortress language features for library writers and Part V presents APIs and documentation for some of the Fortress standard libraries for library writers. Finally, in Part VI, the Fortress calculi, support for Unicode characters, and the Fortress grammars are described.

A note on the presented libraries in Parts III and V: The Fortress standard libraries presented in this draft specification should not be construed as exhaustive or complete. Presentation of additional libraries is planned for future drafts, as are modifications to the libraries included here.

Chapter 2

Overview

In this chapter, we provide a high-level overview of the entire Fortress language. We present most features in this chapter through the use of examples, which should be accessible to programmers of other languages. In this chapter, unlike the rest of the specification, no attempt is made to provide complete descriptions of the various language features presented. Instead, we intend this overview to provide useful context for reading other sections of this specification, which provide rigorous definitions for what is merely introduced here.

2.1 The Fortress Programming Environment

Although Fortress is independent of the properties of a particular platform on which it is implemented, it is helpful to describe a programming model for it that we intend to provide on modern operating systems. In this programming model, Fortress source code is stored in files and organized in directories, and there is a text-based shell from which we can store environment variables and issue commands to execute and compile programs.

There are two ways in which to run a Fortress program:

- As a *script*. The Fortress program is stored in a file with the suffix "`.fsx`" and executed directly from an underlying operating system shell by calling the command "`fortress script`" on it. For example, suppose we write the following "`Hello, world!`" program to a file "`HelloWorld.fsx`":

  `export` Executable
  $run(args) = print$ "`Hello, world!`"

  The first line is an *export statement*; we ignore it for the moment. The second line defines a function run, which takes a parameter named $args$ and prints the string "`Hello, world!`". Note that the parameter $args$ does not include a declaration of its type. In many cases, types can be elided in Fortress and inferred from context. (In this case, the type of $args$ is inferred based on the program's export statement, explained in Section 2.2.)

  We can execute this program by issuing the following command to the shell:

  ```
  fortress script HelloWorld.fsx
  ```

- As a *compiled* file. In this case, the Fortress program is stored in a file with the suffix "`.fss`" and compiled into one or more *components*, which are stored in a persistent database called a *fortress*. Typically, a single fortress holds all the components of a user, or group of users sharing programs and libraries. In our examples, we often refer to the fortress we are issuing commands to as *the resident fortress*.

  For example, we could have written our "`Hello, world!`" program in a compiled file "`HelloWorld.fss`":

```
component HelloWorld
    export Executable
    run(args) = print "Hello, world!"
end
```

We can compile this program, by issuing the command "`fortress compile`" on it:

```
fortress compile HelloWorld.fss
```

As a result of this command, a component named "HelloWorld" is stored in the resident fortress. The name of this component is provided by the enclosing component declaration surrounding the code. If there is no enclosing component declaration, then the contents of the file are understood to belong to a single component whose name is that of the file it is stored in, minus its suffix. For example, suppose we write the following program in a source file named "`HelloWorld2.fss`":

```
export Executable
run(args) = print "Hi, it's me again!"
```

When we compile this file:

```
fortress compile HelloWorld2.fss
```

the result is that a new component with the name HelloWorld2 is stored in the resident fortress. Once this component is compiled, we can execute it by issuing the following command:

```
fortress run HelloWorld2
```

In a script file, there must be at most one component declaration. In a compiled file, multiple component declarations may be included. For example, we could write the following file `HelloWorld3.fss`:

```
component HelloWorld
    export Executable
    run(args) = print "Hello, world!"
end
component HelloWorld2
    export Executable
    run(args) = print "Hi, it's me again!"
end
```

When we compile this file, the result is that both the components HelloWorld and HelloWorld2 are stored in the resident fortress.

If a fortress already contains a component with the same name as a newly installed component, the new component shadows the old one. For example, if we first compile the source file `HelloWorld3.fss` above and then compile the following file `HelloWorld4.fss`:

```
component HelloWorld
    export Executable
    run(args) = print "I didn't expect that!"
end
```

then executing the component HelloWorld on our fortress will result in printing of the following text:

```
I didn't expect that!
```

We can also "remove" a component from a fortress. For example:

```
fortress remove HelloWorld
```

After issuing this command, we can no longer refer to `HelloWorld` component when issuing commands to the fortress. (However, a removed component might still exist as a constituent of other, *linked*, components; see Section 2.2.)

2.2 Exports, Imports, and Linking Components

When a component is defined, it can include *export statements*. For example, all of the components we have defined thus far have included the export statement "`export` Executable". Export statements list various *APIs* that a component implements. Unlike in other languages, APIs in Fortress are themselves program constructs; programmers can rely on standard APIs, and declare new ones. API declarations are sequences of declarations of variables, functions, and other program constructs, along with their types and other supporting declarations. For example, here is the definition of API Executable:

```
api Executable
    run: String... → ()
end
```

This API contains the declaration of a single function run, whose type is $\text{String}\ldots \to ()$. This type is an *arrow type*; it declares the type of a function's parameter, and its return type. The function run includes a single parameter; the notion $\text{String}\ldots$ indicates that it is a *varargs* parameter; the function run can be called with an arbitrary number of string arguments. For example, here are valid calls to this function:

```
run("a simple", " example")
run("run(...)")
run("Nobody", "expects", "that")
```

The return type of run is $()$, pronounced "void". Type $()$ may be used in Fortress as a return type for functions that have no meaningful return value. There is a single value with type $()$: the value $()$, also pronounced "void". References to value $()$ as opposed to type $()$ are resolved by context.

As with components, APIs can be defined in files and compiled. APIs must be defined in files with the suffix `.fsi`. An `.fsi` file contains source code for one or more APIs. If there are no explicit "`api`" headers, the file is understood to define a single API, whose name is the name of the containing file, minus its suffix.

An API is compiled with the shell command "`fortress compile`". When an API is compiled, it is installed in the resident fortress.

For example, if we store the following API in a file named "`Blarf.fsi`":

```
api Zeepf
    foo: String → ()
    baz: String → String
end
```

then we can compile this API with the following shell command:

```
fortress compile Blarf.fsi
```

This command compiles the API Zeepf and installs it in the resident fortress. If we omit the enclosing API declaration, so that the file `Blarf.fsi` consists solely of the following code:

foo: String → ()
baz: String → String

then the file is assumed to consist of the declaration of a single API named Blarf.

Unlike component compilation, API compilation does not shadow existing elements of a fortress. If we attempt to compile an API with the same name as an API already defined in the resident fortress, an error is signaled and the fortress is left unchanged. To remove an API, we must first remove all components referring to the API, and then issue the shell command:

```
fortress removeApi name
```

A component that exports an API must provide a definition for every program construct declared in the API. For example, because our component HelloWorld:

```
component HelloWorld
    export Executable
    run(args) = print "Hello, world!"
end
```

exports the API Executable, it must include a definition for the function run. The definition of run in HelloWorld need not include declarations of the parameter type or return type of run, as these can be inferred from the definition of API Executable.

Components are also allowed to *import* APIs. A component that imports an API is allowed to use any of the program constructs declared in that API. For example, the following component imports API Zeepf and calls the function foo declared in Zeepf:

```
component Blargh
    import Zeepf
    export Executable
    run(args) = Zeepf.foo("whatever")
end
```

Component Blargh imports the API Zeepf and exports the API Executable. Its run function is defined by calling function foo, defined in Zeepf. Note that foo must be referred to by the ***qualified name*** Zeepf.foo, to distinguish it from other declarations of foo that are imported by or defined in Blargh. To call foo as an unqualified name, we can write the following form of import statement:

```
component Blargh
    import {foo} from Zeepf
    export Executable
    run(args) = Zeepf.foo("whatever")
end
```

In an import statement of the form:

```
import S from A
```

all names in the set of names S are imported from API A, and can be referred to as unqualified names within the importing component. In the example above, the set of names we have imported consists of a single name: foo. If we had instead written:

```
import {foo, baz} from Zeepf
```

then we would have been able to refer to both foo and baz as unqualified names in Blargh.

Note that no component refers directly to another component, or to constructs defined in another component. Instead, *all external references go through APIs*. This level of indirection provides us with significant power. As we will see, it is possible to link together arbitrary components, so long as their APIs match. Programmers are able to link together components from separate programming teams, swap in revised components into deployed applications, and even test components that rely on expensive libraries by wiring them up to special *mock components* that provide just enough functionality to allow for testing.

Components that contain no import statements and export the API $\mathrm{Executable}$ are referred to as *executable components*. They can be compiled and executed directly as stand-alone components. All of our $\mathrm{HelloWorld}$ components are executable components. However, if a component imports one or more APIs, it cannot be executed as a stand-alone program. Instead, the component must be compiled and then linked with other components that export all of the APIs it imports, to form a new *compound* component. For example, we define the following component in a file named `Ralph.fss`:

```
export Zeepf
foo(s) = ()
baz(s) = s
```

We can now issue the following shell commands:

```
fortress compile Ralph.fss
fortress compile Blargh.fss
fortress link Gary from Ralph with Blargh
```

The first two commands compile files `Ralph.fss` and `Blargh.fss`, respectively, and install them in the resident fortress. The third command tells the resident fortress to link components Ralph and Blargh together into a compound component, named Gary. Gary is an executable component; we can execute it directly with the command:

```
fortress run Gary
```

All references to API Zeepf in Gary are resolved to the declarations provided in Ralph.

Note that forming the compound component Gary has no effect on the components Ralph and Blargh. These components remain in the resident fortress, and they can be linked together with other components to form yet more compound components. Conversely, if Blargh or Ralph is recompiled, deleted, or otherwise updated, there is no effect on Gary. Conceptually, Gary contains its own copies of the components Blargh and Ralph, and these copies are not corrupted by actions on other components. For this reason, we say that components in Fortress are *encapsulated*. (Of course, there are optimization tricks that a fortress can use to maintain the illusion of encapsulation without actually copying. But these tricks are beyond the scope of this specification.)

Compound components are *upgradable*: They can be upgraded with new components that export some of the APIs used by their constituents. For example, if a new version of Ralph is compiled and installed in the resident fortress, we can manually *upgrade* Gary with the new version by performing the following shell command:

```
fortress upgrade NewGary from Gary with Ralph
```

This command produces a new component, named `NewGary`, resulting from an upgrade of `Gary` with `Ralph`. The components referred to as `Gary` and `Ralph` are unaffected. An important property of fortress components is that they are *stateless*; once constructed, they are never modified. Even if a component is "removed" from a fortress, the components it is a constituent of are unaffected.

However, we can rebind *the name* `Gary` in the resident fortress to our new component with the following command:

```
fortress upgrade Gary from Gary with Ralph
```

or simply:

```
fortress upgrade Gary with Ralph
```

Now, the original component referred to by `Gary` is shadowed; it cannot be referred to directly from the shell. However, it might still exist in the fortress as a constituent of other components, which are unmodified by the upgrade. To upgrade all components in a fortress at once with a new version of `Ralph` (rebinding all names to the resulting upgrades), we can issue the command:

```
fortress upgradeAll Ralph
```

2.3 Automatic Generation of APIs

Note that the component named Zeepf exports the API Zeepf. Components and APIs exist in separate namespaces, and therefore it is allowed for a component to have the same name as an API. In fact, if we hadn't already defined and compiled API Zeepf, we could generate it automatically from the definition of component Zeepf with the following shell command:

```
fortress api Zeepf.fss
```

This command generates a new source file, `Zeepf.fsi`, in the same directory as `Zeepf.fss`, which includes declarations for all program constructs defined in `Zeepf.fss`.

In general, if a component C exports an API A with the same name as C, it is possible to automatically generate a source file for the API A from C by issuing the shell command `fortress api` on the file containing the definition of C. This API contains declarations for all definitions in C that are not declared in other APIs exported by C, and that do not include the modifier `private`. If a source file with the name of A already exists in the same directory as the source file of C, an error is signaled, and no file is created.

If there is more than one component defined in a file, APIs are generated for all components defined in the file that export APIs with the same name as the respective component. Each generated API is placed in a separate source file whose name corresponds to the name of the API it defines.

Note that the API corresponding to an automatically generated source file is not automatically added to the resident fortress; the source file must still be compiled via a separate action. Often, programmers may want to edit the auto-generated file before compiling it to the fortress.

It is not always desirable for a component to export an API of the same name. Many components will export only publicly defined standard APIs. However, automatic generation of APIs from components may be useful, particularly for components defined internally by a development team. Large projects may contain many internally defined components that are never released externally, except as constituents of compound components.

2.4 Rendering

One aspect of Fortress that is quite different from other languages is that various program constructs are rendered in particular fonts, so as to emulate mathematical notation. In the examples above, this was evident by the use of italics when rendering variable names. Many other program constructs have their own rendering rules. For example, the operator `^` indicates superscripting in Fortress. A function definition consisting of the following ASCII characters:

```
f(x) = x^2 + sin x - cos 2 x
```

is rendered as follows:

$$f(x) = x^2 + \sin x - \cos 2x$$

Array indexing, written with brackets:

```
a[i]
```

is rendered as follows:

a_i

There are many other examples of special rendering conventions.

Fortress also supports the use of Unicode characters in identifiers. In order to make it easy to enter such characters with today's input devices, Fortress defines a convention for keyboard entry: ASCII names (and abbreviations) of Unicode characters can be written in a program in all caps (with spaces replaced by underscores). Such identifiers are converted to Unicode characters. For example, the identifier:

```
GREEK_CAPITAL_LETTER_LAMBDA
```

is automatically converted into the identifier:

Λ

There are also ASCII abbreviations for writing down commonly used Fortress characters. For example, ASCII identifiers for all Greek letters are converted to Greek characters (e.g., "`lambda`" becomes λ and "`LAMBDA`" becomes Λ). Here are some other common ASCII shorthands:

`BY`	*becomes*	$\times$	`TIMES`	*becomes*	$\times$
`DOT`	*becomes*	$\cdot$	`CROSS`	*becomes*	$\times$
`CUP`	*becomes*	$\cup$	`CAP`	*becomes*	$\cap$
`BOTTOM`	*becomes*	$\bot$	`TOP`	*becomes*	$\top$
`SUM`	*becomes*	$\sum$	`PRODUCT`	*becomes*	$\prod$
`INTEGRAL`	*becomes*	$\int$	`EMPTYSET`	*becomes*	$\emptyset$
`SUBSET`	*becomes*	$\subset$	`NOTSUBSET`	*becomes*	$\not\subset$
`SUBSETEQ`	*becomes*	$\subseteq$	`NOTSUBSETEQ`	*becomes*	$\not\subseteq$
`EQUIV`	*becomes*	$\equiv$	`NOTEQUIV`	*becomes*	$\not\equiv$
`IN`	*becomes*	$\in$	`NOTIN`	*becomes*	$\notin$
`LT`	*becomes*	$<$	`LE`	*becomes*	$\leq$
`GT`	*becomes*	$>$	`GE`	*becomes*	$\geq$
`EQ`	*becomes*	$=$	`NE`	*becomes*	$\neq$
`AND`	*becomes*	$\wedge$	`OR`	*becomes*	$\vee$
`NOT`	*becomes*	$\neg$	`XOR`	*becomes*	$\oplus$
`INF`	*becomes*	∞	`SQRT`	*becomes*	$\surd$

A comprehensive description of ASCII conversion is provided in Appendix E.

2.5 Some Common Types in Fortress

Fortress provides a wide variety of standard types, including String, $\mathrm{Boolean}$, and various numeric types. The floating-point type $\mathbb{R}$ (written in ASCII as `RR`) denotes 64-bit precision floating-point numbers. For example, the following function takes a 64-bit float and returns a 64-bit float:

$halve(x : \mathbb{R}) : \mathbb{R} = x/2$

The type $\mathbb{R}$ is also written $\mathbb{R}64$. The following definition of $halve$ is semantically equivalent to the one above:

$halve(x : \mathbb{R}64) : \mathbb{R}64 = x/2$

Predictably, 32-bit precision floats are written $\mathbb{R}32$.

64-bit integers are denoted by the type $\mathbb{Z}64$. 32-bit integers by the type $\mathbb{Z}32$, and infinite precision integers by the type $\mathbb{Z}$.

2.6 Functions in Fortress

Fortress allows recursive, and mutually recursive function definitions. Here is a simple definition of a *factorial* function in Fortress:

$factorial(n) =$
    `if` $n = 0$ `then` 1
    `else` $n\ factorial(n-1)$ `end`

Note the juxtaposition of parameter n with the recursive call $factorial(n-1)$. In Fortress, as in mathematics, multiplication is represented through juxtaposition. By default, two expressions of numeric type that are juxtaposed represent a multiplication. On the other hand, juxtaposition of an expression with a function type with another expression to its right represents function application, as in the following example:

$\sin x$

2.6.1 Keyword Parameters

Functions in Fortress can be defined to take keyword arguments by providing default values for parameters. In the following example:

$makeColor(red : \mathbb{Z}64 = 0, green : \mathbb{Z}64 = 0, blue : \mathbb{Z}64 = 0) =$
    `if` $0 \le red \le 255 \wedge 0 \le green \le 255 \wedge 0 \le blue \le 255$
    `then` $\mathrm{Color}(red, green, blue)$
    `else throw` Error `end`

the function $makeColor$ takes three keyword arguments, all of which default to 0. If we call it as follows:

$makeColor(green = 255)$

the arguments red and $blue$ are both given the value 0.

There are some other aspects of this example worth mentioning. For example, the body of this function consists of an `if` expression. The test of the `if` expression checks that all three parameters have values between 0 and 255. The boolean operator $\le$ is *chained*: An expression of the form $x \le y \le z$ is equivalent to the expression $x \le y \wedge y \le z$, just as in mathematical notation. The `then` clause provides an example of a constructor call in Fortress, and the `else` clause shows us an example of a `throw` expression in Fortress.

2.6.2 Varargs Parameters

It is also possible to define functions that take a variable number of arguments. We have already seen such a function: $\mathrm{Executable}.run$. Here is another:

```
printFirst(xs : ℝ...) =
    if |xs| > 0 then print xs_0
    else throw Error end
```

This function takes an arbitrary number of floats and prints the first one (unless it is given zero arguments; then it throws an exception).

2.6.3 Function Overloading

Functions can be overloaded in Fortress by the types of their parameters. Calls to overloaded functions are resolved based on the runtime types of the arguments. For example, the following function is overloaded based on parameter type:

```
size(x : Tree) = 1 + size(leftBranch(x)) + size(rightBranch(x))
size(x : List) = 1 + size(rest(x))
```

Suppose we call $size$ on an object with runtime type List, the second definition of $size$ will be invoked *regardless of the static type of the argument*. Of course, function applications are statically checked to ensure that *some* definition will be applicable at run time, and that the definition to apply will be unambiguous.

2.6.4 Function Contracts

Fortress allows *contracts* to be included in function declarations. Among other things, contracts allow us to *require* that the argument to a function satisfies a given set of constraints, and to *ensure* that the resulting value satisfies some constraints. They provide essential documentation for the clients of a function, enabling us to express semantic properties that cannot be expressed through the static type system.

Contracts are placed at the end of a function header, before the function body. For example, we can place a contract on our $factorial$ function requiring that its argument be nonnegative as follows:

```
factorial(n) requires n ≥ 0
    = if n = 0 then 1
      else n factorial(n − 1) end
```

We can also ensure that the result of $factorial$ is itself nonnegative:

```
factorial(n)
    requires n ≥ 0
    ensures result ≥ 0
    = if n = 0 then 1
      else n factorial(n − 1) end
```

The variable $result$ is bound in the `ensures` clause to the return value of the function.

2.7 Some Common Expressions in Fortress

We have already seen an `if` expression in Fortress. Here's an example of a `while` expression:

```
while x < 10 do
    print x
    x += 1
end
```

Blocks in Fortress are delimited by the special reserved words `do` and `end`. Here is an example of a function that prints three words:

```
printThreeWords() = do
    print "print"
    print "three"
    print "words"
end
```

A *tuple* expression contains a sequence of elements delimited by parentheses and separated by commas:

```
("this", "is", "a", "tuple", "of", "mostly", "strings", 0)
```

When a tuple expression is evaluated, the various subexpressions are evaluated *in parallel*. For example, the following tuple expression denotes a parallel computation:

$$(factorial(100), factorial(500), factorial(1000))$$

The elements in this expression may be evaluated in parallel.

2.8 For Loops Are Parallel by Default

Here is an example of a simple `for` loop in Fortress:

```
for i ← 1:10 do
    print(i " ")
end
```

This `for` loop iterates over all elements i between 1 and 10 and prints the value of i. Expressions such as $1:10$ are referred to as *range expressions*. They can be used in any context where we wish to denote all the integers between a given pair of integers.

A significant difference between Fortress and most other programming languages is that *for loops are parallel by default*. Thus, printing in the various iterations of this loop can occur in an arbitrary order, such as:

5 4 6 3 7 2 9 10 1 8

2.9 Atomic Expressions

In order to control interactions of parallel executions, Fortress includes the notion of *atomic expressions*, as in the following example:

```
atomic do
    x += 1
    y += 1
end
```

An atomic expression is executed in such a manner that all other threads observe either that the computation has completed, or that it has not yet begun; no other thread observes an atomic expression to have only partially completed. For example, in the following parallel computation:

```
do
    x = 0
    y = 0
    z: ℤ := 0
    (atomic do
         x += 1
         y += 1
     end,
     z := x + y)
     z
end
```

the second subexpression in the tuple expression either observes that both x and y have been updated, or that neither has. Thus, possible values of the `do` expression are 0 and 2, but not 1.

2.10 Dimensions and Units

Numeric types in Fortress can be annotated with physical units and dimensions. For example, the following function declares that its parameter is a tuple represented in the units kg and $\mathrm{m/s}$, respectively:

$$kineticEnergy(m : \mathbb{R}\,\mathrm{kg}, v : \mathbb{R}\,\mathrm{m/s}) : \mathbb{R}\,\mathrm{kg\,m^2/s^2} = (m\,v^2)/2$$

A value of type $\mathbb{R}\,\mathrm{kg}$ is a 64-bit float representing a measurement in kilograms. When the function $kineticEnergy$ is called, two values in its tuple argument are statically checked to ensure that they are in the right units.

All commonly used dimensions and units are provided in the Fortress standard libraries. Unit symbols are encoded with trailing underscores; such identifiers are rendered in roman font. For example the unit m is represented as `m_`. For each unit, both longhand and shorthand names are provided (e.g., m, meter, and meters). The various names of a given unit can be used interchangeably. Also, some units (and dimensions) are defined to be synonymous with algebraic combinations of other units (and dimensions). For example, the unit N is defined to be synonymous with the unit $\mathrm{kg\,m/s^2}$ and the dimension Force is defined to be synonymous with $\mathrm{Mass\,Acceleration}$. Likewise, the dimension $\mathrm{Acceleration}$ is defined to be synonymous with $\mathrm{Velocity/Time}$.

Measurements in the same unit can be compared, added, subtracted, multiplied and divided. Measurements in different units can be multiplied and divided. For example, we can write the following variable declaration:

$$v : \mathbb{R}\,\mathrm{m/s} = (3\,\mathrm{meters} + 4\,\mathrm{meters})/5\,\mathrm{seconds}$$

However, the following variable declaration is a static error:

$$v : \mathbb{R}\,\mathrm{m/s} = (3\,\mathrm{meters} + 4\,\mathrm{seconds})/5\,\mathrm{seconds}$$

In addition, the following variable declaration is a static error because the unit of the left hand side of this declaration is $\mathrm{m/s}$ whereas the unit of the right hand side is simply m (or meters):

$$v : \mathbb{R}\,\mathrm{m/s} = (3\,\mathrm{meters} + 4\,\mathrm{meters})/5$$

It is also possible to convert a measurement in one unit to a measurement of another unit of the same dimension, as in the following example:

$$kineticEnergy(3.14\,\mathrm{kg}, 32\,\mathrm{f/s}\ \texttt{in}\ \mathrm{m/s})$$

The second argument to $kineticEnergy$ is a measurement in feet per second, converted to meters per second.

2.11 Aggregate Expressions

As with mathematical notation, Fortress includes special syntactic support for writing down many common kinds of collections, such as tuples, arrays, matrices, vectors, maps, sets, and lists simply by enumerating all of the collection's elements. We refer to an expression formed by enumerating the elements of a collection as an *aggregate expression*. The elements of an aggregate expression are computed in parallel.

For example, we can define an array a in Fortress by explicitly writing down its elements, enclosed in brackets and separated by whitespaces, as follows:

$$a = [0\ 1\ 2\ 3\ 4]$$

Two-dimensional arrays can be written down by separating rows by newlines (or by semicolons). For example, we can bind b to a two-dimensional array as follows:

$$b = [3\ 4$$
$$\quad 5\ 6]$$

There is also support for writing down arrays of dimension three and higher. We bind c to a three-dimensional array as follows:

$$c = [1\ 2$$
$$\quad 3\ 4;;5\ 6$$
$$\quad\quad 7\ 8;;9\ 10$$
$$\quad\quad\quad 11\ 12]$$

Various slices of the array along the third dimension are separated by pairs of semicolons. (Higher dimensional arrays are also supported. When writing a four-dimensional array, slices along the fourth dimension are separating by triples of semicolons, and so on.)

Vectors are written down just like one-dimensional arrays. Similarly, matrices are written down just like two-dimensional arrays. Whether an array aggregate expression reduces to an array, a vector, or a matrix is inferred from context (e.g., the static type of a variable that such an expression is bound to). Of course, all elements of vectors and matrices must be numbers.

A set can be written down by enclosing its elements in braces and separating its elements by commas. Here we bind s to the set of integers 0 through 4:

$$s = \{0, 1, 2, 3, 4\}$$

The elements of a list are enclosed in angle brackets (written in ASCII as `<|` and `|>`):

$$l = \langle 0, 1, 2, 3, 4 \rangle$$

The elements of a map are enclosed in curly braces, with key/value pairs joined by the arrow $\mapsto$ (written in ASCII as `|->`):

$m = \{'a' \mapsto 0, 'b' \mapsto 1, 'c' \mapsto 2\}$

2.12 Comprehensions

Another way in which Fortress mimics mathematical notation is in its support for *comprehensions*. Comprehensions describe the elements of a collection by providing a rule that holds for all of the collection's elements. The elements of the collection are computed in parallel by default. For example, we define a set s that consists of all elements of another set t divided by 2, as follows:

$s = \{x/2 \mid x \leftarrow t\}$

The expression to the left of the vertical bar explains that elements of s consist of every value $x/2$ for every valid value of x (determined by the right hand side). The expression to the right of the vertical bar explains how the elements x are to be *generated* (in this case, from the set t). The right hand side of a comprehension can consist of multiple generators. For example, the following set consists of every element resulting from the sum of an element of s with an element of t:

$u = \{x + y \mid x \leftarrow s, y \leftarrow t\}$

The right hand side of a comprehension can also contain *filtering expressions* to constrain the elements provided by other clauses. For example, we can stipulate that v consists of all nonnegative elements of t as follows:

$v = \{x \mid x \leftarrow t, x \geq 0\}$

There is a comprehension expression for every form of aggregate expression except tuple expressions. For example, here is a list comprehension in Fortress:

$\langle 2x | x \leftarrow v \rangle$

The elements of this list consist of all elements of the set v multiplied by 2.

In the case of an array comprehension, the expression to the left of the bar consists of a tuple indexing the elements of the array. For example, the following comprehension describes a 3×3 array, all of whose elements are zero.

$[(x, y) = 0 \mid x \leftarrow \{0, 1, 2\}, y \leftarrow \{0, 1, 2\}]$

The collection of elements 0 through 2 can also be expressed via a range expression, as follows:

$[(x, y) = 0 \mid x \leftarrow 0:2, y \leftarrow 0:2]$

An array comprehension can consist of multiple clauses. The clauses are run in the order provided, with declarations from later clauses shadowing declarations from earlier clauses. For example, the following comprehension describes a 3×3 identity matrix:

$$[(x, y) = 0 \mid x \leftarrow 0:2, y \leftarrow 0:2$$
$$(x, x) = 1 \mid x \leftarrow 0:2]$$

2.13 Summations and Products

As with mathematical notation, Fortress provides syntactic support for summations and productions (and other big operations) over the elements of a collection. For example, an alternative definition of $factorial$ is as follows:

$$factorial(n) = \prod_{i \leftarrow 1:n} i$$

This function definition can be written in ASCII as follows:

```
factorial(n) = PRODUCT[i <- 1:n] i
```

The character $\prod$ is written `PRODUCT`. Likewise, $\sum$ is written `SUM`. As with comprehensions, the values in the iteration are generated from specified collections.

2.14 Tests and Properties

Fortress includes support for automated program testing. New tests in a component can be defined using the *test* modifier. For example, here is a test that calls to the $factorial$ function result in values greater than or equal to what was provided:

`test` $factorialResultLarger[x \leftarrow 0:100] = x \leq factorial(x)$

The values for x are generated from the provided range $0:100$.

Programs are also allowed to include *property* declarations, documenting boolean conditions that a program is expected to obey. Property declarations are similar syntactically to test declarations. Unlike test declarations, property declarations do not specify explicit finite collections over which the property is expected to hold. Instead, the parameters in a property declaration are declared to have types, and the property is expected to hold over all values of those types. For example, here is a property declaring that $factorial$ is greater than or equal to every argument of type $\mathbb{Z}$:

`property` $factorialResultAlwaysLarger = \forall(x : \mathbb{Z})\ (x \leq factorial(x))$

When a property declaration includes a name, the property identifier is bound to a function whose parameter and body are that of the property, and whose return type is $\mathrm{Boolean}$. A function bound in this manner is referred to as a *property function*. A property function can be called from within tests to ensure than a property holds at least for all values in some finite set of values.

2.15 Objects and Traits

A great deal of programming can be done simply through the use of functions, top-level variables, and standard types. However, Fortress also includes a trait and object system for defining new types, as well as objects that belong to them. Traits in Fortress exist in a multiple inheritance hierarchy rooted at trait Object. A trait declaration includes a set of method declarations, some of which may be abstract. For example, here is a declaration of a simple trait named Moving:

```
trait Moving extends { Tangible, Object }
  position(): (ℝ Length)^3
  velocity(): (ℝ Velocity)^3
end
```

The set of traits extended by trait Moving are listed in braces after the special reserved word `extends`. Trait Moving inherits all methods declared by each trait it extends. The two methods $position$ and $velocity$ declared in trait Moving are *abstract*; they contain no body. Their return types are vectors of length 3, whose elements are of types $\mathbb{R}\,\mathrm{Length}$ and $\mathbb{R}\,\mathrm{Velocity}$ respectively. As in mathematical notation, a vector of length n with element type T is written T^n.

Traits can also declare concrete methods, as in the following example:

```
trait Fast extends Moving
    velocity() = [0 0 (299 792 458 m/s)]
end
```

Trait Fast extends a single trait, Moving. (Because it extends only one trait, we can elide the braces in its `extends` clause.) It inherits both abstract methods defined in Moving, and it provides a concrete body for method *velocity*.

Trait declarations can be extended by other trait declarations, as well as by *object declarations*. There are two kinds of object definitions. A *singleton object declaration* defines a sole, stand-alone object. For example:

```
object Sol extends { Moving, Stellar }
    spectralClass = G2
    position() = [0 0 0]
    velocity() = [0 0 0]
end
```

The object Sol extends two traits: Moving and Stellar, and provides definitions for the abstract methods it inherits. Objects must provide concrete definitions for all abstract methods they inherit. Sol also defines a field *spectralClass*. For every field included in an object definition, an implicit getter is defined for that field. Calls to the getter of the field of an object consist of an expression denoting the object followed by a dot and the name of the field. For example, here is a call to the getter *spectralClass* of object Sol:

```
Sol.spectralClass
```

If a field includes modifier `settable`, an implicit setter is defined for the field. Calls to setters look like assignments. Here is an assignment to field *spectralClass* of object Sol:

```
Sol.spectralClass := G3
```

In fact, all accesses to a field from outside the object to which the field belongs must go through the getter of the field, and all assignments to it must go through the setter. (There is simply no other way syntactically to refer to the field.)

If a field includes modifier `hidden`, no implicit getter is defined for the object.

Every method declared in an object or trait includes an implicit *self* parameter, denoting the receiver of the method call. If desired, this parameter can be made explicit, by including it before the name of the method, along with a trailing dot. For example, the following definition of Sol is equivalent to the one above:

```
object Sol extends {Moving, Stellar}
    spectralClass = G2
    self.position() = [0 0 0]
    velocity() = [0 0 0]
end
```

In this definition, the self parameter of *position* is provided explicitly.

In fact, the self parameter of a method can even be given a position other than the default position. For example, here is a definition of a type where the self parameters appear in nonstandard positions:

```
trait List
    cons(x: Object, self) : List
    append(xs: List, self) : List
end
```

In both methods $cons$ and $append$, the self parameter occurs as the second parameter. Calls to these methods look more like function calls than method calls. For example, in the following call to $append$, the receiver of the call is l_2:

$append(l_1, l_2)$

In contrast to singleton object definitions, a *dynamically parametric object definition* includes a *constructor declaration* in its header, as in the following example:

```
object Particle(position : (ℝ Length)^3,
                velocity : (ℝ Velocity)^3)
  extends Moving
end
```

Every call to the constructor of dynamically parametric object Particle yields a new object. For example:

$p_1 = \text{Particle}([3\,\text{m}\ 2\,\text{m}\ 5\,\text{m}], [(1\,\text{m/s})\ 0\ 0])$

The parameters to the constructor of a dynamically parametric object implicitly define fields of the object, along with their getters (by default). These parameters can include all of the modifiers that an ordinary field can. For example, a parameter can include the modifier `hidden`, in which case a getter is not implicitly defined for the field.

In the definition of Particle, it is the implicitly defined getters of fields $position$ and $velocity$ that provide concrete definitions for the inherited abstract methods from trait Moving.

In both singleton and dynamically parametric objects, implicitly defined getters and setters can be overridden by defining explicit getters and setters, using the modifiers `getter` and `setter`. For example, we alter the definition of Particle to include an explicit getter for field $velocity$ as follows:

```
object Particle(position : (ℝ Length)^3,
                velocity : (ℝ Velocity)^3)
  extends Moving
  getter velocity() = do
    print "velocity getter accessed"
    velocity
  end
end
```

The explicitly defined getter first prints a message and then returns the value of the field $velocity$. *Note that the final variable reference in this getter refers directly to the field $velocity$, not to the getter.* Only within an object definition can fields be accessed directly. In fact, even in an object definition, fields are only accessed directly when they are referred to as simple variables. In the following definition of Particle, the getter $velocity$ is recursively accessed through the special variable `self` (bound to the receiver of the method call):

```
object Particle(position : (ℝ Length)^3,
                velocity : (ℝ Velocity)^3)
  extends Moving
  getter velocity() = do
    print "velocity getter accessed"
    self.velocity
  end
end
```

Now, the result of a call to getter $velocity$ is an infinite loop (along with a lot of output).

2.15.1 Traits, Getters, and Setters

Although traits do not include field declarations, they can include getter and setter declarations, as in the following alternative definition of trait Moving:

```
trait Moving extends { Tangible, Object }
  getter position(): (ℝ Length)^3
  getter velocity(): (ℝ Velocity)^3
end
```

Now, an object with trait Moving must provide the definitions of *getters* (as opposed to ordinary methods) for $position$ and $velocity$. The advantage of this definition is that we can use getter notation on a variable of static type Moving:

```
v : Moving = ...
v.position
```

In fact, getters can be declared in a trait definition using field declaration syntax, as in the following example (which is equivalent to the definition of Moving above):

```
trait Moving extends { Tangible, Object }
  position: (ℝ Length)^3
  velocity: (ℝ Velocity)^3
end
```

Getter declarations in trait definitions must not include bodies. A getter definition can include the various modifiers allowed on a field declaration, with similar results. For example, if the modifier `settable` is used, then a setter is implicitly declared, in addition to a getter.

2.16 Features for Library Development

The language features introduced above are sufficient for the vast majority of applications programming. However, Fortress has also be designed to be a good language for *library* programming. In fact, much of the Fortress language as viewed by applications programmers actually consists of code defined in libraries, written in a small core language. By defining as much of the language as possible in libraries, our aim is to allow the language to evolve gracefully as new demands are placed on it. In this section, we briefly mention some of the features that make Fortress a good language for library development.

2.16.1 Generic Types and Static Parameters

As in other languages, Fortress allows types to be parametric with respect to other types, as well as other "static" parameters, including integers, booleans, dimensions, units, operators, and identifiers. We have already seen some standard parametric types, such as Array and Vector. Programmers can also define new traits, objects, and functions that include static parameters.

2.16.2 Specification of Locality and Data Distribution

Fortress includes a facility for expressing programmer intent concerning the distribution of large data structures at run time. This intent is expressed through special data structures called *distributions*. In fact, all arrays and matrices include distributions. These distributions are defined in the Fortress standard libraries. In most cases, the

default distributions will provide programmers with good performance over a variety of machines. However, in some circumstances, performance can be improved by overriding the default selection of distributions. Moreover, some programmers might wish to define these distributions, tailored for particular platforms and programs.

2.16.3 Operator Overloading

Operators in Fortress can be overloaded with new definitions. Here is an alternative definition of the $factorial$ function, defined as a postfix operator:

$$\texttt{opr}\ (n)! = \prod_{i \leftarrow 1:n} i$$

As with mathematical notation, Fortress allows operators to be defined as prefix, postfix, and infix. More exotic operators can even be defined as subscripting (i.e., applications of the operators look like subscripts), and as bracketing (i.e., applications of the operators look like the operands have been simply enclosed in brackets).

2.16.4 Definition of New Syntax

Fortress provides a facility for the definition of new syntax in libraries. This facility is useful for defining libraries for domain-specific languages, such as SQL, XML, etc. It is also used to encode some of the language constructs seen by applications programmers.

Part II

Fortress for Application Programmers

Chapter 3

Programs

A *program* consists of a finite sequence of Unicode 5.0 abstract characters. (Fortress does not distinguish between different code-point encodings of the same character.) In order to more closely approximate mathematical notation, certain sequences of characters are rendered as subscripts or superscripts, italicized or boldface text, or text in special fonts, according to the rules in Appendix E. For the purposes of entering program text, ASCII encodings of Unicode characters are also provided in Appendix E. Although much of the program text in this specification is rendered as formatted Unicode, some text is presented unformatted, or in its ASCII encoding, to aid in exposition.

A program is *valid* if it satisfies all *static constraints* stipulated in this specification. Failure to satisfy a static constraint is a *static error*. Only valid programs can be *executed*; the validity of a program must be checked before it is executed.

Executing a valid Fortress program consists of *evaluating expressions*. Evaluation of an expression may modify the *program state* yielding a *result*. A result is either a *value*, or an *abrupt completion*.

The characters of a valid program determine a sequence of *input elements*. In turn, the input elements of a program determine a sequence of *program constructs*, such as *trait declarations*, *object declarations*, *function declarations*, and *variable declarations*. Programs are developed, compiled, and deployed as *encapsulated upgradable components* as described in Chapter 22. We explain the structure of input elements, and of each program construct, in turn, along with accompanying static constraints. We also explain how the outcome of a program execution is determined from the sequence of constructs in the program.

Fortress is a block-structured language: A Fortress program consists of nested *blocks* of code. The entire program is a single block. Each component is a block. Any top-level declaration is a block, as is any function declaration. Several expressions are also blocks, or have blocks as parts of the expression, or both (e.g., a `while` expression is a block, and its body is a different block). See Chapter 13 for a discussion of expressions in Fortress. In addition, a local declaration begins a block that continues to the end of the smallest enclosing block unless it is a local function declaration and is immediately preceded by another local function declaration. (This exception allows overloaded and mutually recursive local function declarations.) Because Fortress is block structured, and because the entire program is a block, the smallest block that syntactically contains a program construct is always well defined.

Fortress is an expression-oriented language: In Fortress, statements are just expressions with type $()$. They do not evaluate to an interesting value. Typically, they are evaluated solely for their effects.

Fortress is a whitespace-sensitive language: Fortress has different contexts influencing the whitespace-sensitivity of expressions as described in Appendix G.

Chapter 4

Evaluation

The state of an executing Fortress program consists of a set of *threads*, and a *memory*. Communication with the outside world is accomplished through *input and output actions*. In this chapter, we give a general overview of the evaluation process and introduce terminology used throughout this specification to describe the behavior of Fortress constructs.

Executing a Fortress program consists of *evaluating* the expressions in each of its threads. Threads evaluate expressions by taking *steps*. We say evaluation of an expression *begins* when the first step is taken; each step yields a new expression. A step may *complete* the evaluation, in which case no more steps are possible on that expression, or it may result in an *intermediate expression*, which requires further evaluation. Intermediate expressions are generalizations of ordinary Fortress expressions: some intermediate expressions cannot be written in programs. We say that one expression is *dynamically contained* within a second expression if all steps taken in evaluating the first expression are taken between the beginning and completion of the second. A step may also have effects on the program state beyond the thread taking the step, for example, by modifying one or more locations in memory, creating new threads to evaluate other expressions, or performing an input or output action. Threads are discussed further in Section 4.4. The memory consists of a set of *locations*, which can be *read* and *written*. New locations may also be *allocated*. The memory is discussed further in Section 4.3. Finally, *input actions* and *output actions* are described in Section 4.6.

4.1 Values

A *value* is the result of normal completion of the evaluation of an expression. (See Section 4.2 for a discussion of completion of evaluation.) A value has a *type*, an *environment* (see Section 4.5), and a finite set of *fields*. Every value is an object (see Chapter 10); it may be a *value object*, a *reference object*, or a *function*. See Chapter 8 for a description of the types corresponding to these different values. The type of a value specifies the names and types of its fields, and which names must be bound in its environment (and the types of the locations they are bound to). The type also specifies the methods (including their bodies) of the object. Only trait types have methods other than those inherited from the type Object.

In a value object, each field is a value. In a reference object, each field is a location. Every field has a name, which may be an identifier or an index. Only values of type $\mathrm{LinearSequence}$ (defined in Section 40.1) or $\mathrm{HeapSequence}$ (defined in Section 40.3) have fields named by indices. Functions and the value $()$ have no fields. Every field in a value object is *immutable*. Reference objects may have both *mutable* and *immutable* fields. No two distinct values share any mutable field.

Values are constructed by top-level function declarations (see Section 12.1) and singleton object declarations (see Section 10.1), and by evaluating an object expression (see Section 13.9), a function expression (see Section 13.7), a local function declaration (see Section 6.4), a call to an object constructor (declared by an object declaration; see

Chapter 10), a literal (see Section 13.1), a `spawn` expression (see Section 13.24), an aggregate expression (see Section 13.28), or a comprehension (see Section 13.29). In the latter case, the constructed value is the result of the normal completion of such an evaluation.

4.2 Normal and Abrupt Completion of Evaluation

Conceptually, an expression is evaluated until it *completes*. Evaluation of an expression may *complete normally*, resulting in a value, or it may *complete abruptly*. Each abrupt completion has an associated value which caused the abrupt completion: the value is either an exception value that is thrown and uncaught or the exit value of an `exit` expression (described in Section 13.13). In addition to programmer-defined exceptions thrown explicitly by a `throw` expression (described in Section 13.25), there are predefined exceptions thrown by the Fortress standard libraries. For example, dividing an integer by zero (using the / operator) causes a $\mathrm{DivideByZeroException}$ to be thrown.

When an expression completes abruptly, control passes to the dynamically immediately enclosing expression. This continues until the abrupt completion is handled either by a `try` expression (described in Section 13.26) if an exception is being thrown or by an appropriately tagged `label` expression (described in Section 13.13) if an `exit` expression was executed. If abrupt completion is not handled within a thread and its outermost expression completes abruptly, the thread itself completes abruptly. If the main thread of a program completes abruptly, the program as a whole also completes abruptly.

4.3 Memory and Memory Operations

In this specification, the term *memory* refers to a set of abstract *locations*; the memory is used to model sharing and mutation. A location has an associated type, and a value of that type (i.e., the type of the value is a subtype of the type of the location); we say that the location *contains* its associated value. Unlike values, locations can have non-object trait types.

There are three kinds of operations that can be performed on memory: *allocation*, *reads*, and *writes*. Reads and writes are collectively called *memory accesses*. Intuitively, a read operation takes a location and returns the value contained in that location, and a write operation takes a location and a value of the location's type and changes the contents of the location so the location contains the given value. Accesses need not take place in the order in which they occur in the program text; a detailed account of memory behavior appears in Chapter 21.

Allocation creates a new location of a given type. Allocation occurs when a mutable variable is declared, or when a reference object is constructed. In the latter case, a new location is allocated for each field of the object. Abstractly, locations are never reclaimed; in practice, memory reclamation is handled by garbage collection.

A freshly allocated location is *uninitialized*. The type system and memory model in Fortress guarantee that an *initializing write* is performed to every location, and that this write occurs before any read of the location. Any location whose value can be written after initialization is *mutable*. Any location whose value cannot be written after initialization is *immutable*. Mutable locations include mutable variables and `settable` fields of a reference object. Immutable locations include non-`transient`, non-`settable` fields of a reference object.

4.4 Threads and Parallelism

There are two kinds of threads in Fortress: *implicit* threads and *spawned* (or *explicit*) threads. Spawned threads are objects created by the `spawn` construct, described in Section 13.24.

A thread may be in one of five states: *not started*, *executing*, *suspended*, *normally completed* and *abruptly completed*. We say a thread is *not started* after it is created but before it has taken a step; it has an expression that it needs to evaluate. This is only important for purposes of determining whether two threads are executing simultaneously; see Section 32.4. A thread is *executing* or *suspended* if it has taken a step, but has not completed; see below for the distinction between executing and suspended threads. A thread is *complete* if its expression has completed evaluation. If the expression completes normally, its value is the result of the thread.

Every thread has a *body* and an *execution environment*. The body is an intermediate expression, which the thread evaluates in the context of the execution environment; both the body and the execution environment may change when the thread takes a step. This environment is used to look up names in scope but bound in a block that encloses the construct that created the thread. The execution environment of a newly created thread is the environment of the thread that created the new thread.

In Fortress, a number of constructs are *implicitly parallel*. An implicitly parallel construct creates a *group* of one or more implicit threads. The implicitly parallel constructs are:

- **Tuple expressions:** Each element expression of the tuple expression is evaluated in a separate implicit thread (see Section 13.28).
- `also do` **blocks:** each sub-block separated by an `also` clause is evaluated in a separate implicit thread (see Section 13.12).
- **Method invocations and function calls:** The receiver or the function and each of the arguments is evaluated in a separate implicit thread (see Section 13.4, Section 13.5, and Section 13.6).
- `for` **loops, comprehensions, sums, generated expressions, and big operators:** Parallelism in looping constructs is specified by the generators used (see Section 13.17). Programmers should assume that generators other than the $sequential$ generator can execute each iteration of the body expression in a separate implicit thread. Reduction variables in `for` loops, and results returned from comprehensions and big operators correspond to reductions (see Section 13.15.1).
- **Extremum expressions:** Each guarding expression of the extremum expression is evaluated in a separate implicit thread (see Section 13.21).
- **Tests:** Each test is evaluated in a separate implicit thread (see Chapter 19).

Implicit threads run fork-join style: all threads in a group are created together, and they all must complete before the group as a whole completes. There is no way for a programmer to single out an implicit thread and operate upon it in any way; they are managed purely by the compiler, runtime, and libraries of the Fortress implementation. Implicit threads need not be scheduled fairly; indeed, a Fortress implementation may choose to serialize portions of any group of implicit threads, interleaving their operations in any way it likes. For example, the following code fragment may loop forever:

$r : \mathbb{Z}64 := 0$
$(r := 1, \texttt{while}\ r = 0\ \texttt{do end})$

If any implicit thread in a group completes abruptly, the group as a whole will complete abruptly as well. Every thread in the group will either not run at all, complete (normally or abruptly), or may be terminated early as described in Section 32.6. The result of the abrupt completion of the group is the result of one of the constituent threads that completes abruptly. After abrupt completion of a group of implicit threads, each reduction variable (see Section 13.15.1) may reflect an arbitrary subset of updates performed by the threads in the group. This means in general that reduction variables should not be accessed after abrupt completion. The exact behavior of reduction variables depends on the supporting generator structure and is described in Section 32.8.

Spawned thread objects are reference objects with four methods, val, $wait$, $ready$, and $stop$ (described in Section 32.6). None of these methods take arguments (other than the thread). The val method waits until the thread is complete; if the thread completes normally, the resulting value is returned. A spawned thread is not permitted to exit

(discussed in Section 13.13) to a surrounding label (the label is considered to be out of scope), but may fail to catch an exception and thus complete abruptly. When this happens, the exception is *deferred*. Any invocation of the val method on the spawned thread throws the deferred exception. If the spawned thread object is discarded, the exception will be silently ignored. In particular, if the result of a `spawn` expression is dropped, it is impossible to detect completion of the spawned thread or to recover from any deferred exceptions.

The $wait$ method waits until the thread is complete and then returns the void value. The $ready$ method does not wait, but instead returns a boolean value indicating whether the thread is complete. Invocations of these methods do not cause deferred exceptions to be thrown.

We say a spawned thread has been *observed to complete* after invoking the val or $wait$ methods, or when an invocation of the $ready$ method returns $true$.

There are three ways in which a thread can be suspended. First, a thread that begins evaluating an implicitly parallel construct is suspended until the thread group has completed. Second, a thread that invokes val or $wait$ is suspended until the spawned thread is complete. Finally, invoking the $abort$ function from within an `atomic` expression may cause a thread to suspend; see Section 32.3.

Threads in a Fortress program can perform operations simultaneously on shared objects. In order to synchronize data accesses, Fortress provides `atomic` expressions (see Section 13.23). Chapter 21 describes the memory model which is obeyed by Fortress programs.

Different implicit threads (even different iterations of the same loop body) may execute in very different amounts of time. A naively parallelized loop will cause processors to idle until every iteration finishes. The simplest way to mitigate this delay is to expose substantially more parallel work than the number of underlying processors available to run them. Load balancing can move the resulting (smaller) units of work onto idle processors to balance load.

The ratio between available work and number of threads is called *parallel slack* [3, 4]. With support for very lightweight threading and load balancing, slack in hundreds or thousands (or more) proves beneficial; very slack computations easily adapt to differences in the number of available processors. Slack is a desirable property, and the Fortress programmer should endeavor to expose parallelism where possible.

4.5 Environments

An *environment* maps *names* to values or locations. Environments are immutable, and two environments that map exactly the same names to the same values or locations are identical.

A program starts executing with an empty environment. Environments are extended with new mappings by variable, function, and object declarations, and functional calls (including calls to object constructors). After initializing all top-level variables and singleton objects as described in Section 22.6, the *top-level environment* for each component is constructed.

The environment of a value is determined by how it is *constructed*. For all but object expressions, function expressions and local function declarations, the environment of the constructed value is the top-level environment of the component in which the expression or declaration occurs. For object and function expressions and local function declarations, the environment of the constructed value is the lexical environment in which the expression or declaration was evaluated.

We carefully distinguish a spawned thread from its associated spawned thread object. In particular, note that the execution environment of a spawned thread, in which the body expression is evaluated, is distinct from the environment of the associated thread object, in which calls to the thread methods are evaluated.

4.6 Input and Output Actions

Certain functionals in Fortress perform primitive input/output (I/O) actions. These actions have an externally visible effect. Any functional which may perform an I/O action—either because it is a primitive action or because it invokes other functionals which perform I/O actions—must be declared with the `io` modifier.

Any primitive I/O action may take many internal steps; each step may read or write any memory locations referred to either directly or transitively by object references passed as arguments to the action. Each I/O action is free to complete either normally or abnormally. I/O actions may block and be prevented from taking a step until any necessary external conditions are fulfilled (input is available, data has been written to disk, and so forth).

Each I/O action taken by an expression is considered part of that expression's effects. The steps taken by an I/O action are considered part of the context in which an expression executes, in much the same way as effects of simultaneously-executing threads must be considered when describing the behavior of an expression. For example, we cannot consider two functionals to be equivalent unless the possible I/O actions they take are the same, given identical internal steps by each I/O action.

Chapter 5

Lexical Structure

A Fortress program consists of a finite sequence of Unicode 5.0 abstract characters. Every character in a program is part of an input element. The partitioning of the character sequence into input elements is uniquely determined by the characters themselves. In this chapter, we explain how the sequence of input elements of a program is determined from a program's character sequence.

This chapter also describes standard ways to *render* (that is, *display*) individual input elements in order to approximate conventional mathematical notation more closely. Other rules, presented in later chapters, govern the rendering of certain *sequences* of input elements; for example, the sequence of three input elements 1, $/$, and 2 may be rendered as $\frac{1}{2}$, and the sequence of three input elements a, `^`, and b may be rendered a^b. The rules of rendering are "merely" a convenience intended to make programs more readable. Alternatively, the reader may prefer to think of the rendered presentation of a program as its "true form" and to think of the underlying sequence of Unicode characters as "merely" a convenient way of encoding mathematical notation for keyboarding purposes.

Most of the program text in this specification is shown in rendered presentation form. However, sometimes, particularly in this chapter, unformatted code is presented to aid in exposition. In may cases the unformatted form is shown alongside the rendered form in a table, or following the rendered form in parentheses; as an example, consider the operator $\oplus$ (`OPLUS`).

5.1 Characters

A Unicode 5.0 abstract character is the smallest element of a Fortress program.[1] Many characters have standard *glyphs*, which are how these characters are most commonly depicted. However, more than one character may be represented by the same glyph. Thus, Unicode 5.0 specifies a representation for each character as a sequence of *code points*. Each character used in this specification maps to a single code point, designated by a hexadecimal numeral preceded by "U+". Unicode also specifies a name for each character;[2] when introducing a character, we specify its code point and name, and sometimes the glyph we use to represent it in this specification. In some cases, we use such glyphs without explicitly introducing the characters (as, for example, with the simple upper- and lowercase letters of the Latin and Greek alphabets). When the character represented by a glyph is unclear and the distinction is important, we specify the code point or the name (or both). The Unicode Standard specifies a *general category* for each character, which we use to describe sets of characters below.

[1] Note that a single Unicode abstract character may have multiple encodings. Fortress does not distinguish between different encodings of the same character that may be used in representing source code, and it treats canonically equivalent characters as identical. See the Unicode Standard for a full discussion of encoding and canonical equivalence.

[2] There are sixty-five "control characters", which do not have proper names. However, many of them have *Unicode 1.0 names*, or other standard names specified in the Unicode Character Database, which we use instead.

We partition the Unicode 5.0 character set into the following (disjoint) classes:

- *special non-operator characters*, which are:

U+0026	AMPERSAND	&	U+0027	APOSTROPHE	′
U+0028	LEFT PARENTHESIS	(	U+0029	RIGHT PARENTHESIS	)
U+002C	COMMA	,	U+002E	FULL STOP	.
U+0038	SEMICOLON	;	U+005C	REVERSE SOLIDUS	\
U+2026	HORIZONTAL ELLIPSIS	...	U+21A6	RIGHTWARDS ARROW FROM BAR	↦
U+2200	FOR ALL	∀	U+2203	THERE EXISTS	∃
U+27E6	MATHEMATICAL LEFT WHITE SQUARE BRACKET	⟦			
U+27E7	MATHEMATICAL RIGHT WHITE SQUARE BRACKET	⟧			

- *special operator characters*, which are

U+003A	COLON	:	U+003D	EQUALS SIGN	=
U+005B	LEFT SQUARE BRACKET	[	U+005D	RIGHT SQUARE BRACKET	]
U+005E	CIRCUMFLEX ACCENT	ˆ	U+007B	LEFT CURLY BRACKET	{
U+007C	VERTICAL LINE	|	U+007D	RIGHT CURLY BRACKET	}
U+2192	RIGHTWARDS ARROW	→	U+21D2	RIGHTWARDS DOUBLE ARROW	⇒

- *letters*, which are characters with Unicode general category Lu, Ll, Lt, Lm or Lo—those with Unicode general category Lu are *uppercase letters*—and the following *special letters*:

U+221E	INFINITY	∞	U+22A4	DOWN TACK	⊤	U+22A5	UP TACK	⊥

- *connecting punctuation* (Unicode general category Pc);
- *digits* (Unicode general category Nd);
- *prime characters*, which are:

U+2032	PRIME	U+2033	DOUBLE PRIME
U+2034	TRIPLE PRIME	U+2035	REVERSED PRIME
U+2036	REVERSED DOUBLE PRIME	U+2037	REVERSED TRIPLE PRIME

- *whitespace characters*, which are *spaces* (Unicode general category Zs) and the following characters:

U+0009	CHARACTER TABULATION	U+000A	LINE FEED
U+000C	LINE TABULATION	U+000C	FORM FEED
U+000D	CARRIAGE RETURN		
U+001C	INFORMATION SEPARATOR FOUR	U+001D	INFORMATION SEPARATOR THREE
U+001E	INFORMATION SEPARATOR TWO	U+001F	INFORMATION SEPARATOR ONE
U+2028	LINE SEPARATOR	U+2029	PARAGRAPH SEPARATOR

- *character literal delimiters*, which are:

U+0060	GRAVE ACCENT	‘
U+2018	LEFT SINGLE QUOTATION MARK	‘
U+2019	RIGHT SINGLE QUOTATION MARK	’

- *string literal delimiters*, which are:

U+0022	QUOTATION MARK	"
U+201C	LEFT DOUBLE QUOTATION MARK	“
U+201D	RIGHT DOUBLE QUOTATION MARK	”

- *ordinary operator characters*, enumerated (along with the special operator characters) in Appendix F, which include the following characters with code points less than U+007F:

U+0021	EXCLAMATION MARK	!	U+0023	NUMBER SIGN	#
U+0024	DOLLAR SIGN	$	U+0025	PERCENT SIGN	%
U+002B	PLUS SIGN	+	U+002D	HYPHEN-MINUS	−
U+003F	QUESTION MARK	?	U+0040	COMMERCIAL AT	@
U+002A	ASTERISK	*	U+002F	SOLIDUS	/
U+003C	LESS-THAN SIGN	<	U+003E	GREATER-THAN SIGN	>
U+007E	TILDE	~			

and most (but not all) Unicode characters specified to be *mathematical operators* (i.e., characters with code points in the range 2200-22FF) are operators in Fortress.

- *other characters*

Some other classes of characters, which overlap with the ones above, are useful to distinguish:

- *control characters* are those with Unicode general category Cc;
- *ASCII characters* are those with code points U+007F and below;
- *printable ASCII characters* are ASCII characters that are not control characters (i.e., with code points from U+0020 to U+007E);
- *word characters* are letters, digits, connecting punctuation, prime characters, and apostrophe;
- *restricted-word characters* are ASCII letters, ASCII digits, and the underscore character (i.e., ASCII word characters other than apostrophe);
- *hexadecimal digits* are the digits and the letters `A`, `B`, `C`, `D`, `E` and `F`;
- *operator characters* are special operator characters and ordinary operator characters;
- *special characters* are special non-operator characters and special operator characters;
- *enclosing characters* are the enclosing operator characters enumerated in Section F.1, left and right parenthesis characters, and mathematical left and right white square brackets;
- *operator combining characters*, which combine to form multicharacter enclosing operator tokens, are the following:

U+0028	LEFT PARENTHESIS	(	U+0029	RIGHT PARENTHESIS	)
U+002A	ASTERISK	*	U+002E	FULL STOP	.
U+002F	SOLIDUS	/	U+003C	LESS-THAN SIGN	<
U+003E	GREATER-THAN SIGN	>	U+005B	LEFT SQUARE BRACKET	[
U+005C	REVERSE SOLIDUS	\	U+005D	RIGHT SQUARE BRACKET	]
U+007B	LEFT CURLY BRACKET	{	U+007C	VERTICAL LINE	|
U+007D	RIGHT CURLY BRACKET	}			

Forbidden Characters

It is a static error for a Fortress program to contain any control character other than the above-listed whitespace characters, except that the character `SUBSTITUTE` (U+001A; also known as “control-Z”) is allowed and ignored if it is the last character of the program.

It is a static error for the following characters to occur outside a comment:

U+0009	CHARACTER TABULATION	U+000C	LINE TABULATION
U+001C	INFORMATION SEPARATOR FOUR	U+001D	INFORMATION SEPARATOR THREE
U+001E	INFORMATION SEPARATOR TWO	U+001F	INFORMATION SEPARATOR ONE

Thus, LINE FEED, FORM FEED and CARRIAGE RETURN are the only control characters—and the only whitespace characters other than spaces, LINE SEPARATOR, and PARAGRAPH SEPARATOR—outside of comments in a valid Fortress program.

It is a static error for REVERSE SOLIDUS (i.e., U+005C), any connecting punctuation other than LOW LINE (i.e., '_'; U+005F, also known as SPACING UNDERSCORE), any character in the "other characters" class above, or the following whitespace characters to appear outside of comments and string and character literals:

U+2000	EN QUAD	U+2003	EM SPACE	U+2006	SIX-PER-EM SPACE
U+2001	EM QUAD	U+2004	THREE-PER-EM SPACE	U+2007	FIGURE SPACE
U+2002	EN SPACE	U+2005	FOUR-PER-EM SPACE	U+2008	PUNCTUATION SPACE

5.2 Words

A *word* of a Fortress program is a maximal contiguous nonempty subsequence of word characters; that is, a word is one or more consecutive word characters delimited by characters other than word characters (or the beginning or end of the program). Recall that a word character is a letter, digit, connecting punctuation, prime character or apostrophe. Note that words partition the word characters in a program: every word character belongs to exactly one word; words do not overlap.

A *restricted word* is a maximal contiguous fragment of a word that has only restricted-word characters (i.e., ASCII letters, ASCII digits and underscore characters). Note that a restricted word might not be a word (i.e., if it is not delimited by non-word characters) and that the restricted words partition the restricted-word characters of a program.

5.3 Lines, Pages and Position

The sequence of characters in a Fortress program is partitioned into *lines* and *pages*, which are delimited by *line terminators* and *page terminators* respectively. A *page terminator* is an occurrence of the character FORM_FEED. A *line terminator* is an occurrence of any of the following:

- a CARRIAGE RETURN immediately followed by a LINE FEED,
- a CARRIAGE RETURN not immediately followed by LINE FEED,
- a LINE FEED not immediately preceded by CARRIAGE RETURN,
- a LINE SEPARATOR, or
- a PARAGRAPH SEPARATOR.

A character in a program is on page n (respectively line n) if there are $n-1$ page terminators (respectively line terminators) preceding that character in the program. A character is on line k of page n if it is on page n and there are $k-1$ line terminators preceding the character after the last preceding page terminator (or from the beginning of the program, if the character is on page 1).

A character is at line position k if there are $k-1$ characters preceding it and after the last preceding line terminator (or from the beginning of the program, if the character is on line 1). Note that a page terminator does *not* terminate a line, and hence the character immediately following a page terminator need not be at line position 1.

As discussed in Section 5.4, before any other processing, a Fortress program undergoes a process called *ASCII conversion*, which may replace sequences of ASCII characters with single (non-ASCII) characters. We expect that IDEs will typically display a program by rendering the converted sequence of characters rather than the actual input sequence. Thus, a program may appear to have fewer characters than it actually does. Nonetheless, the page, line and position of a character is based on the program before conversion.

If a character (or any other syntactic entity) x precedes another character y in the program, we say that x is *to the left of* y and that y is *to the right of* x, regardless of how they may appear in a typical rendered display of the program. Thus, it is always meaningful to speak, for example, of the left-hand and right-hand operands of a binary operator, or the left-hand side of an assignment expression.

5.4 ASCII Conversion

To facilitate interaction with legacy tools and particularly to aid in program entry, Fortress specifies an ASCII encoding for programs. For every valid Fortress program, there is an equivalent program that contains only ASCII characters.[3] To support this encoding, a Fortress program undergoes *ASCII conversion*, which produces an equivalent Fortress program. ASCII conversion is idempotent: converting a program that resulted from conversion results in the same program. Unless otherwise specified, all constraints and properties of Fortress programs stipulated in this specification apply to the programs after they have been converted. This section gives a high-level overview of ASCII conversion.

ASCII conversion consists of two steps. The first step consists of "pasting" words across line breaks, so that long identifiers and numerals can be split across lines. Identifiers may be very long in ASCII because many Unicode characters are encoded with long sequences of ASCII characters (the actual conversion to Unicode characters is done in the next step). Roughly speaking, in this step, two consecutive lines are pasted together if the first ends with an ampersand that is immediately preceded by a word character, and the second begins with an ampersand that is immediately followed by a word character.

The second step replaces certain restricted words, and sequences of operator and special characters, with single Unicode characters. Roughly speaking, if a restricted word is either the official Unicode 5.0 name with underscores in place of spaces and hyphens, or a specified alternative name, of some character that is not a printable ASCII character, then the restricted word is replaced by that character. In some cases, even a fragment of a restricted name may be replaced by a single character (most commonly a Greek letter). Some multicharacter sequences of ASCII operator and special characters are also replaced by non-ASCII operator or special characters; we call such a sequence *ASCII shorthand*. However, this replacement is *not* generally done within string literals, which instead provide *escape sequences* to get non-ASCII characters (see Section 5.10).

Precise descriptions of both these steps are given in Appendix E, including the rules for replacing fragments of restricted words and the specification of alternative names for non-operator characters. Alternative names for operator characters are given in Appendix F.

5.5 Input Elements and Scanning

After ASCII conversion, a Fortress program is broken up into *input elements* by a process called *scanning*.[4] That is, scanning transforms a Fortress program from a sequence of Unicode characters to a sequence of input elements. The characters that comprise an input element always appear contiguously in the input sequence. Every input element is a *whitespace element* (*comments* are whitespace elements) or a *token*. Every token is a *reserved word*, a *literal*, an *identifier*, an *operator token*, or a *special token*. There are five kinds of literals: boolean literals, character literals, string literals, the void literal, and numerals (i.e., numeric literals).

Conceptually, we can think of scanning as follows: First, the comments, character literals and string literals are identified. Then the remaining characters are divided into *words* (i.e., contiguous sequences of word characters: letters, digits, connecting punctuation, primes and apostrophes), whitespace characters, and other characters. In some cases, words separated by a single '.' or whitespace character (and no other characters) are joined to form a single

[3] See Appendix E for the precise notion of equivalence guaranteed by ASCII conversion.

[4] Fortress has a facility for defining new syntax, discussed in Chapter 36. However, except for that chapter, this specification generally ignores this facility, and describes the Fortress language only for programs that use the standard Fortress syntax.

numeral (see Section 5.13). Words that are not so joined are classified as reserved words, boolean literals, numerals, identifiers, or operator tokens, as described in later sections in this chapter. It is a static error if any word in a Fortress program cannot be classified as one of these. All remaining whitespace characters, together with the comments, form whitespace elements, which may be *line-breaking*. Finally, contiguous sequences of symbols (and a few other special cases) are checked to see whether they form multicharacter operator tokens, as described in Section 5.14, or the void literal (see Section 5.12). Every other character is a token by itself, either a special token (if it is a special character) or an operator token.

5.6 Comments

The character sequences "$(*$" and "$*)$" are referred to as *comment delimiters*. In a valid program, every occurrence of $(*$ (outside of string literals) is balanced by a subsequent occurrence of $*)$; it is a static error if comment delimiters are not properly balanced. All the characters between a balanced pair of comment delimiters, including the comment delimiters themselves, comprise a single input element, called a *comment*. Because comment delimiters are required to be balanced, comments may be nested: only comments defined by outermost balanced pairs of comment delimiters are considered input elements.

5.7 Whitespace Elements

Whitespace elements consist of comments, whitespace characters and ampersands. However, some whitespace characters and ampersands might not be part of whitespace elements. In particular, whitespace characters may occur within string literals, character literals and numerals, and ampersands may occur within string literals and character literals.

We need to distinguish *line-breaking whitespace* from *non-line-breaking whitespace*. We adopt the following terminology:

- A *line-terminating comment* is a comment that encloses one or more line terminators. All other comments are called *spacing comments*.
- *Spacing* refers to any nonempty contiguous sequence of spaces, `FORM FEED` characters and spacing comments.
- A *line break* is a line terminator or line-terminating comment that is not immediately preceded by an ampersand (U+0026), possibly with intervening spacing.
- *Whitespace* refers to any nonempty sequence of spacing, ampersands, line terminators, and line-terminating comments.
- *Line-breaking whitespace* is whitespace that contains at least one line break.

It is a static error if an ampersand occurs in a program (after ASCII conversion) unless it is within a character or string literal or a comment, or it is immediately followed by a line terminator or line-terminating comment (possibly with intervening spacing).

5.8 Special Reserved Words

The following tokens are *special reserved words*:

BIG	SI_unit	absorbs	abstract	also	api	as
asif	at	atomic	bool	case	catch	coerces
coercion	component	comprises	default	dim	do	elif
else	end	ensures	except	excludes	exit	export
extends	finally	fn	for	forbid	from	getter
hidden	ident	idiom	if	import	in	int
invariant	io	juxtaposition	label	largest	nat	object
of	or	opr	private	property	provided	requires
self	settable	setter	smallest	spawn	syntax	test
then	throw	throws	trait	transient	try	tryatomic
type	typecase	unit	value	var	where	while
widening	widens	with	wrapped			

The operators on units, namely cubed, cubic, inverse, per, square, and squared, are also special reserved words.

To avoid confusion, Fortress reserves the following tokens:

```
goto   idiom   public   pure   reciprocal   static
```

They do not have any special meanings but they cannot be used as identifiers.

5.9 Character Literals

A character literal consists of a sequence of characters enclosed in single quotation marks. The character that begins a character literal is called the literal's *opening mark*; the character that ends it is its *closing mark*. For convenience, the marks may be true typographical "curly" single quotation marks (U+2018 and U+2019), a pair of apostrophe characters (U+0027), or a "backquote" character (U+0060) and an apostrophe character. As discussed in Section 13.1, a character literal evaluates to a value of type Character (see Section 8.7), which represents an abstract Unicode character.

A left single quotation mark (U+2018) or a backquote begins a character literal unless it is within a comment, another character literal, or a string literal. An apostrophe (U+2018) begins a character literal unless it is within a comment, another character literal, or a string literal, or it is immediately preceded by a word character (i.e.., a letter, digit, connecting punctuation, prime, or apostrophe). In either case, the character literal ends with the nearest apostrophe or right single quotation mark *after* the first character following the opening mark. In particular, an apostrophe or right single quotation mark immediately following the opening mark of a character literal is *not* a closing mark. Thus, for example, the sequence `'''` is a single character literal with one enclosed character `'`.

It is a static error if any of the enclosed characters of a character literal is a line feed, form feed, carriage return, or any character forbidden outside comments in a Fortress program (see Section 5.1), or if there is exactly one enclosed character and it is a backslash (U+005C). It is also a static error if any of the enclosed characters is a string literal delimiter that is not immediately preceded by a backslash (i.e., an *unescaped* string literal delimiter). This last restriction is necessary to prevent ASCII conversion from changing the boundaries of string literals (see Appendix E).

The sequence of enclosed characters may be a single character (e.g., 'a', `'$'`, 'α', `'`$\oplus$`'`), a sequence of four or more hexadecimal digits identifying the code point of a Unicode character (e.g., '`001C`', '`FBAB`', '`1D11E`'), the official Unicode 5.0 name or an alternative name of a Unicode character with spaces and hyphens intact (e.g., '`PLUS-MINUS SIGN`'), or a *character-literal escape sequence*. The character-literal escape sequences, and the characters such character literals evaluate to, are:

`\b`	U+0008	`BACKSPACE`
`\t`	U+0009	`CHARACTER TABULATION`
`\n`	U+000A	`LINE FEED`
`\f`	U+000C	`FORM FEED`
`\r`	U+000D	`CARRIAGE RETURN`
`\"`	U+0022	`QUOTATION MARK`
`\\`	U+005C	`REVERSE SOLIDUS`
`\“`	U+201C	`LEFT DOUBLE QUOTATION MARK`
`\”`	U+201D	`RIGHT DOUBLE QUOTATION MARK`

It is a static error if the sequence of enclosed characters is not one of the kinds listed above. In particular, it is a static error if the hexadecimal digits enclosed do not correspond to the code point of a Unicode 5.0 abstract character.

Note that ASCII conversion is performed within character literals. Thus a character literal written as

```
'GREEK_CAPITAL_LETTER_LAMBDA'
```

is equivalent to a character literal written as

```
'Λ'
```

Note also that names for control characters are *not* converted during ASCII conversion, but they *are* permitted within character literals. Both the standard form of such names (i.e., with spaces and hyphens) and the form with spaces and hyphens replaced by underscore characters are permitted.

5.10 String Literals

A string literal is a sequence of characters enclosed in double quotation marks (for example, “$\pi\ r^2$” or `"Hello, world!"`). The character that begins a string literal is the literal’s *opening mark*; the character that ends it is its *closing mark*. For convenience, the opening and closing marks of a string literal may be either true typographical “curly” double quotation marks (U+201C and U+201D) or a pair of “neutral” double-quote characters. It is a static error if the opening and closing marks of a string literal do not “match”, that is, if one is “curly” and the other “neutral”. As discussed in Section 13.1, a string literal evaluates to a value of type String (see Section 8.7), which represents a finite sequence of abstract Unicode characters.

A left double quotation mark (U+201C) or “neutral” quotation mark (U+0022) begins a string literal unless it is within a comment, a character literal, or another string literal. It ends with the nearest following unescaped (i.e., not immediately preceded by backslash) right double quotation mark or “neutral” quotation mark. Therefore, it is not possible for a string literal to include an unescaped right double quotation mark or “neutral” quotation mark as an enclosed character. In addition, it is a static error for an unescaped left double quotation mark to be an enclosed character of a string literal.

Within a string literal, a backslash introduces an *escape sequence*, unless it is immediately preceded by an odd number of backslashes, in which case the backslash is itself escaped. There are three kinds of escape sequences recognized within a string literal: the character-literal escape sequences (see Section 5.9), *restricted-word escape sequences*, and *quoted-character escape sequences*.

A restricted-word escape sequence consists of an unescaped backslash immediately followed by a restricted word not beginning with a lowercase letter. After ASCII conversion, it is a static error for a string literal to contain any restricted-word escape sequence other than “`\BACKSPACE`”, “`\TAB`”, “`\NEWLINE`”, “`\FORM_FEED`”, or “`\RETURN`”.

A quoted-character escape sequence consists of an unescaped backslash immediately followed by an apostrophe and the sequence of characters immediately following the apostrophe up to and including the next apostrophe. Note that this kind of escape sequence looks just like a backslash followed by a character literal. It is a static error if such a sequence with the initial backslash removed would not be a valid character literal (see Section 5.9). Quoted-character

escape sequences are useful for ASCII "shorthands" that begin with a lowercase letter or are not restricted words, and when the escape sequence is immediately followed in the string by a letter, digit, or underscore. For example, `"\beta"` evaluates to a string containing a backspace character (indicated by `\b`) followed by the letters `e`, `t`, and `a`, but `"\'beta'"` evaluates to a string containing the single letter β.[5] As another example, `"x\AND\'NOT'y"` becomes `"x`$\wedge\neg$`y"`, but `"x\AND\NOTy"` is a static error, because the name "`NOTy`" does not correspond to a Unicode character. Also, the string `"Foo\'[\'T\'\]'"` becomes `"Foo`$[\![$`T`$]\!]$`"`.

It is a static error if an unescaped backslash is immediately followed by a character other than another backslash, an apostrophe, a string literal delimiter, or a restricted-word character. In addition, it is also a static error if an unescaped backslash within a string literal is followed by any lowercase letter other than '`b`', '`t`', '`n`', '`f`' or '`r`'. This rule preserves a certain level of compatibility with the C and Java programming languages.

Unlike elsewhere in a program, the enclosed characters of a string literal are *not* generally subject to the second phase of ASCII conversion (word pasting across line terminators still occurs within string literals). However, ASCII conversion may affect (and often replace) restricted-word and quoted-character escape sequences. Specifically, the restricted word of a restricted-word escape sequence and the sequence of characters between the apostrophes of a quoted-character escape sequence are subject to ASCII conversion. If that sequence of characters would be replaced by a single Unicode character, then the entire escape sequence is replaced by that character, unless the replacement character is a string literal delimiter, in which case the escape sequence is replaced by a backslash followed by the replacement character. For example, the string literal `"GAMMA"` is unchanged by ASCII conversion, but `"\GAMMA"` and `"\'GAMMA'"` are both changed to `"`Γ`"`. The string literal `"\\GAMMA"` is also unchanged by ASCII conversion, because the sequence `\\` is an escaped backslash; this string literal evaluates to a string value consisting of the six characters: `\`, `G`, `A`, `M`, `M`, and `A`. See Appendix E for a detailed discussion of the ASCII conversion process.

The formatting of identifiers and numerals described in Section 5.17 is not performed within string literals.

5.11 Boolean Literals

The boolean literals are $false$ and $true$.

5.12 The Void Literal

The void literal is $()$ (pronounced "void").

5.13 Numerals

Numeric literals in Fortress are referred to as *numerals*. Numerals may be either *simple* or *compound*.

A numeral may consist of several words separated by spaces or a '`.`' character. We adopt the following terminology:

- A *simple numeral fragment* is a word that begins with a digit. and consists of only letters and digits.
- A *compound numeral fragment* is a simple numeral fragment immediately followed by a '`.`' character immediately followed by a word that consists of only letters and digits.
- A sequence of characters is a *numeral prefix* if it is either a numeral fragment (simple or compound) or a numeral prefix immediately followed by a space immediately followed by a numeral fragment (simple or compound).

[5] Actually, the escape sequence `\'beta'` is replaced by β during ASCII conversion.

- A *radix specifier* consists of an underscore immediately followed either by a sequence of one or more digits or by the English name in all uppercase ASCII letters of an integer from 2 to 16.
- A *simple numeral base* is a word that consists only of letters and digits immediately followed by a radix specifier. The *radix* of a simple numeral base is the value corresponding to the sequence of digits in radix specifier interpreted in base ten, or the value of the integer whose English name is spelled out by the radix specifier. It is a static error if the radix is 0 or 1.
- A *compound numeral base* is a word that consists only of letters and digits immediately followed by a '.' character immediately followed by a simple numeral base.
- A *numeral* in a Fortress program is a maximal contiguous subsequence that is either a numeral base (simple or compound), a numeral prefix or a numeral prefix immediately followed by a space immediately followed by a numeral base (simple or compound).

A numeral is *simple* if it does not contain a '.' character; otherwise, the number is *compound*. It is a static error if a numeral contains more than one '.' character.

Here are some examples of numerals, shown in unformatted form:

```
17   7fff_16   0fff_SIXTEEN  1fab  10101101_2    3.14159265
0.a  FF_EIGHT  k12.52_24     P_50  DEAD.BEEF_16  PI_FIFTEEN
```

5.14 Operator Tokens

In this section, we describe how to determine the operator tokens of a program. Because operator tokens do not occur within comments, string literals and character literals, we henceforth in this section consider only characters that are outside these constructs.

An *operator word* of a program is a word that is not reserved, consists only of uppercase letters and underscores (no digits or non-uppercase letters), does not begin or end with an underscore, and has at least two different letters.

A *base operator* is an ordinary operator character,[6] the two-character sequence "`**`", an operator word, a (contiguous) sequence of two or more vertical-line characters (U+007C), or a multicharacter enclosing operator, as defined in Section 5.14.1. A base operator is *maximal* of a program if it is not contained within any other base operator of that program. It is a static error if two maximal base operators overlap (which is only possible if both are multicharacter enclosing operators). A *simple operator* is a maximal base operator that is an operator character, the two-character sequence "`**`", or an operator word.

A maximal base operator is an operator token unless it is a simple operator other than an enclosing or vertical-line operator character, and it is immediately preceded by '^' or immediately followed by '='. Such an operator is an *enclosing operator* if it is an enclosing operator character (see Section F.1) or a multicharacter enclosing operator; it is a *vertical-line operator* if it has only vertical-line operator characters (see Section F.2); otherwise, it is an *ordinary operator*.

If a simple operator is immediately preceded by '^' then the '^' followed by the simple operator is an operator token; such an operator token is called a *superscripted postfix operator*. In addition, "$\hat{}T$" is also a superscripted postfix operator provided that it is not immediately followed by a word character. It is a static error for a superscripted postfix operator to be immediately followed by a word character other than apostrophe.

Finally, if a simple operator is not immediately preceded by '^' and is immediately followed by '=' then the simple operator followed by the '=' is an operator token; such an operator token is called a *compound assignment operator*.

[6] The operator characters are enumerated in Appendix F, less the special operator characters listed in Section 5.1.

5.14.1 Multicharacter Enclosing Operators

The following multicharacter sequences (in which there must be no other characters, and particularly no whitespace) can be used as brackets as described below:

1. Any contiguous nonempty sequence of vertical-line characters is a vertical-line operator. Such an operator can be used in an enclosing pair matching itself.
2. Any of '(' or '[' or '{' may be immediately followed by any number of '`/`' characters or by any number of '`\`' characters. Such a token is a left bracket, and it is matched by the multicharacter token consisting of the same number and kind of '`/`' or '`\`' characters followed immediately by a matching ')' or ']' or '}', as appropriate. Thus, for example, "`(////`" and "`////)`" are matching left and right brackets respectively. (In the future, we may allow tokens with mixtures of '`/`' and '`\`', in which case the left and right brackets should match from outside in. But for now, such tokens are simply illegal.)
3. One or more '`<`' characters may be followed immediately by one or more '`|`' characters. Such a token is a left bracket, and it is matched by the multicharacter token consisting of the same number of '`|`' characters followed by as many '`>`' characters as there are '`<`' characters in the left bracket.
4. One or more '`<`' characters, or one or more '`|`' characters may be followed immediately by one or more '`/`' characters or by one or more '`\`' characters. Such a token is a left bracket, and it is matched by the multicharacter token consisting of the same number and *opposite* kind of '`/`' or '`\`' characters followed immediately by as many matching '`>`' or '`|`' characters as appropriate. Thus, for example, "`<<//`" matches "`\\>>`', and "`|\\\`" matches "`///|`". (As in case 2 above, we may allow tokens with mixtures of '`/`' and '`\`' in the future.)
5. Finally, any number of '`*`' (U+002A) or '`.`' (U+002E) characters may be placed within any of the above multicharacter sequences, except those that contain '(' or ')', as long as no '`*`' or '`.`' is the first or last character in the sequence, and no '`*`' or '`.`' characters are adjacent. The rule for matching is as above, except that in addition, the positions of the '`*`' and the '`.`' characters must match from the outside in.

Note that some of the character sequences described above cannot occur in programs after ASCII conversion. For example, "`||`" is converted to '‖' (U+2016). Note also that `[\ \]` (which are converted to ⟦ ⟧) are not operators; they play a special role in the syntax of Fortress, and their behavior cannot be redefined by a library.

5.14.2 Special Operators

Note that in the preceding discussion, a single operator character can be an operator token only if it is an ordinary operator character. In some cases, some of the special operator characters (and even some special non-operator characters) form part of an operator token. However, most of the special operator characters cannot be determined to be operators before parsing the program because they are also used for various parts of Fortress syntax. The one exception is '^': if an occurence of this character is not part of a superscripted postfix operator, then it is an operator token by itself: the special *superscripting operator*. This operator is always an infix operator, and it is a static error if it appears in a context in which a prefix or postfix operator is expected.

Every other special operator character, when not part of an operator token, is a special token that may be used as an operator. There is also a special `juxtaposition` operator (described in Section 16.7), which is also always infix, but this operator is a special reserved word rather than an operator token. Occurrences of this operator are determined by the Fortress grammar. The special reserved words `in`, cubed, cubic, inverse, per, square, and squared, are used as operators on units.

5.15 Identifiers

A word is an identifier if it begins with a letter and is not a reserved word, an operator, or all or part of a numeral.

5.16 Special Tokens

Every special character (operator or non-operator) that is not part of a token (or within a comment) as described above is a *special token* by itself. The special operator characters may be operators in the appropriate context.

5.17 Rendering of Fortress Programs

In order to more closely approximate mathematical notation, Fortress mandates standard rendering for various input elements, particularly for numerals and identifiers, as specified in this section. In the remainder of this specification, programs are presented formatted unless stated otherwise.

5.17.1 Fonts

Throughout this section, we refer to different fonts or styles in which certain characters are rendered, with names suggestive of their appearance.

- roman
- italic
- math (often identical to italic)
- script
- fraktur
- sans-serif
- italic sans-serif
- monospace
- double-struck

Additionally, the following fonts may be specified to be bold: roman, italic, script, fraktur, sans-serif, italic sans-serif.

However, a particular environment may substitute different fonts either because of local practice or because the desired fonts are not available.

5.17.2 Numerals

A numeral is rendered in roman type, with the radix, if present, as a subscript.

`27`	*is rendered as*	27
`7FFF_16`	*is rendered as*	$7\mathrm{FFF}_{16}$
`10101101_TWO`	*is rendered as*	10101101_{TWO}
`37X8E2_12`	*is rendered as*	$37\mathrm{X8E2}_{12}$
`deadbeef_SIXTEEN`	*is rendered as*	$\mathrm{deadbeef}_{\mathrm{SIXTEEN}}$
`dead.beef_16`	*is rendered as*	$\mathrm{dead.beef}_{16}$
`3.143159265`	*is rendered as*	3.143159265
`3.11037552_8`	*is rendered as*	3.1137552_{8}
`3.243f6b_16`	*is rendered as*	$3.\mathrm{243f6b}_{16}$
`11.00100100001111101101010_2`	*is rendered as*	$11.00100100001111101101010_{2}$

Note: the elegant way to write Avogadro's number is `6.02 TIMES 10^23`, which is not a single token but is a constant expression; its rendered form is 6.02×10^{23}.

5.17.3 Identifiers

Fortress has rather complicated rules for rendering an identifier; as in other parts of Fortress, the rules are complicated so that the simple cases will be very simple, but also so that difficult cases of interest will be possible.

It is conventional in mathematical notation to make use of variables, particularly single-letter variables, in a number of different fonts or styles, with italic being the most common, then boldface, and roman: a, $\mathbf{b}$, c. Frequently such variables are also decorated with accents and subscripts: $\bar{p}$, q', $\hat{r}$, $T_{\max}$, $\vec{\mathbf{u}}$, $\mathbf{v}_{\mathrm{x}}$, w_{17}, $\bar{z}'_{17}$. Fortress provides conventions for typing such variables using plain ASCII characters: for example, the unformatted presentations of these same variables are `a`, `_b`, `c_`, `p_bar`, `q'` or `q_prime`, `r_hat`, `T_max`, `_u_vec`, `_v_x`, `w17`, and `z17_bar'`. The rules are also intended to accommodate the typical use of multicharacter variable names for computer programming, such as $count$, $isUpperCase$, and $\mathrm{Boolean}$.

The most important rules of thumb are that simple variables are usually italic z (`z`), a leading underscore usually means boldface font $\mathbf{z}$ (`_z`), a trailing underscore usually means roman font z (`z_`), and a doubled capital letter means double-struck (or "blackboard bold") font $\mathbb{Z}$ (`ZZ`). However, mixed-case variable names that begin with a capital letter, which are usually used as names of types, are rendered in roman font even if there is no trailing underscore.

The detailed rules are described in Appendix D.

5.17.4 Other Formatting Rules

Special reserved words are rendered in monospace, except that the special reserved words that are used as operators on units, namely cubed, cubic, $\mathrm{inverse}$, per, square, and $\mathrm{squared}$, are rendered in roman type. Operator words are rendered in monospace. Comments are rendered in roman font. Any character within a character literal or string literal is rendered in monospace if possible. Delimiters of syntax expanders (described in Section 36.1) are rendered in monospace.

Chapter 6

Declarations

Declarations introduce *named entities*; we say that a declaration *declares* an entity and a name by which that entity can be referred to, and that the declared name *refers to* the declared entity. As discussed later, there is not a one-one correspondence between declarations and named entities: some declarations declare multiple named entities, some declare multiple names, and some named entities are declared by multiple declarations.

Some declarations contain other declarations. For example, a trait declaration may contain method declarations, and a function declaration may contain parameter declarations.

Syntactically, the positions in which a declaration may legally appear is determined by the nonterminal *Decl* in the Fortress grammar, defined in Appendix G.

6.1 Kinds of Declarations

Syntax:

Decl	::=	*TraitDecl*
	|	*ObjectDecl*
	|	*FnDecl*
	|	*VarDecl*
	|	*DimUnitDecl*
	|	*TypeAlias*
	|	*TestDecl*
	|	*PropertyDecl*

There are two kinds of declarations: *top-level declarations* and *local declarations*.

Top-level declarations occur at the top level of a program (or component), not within any other declaration or expression. A top-level declaration is one of the following:[1]

- trait declarations (see Chapter 9)
- object declarations (see Chapter 10), which may be parameterized or nonparameterized
- top-level function declarations (see Chapter 12), including top-level operator declarations (see Chapter 16)

[1] The Fortress component system, defined in Chapter 22, includes declarations of *components* and *APIs*. Because component names are not used in a Fortress program and API names are used only in qualified names and declarations of `import` and `export`, we do not discuss them in this chapter.

- top-level variable declarations (see Section 6.2)
- dimension declarations (see Chapter 18)
- unit declarations (see Chapter 18)
- top-level type aliases (see Section 8.9)
- test declarations (see Chapter 19)
- top-level property declarations (see Chapter 19)

Local declarations occur in another declaration or in some expression (or both). These may be one of the following:

- method declarations (see Section 9.2): these occur in trait and object declarations and object expressions
- field declarations (see Section 10.2): these occur in object declarations and object expressions, and include field declarations in the parameter list of an object declaration
- local function declarations (see Section 6.4): these occur in block expressions
- local variable declarations (see Section 6.3): these occur in block expressions
- local property declarations (see Chapter 19): these occur in trait and object declarations and object expressions
- labeled blocks (see Section 13.13)
- static-parameter declarations, which may declare type parameters, `nat` parameters, `int` parameters, `bool` parameters, `dim` parameters, `unit` parameters, `opr` parameters, or `ident` parameters (see Chapter 11): these occur in static-parameter lists of trait and parameterized object declarations, top-level type aliases, top-level function declarations, and method declarations
- hidden-type-variable declarations: these occur in `where` clauses of trait and object declarations, top-level function declarations, and method declarations
- type aliases in `where` clauses: these occur in `where` clauses of trait and object declarations, top-level function declarations, and method declarations
- (value) parameter declarations, which may be keyword-parameter declarations: these occur in parameter lists of parameterized object declarations, top-level function declarations, method declarations, local function declarations, and function expressions (but do not include nontransient parameter declarations in the parameter list of an object declaration, which are field declarations)

Some declarations are syntactic sugar for other declarations. Throughout this chapter, we consider declarations after they have been desugared. Thus, apparent field declarations in trait declarations are actually method declarations (as described in Section 9.2), and a dimension and unit declaration may desugar into several separate declarations (as described in Section 35.3). After desugaring, the kinds of declarations listed above are disjoint.

In addition to these explicit declarations, there are two cases in which names are declared implicitly: The special name `self` is implicitly declared as a parameter by some method declarations. See Section 9.2 for details about when `self` is implicitly declared. The name $result$ is implicitly declared by the ensures clause of a contract. See Section 12.4 for a discussion of contracts.

Trait declarations, object declarations, top-level type aliases, type-parameter declarations, and hidden-type-variable declarations are collectively called *type declarations*; they declare names that refer to *types* (see Chapter 8). Dimension declarations and `dim`-parameter declarations are *dimension declarations*, and unit declarations and `unit`-parameter declarations are *unit declarations*. Parameterized object declarations, top-level function declarations, method declarations, and local function declarations are collectively called *functional declarations*. Nonparameterized object declarations, top-level variable declarations, field declarations, local variable declarations, and (value) parameter declarations (including implicit declarations of `self`) are collectively called *variable declarations*. Nonparameterized object

declarations are also called *singleton object declarations*. Static-parameter declarations and hidden-type-variable declarations are collectively called *static-variable declarations*. Note that static-variable declarations are disjoint from variable declarations.

The groups of declarations defined in the previous paragraph are neither disjoint nor exhaustive. For example, labeled blocks are not included in any of these groups, and an object declaration is both a type declaration and either a function or variable declaration, depending on whether it is parameterized.

Most declarations declare a single name given explicitly in the declaration (though, as discussed in Section 7.1, they may declare this name in multiple namespaces). There is one exception: wrapped field declarations (described in Section 10.2) in object declarations and object expressions declare both the field name and names for methods provided by the declared type of the field.

Method declarations in a trait may be either *abstract* or *concrete*. Abstract declarations do not have bodies; concrete declarations, sometimes called *definitions*, do.

6.2 Top-Level Variable Declarations

Syntax:

VarDecl	::=	*Vars* (= | :=) *Expr*
	|	*VarWTypes*
	|	*VarWoTypes* : *TypeRef* ... [(= | :=) *Expr*]
	|	*VarWoTypes* : *SimpleTupleType* [(= | :=) *Expr*]
Vars	::=	*Var*
	|	(*Var* (, *Var*)$^+$)
Var	::=	*VarMod** *Id* [*IsType*]
VarWTypes	::=	*VarWType*
	|	(*VarWType* (, *VarWType*)$^+$)
VarWType	::=	*VarMod** *Id IsType*
VarWoTypes	::=	*VarWoType*
	|	(*VarWoType* (, *VarWoType*)$^+$)
VarWoType	::=	*VarMod** *Id*
SimpleTupleType	::=	(*TypeRef* , *TypeRefList*)
TypeRefList	::=	*TypeRef* (, *TypeRef*)*
VarMod	::=	`var` | *UniversalMod*
IsType	::=	: *TypeRef*

A *variable*'s name can be any valid Fortress identifier. There are four forms of variable declarations. The first form:

$id : \mathrm{Type} = expr$

declares id to be an immutable variable with static type Type whose value is computed to be the value of the expression $expr$. The static type of $expr$ must be a subtype of Type.

The second (and most convenient) form:

$id = expr$

declares id to be an immutable variable whose value is computed to be the value of the expression $expr$; the static type of the variable is the static type of $expr$.

The third form:

`var` $id : \mathrm{Type} = expr$

declares id to be a mutable variable of type Type whose initial value is computed to be the value of the expression $expr$. As before, the static type of $expr$ must be a subtype of Type. The modifier `var` is optional when '$:=$' is used instead of '$=$' as follows:

$[\texttt{var}]\ id : \mathrm{Type} := expr$

The first three forms are referred to as *variable definitions*. The fourth form:

$[\texttt{var}]\ id : \mathrm{Type}$

declares a variable without giving it an initial value (where mutability is determined by the presence of the `var` modifier). It is a static error if a variable is referred to before it has been given a value; an immutable variable is initialized by another variable declaration and a mutable variable is initialized by assignment. It is also a static error if an immutable variable is initialized more than once. Whenever a variable bound in this manner is assigned a value, the type of that value must be a subtype of its declared type. This form allows declaration of the types of variables to be separated from definitions, and it allows programmers to delay assigning to a variable before a sensible value is known.

In short, immutable variables are declared and initialized by '$=$' and mutable variables are declared and initialized by '$:=$' except when they are declared as the third form above with the modifier `var`.

All forms can be used with *tuple notation* to declare multiple variables together. Variables to declare are enclosed in parentheses and separated by commas, as are the types declared for them:

$(id[, id]^+) : (\mathrm{Type}[, \mathrm{Type}]^+)$

Alternatively, the types can be included alongside the respective variables, optionally eliding types that can be inferred from context (see Chapter 20 for a discussion of type inference in Fortress):

$(id[: \mathrm{Type}][, id[: \mathrm{Type}]]^+)$

Alternatively, a single type followed by '$\ldots$' can be declared for all of the variables:

$(id[, id]^+) : \mathrm{Type} \ldots$

This notation is especially helpful when a function application returns a tuple of values.

Here are some simple examples of variable declarations:

$\pi = 3.14159265358979323846264338327950288419716939937510820974944592307 8$

declares the variable π to be an approximate representation of the mathematical object π. It is also legal to write:

$\pi : \mathbb{R}64 = 3.14159265358979323846264338327950288419716939937510820974944592307 8$

This definition enforces that π has static type $\mathbb{R}64$.

In the following example, the declaration of the type of a variable and its definition are separated:

$\pi : \mathrm{Float}$
$\pi = 3.14159265358979323846264338327950288419716939937510820974944592307 8$

The following example declares multiple variables using tuple notation:

$\texttt{var}\ (x, y) : \mathbb{Z}64 \ldots = (5, 6)$

The following three declarations are equivalent:

$(x, y, z) : (\mathbb{Z}64, \mathbb{Z}64, \mathbb{Z}64) = (0, 1, 2)$
$(x: \mathbb{Z}64, y: \mathbb{Z}64, z: \mathbb{Z}64) = (0, 1, 2)$
$(x, y, z) : \mathbb{Z}64 \ldots = (0, 1, 2)$

Special syntax is provided for declaring variables to be parts of a matrix, as described in Section 6.5.

6.3 Local Variable Declarations

Syntax:

LocalVarDecl	::=	*LocalVars* (= | :=) *Expr*
	|	*LocalVarWTypes*
	|	*LocalVarWoTypes* : *TypeRef* ... [(= | :=) *Expr*]
	|	*LocalVarWoTypes* : *SimpleTupleType* [(= | :=) *Expr*]
LocalVars	::=	*LocalVar*
	|	(*LocalVar* (, *LocalVar*)$^+$)
LocalVar	::=	*LocalVarWType*
	|	*LocalVarWoType*
LocalVarWTypes	::=	*LocalVarWType*
	|	(*LocalVarWType* (, *LocalVarWType*)$^+$)
LocalVarWType	::=	[`var`] *Id IsType*
LocalVarWoTypes	::=	*LocalVarWoType*
	|	(*LocalVarWoType* (, *LocalVarWoType*)$^+$)
LocalVarWoType	::=	[`var`] *Id*
	|	*Unpasting*

Variables can be declared within block expressions (described in Section 13.11) via the same syntax as is used for top-level variable declarations (described in Section 6.2). A local variable declaration must not appear as the last of the block expression.

6.4 Local Function Declarations

Syntax:

LocalVarFnDecl	::=	*Id ValParam* [*IsType*] [*Throws*] = *Expr*

Functions can be declared within block expressions (described in Section 13.11) via the same syntax as is used for top-level function declarations (described in Chapter 12) except that locally declared functions must not include modifiers, contracts, static parameters, or `where` clauses (described in Chapter 11). As with top-level function declarations, locally declared functions in a single scope are allowed to be overloaded and mutually recursive. A local function declaration must not appear as the last of the block expression.

6.5 Matrix Unpasting

Syntax:

Unpasting	::=	[*L-Elt* (*Paste L-Elt*)*]
L-Elt	::=	*Id* [[*L-ArraySize*]]
	|	*Unpasting*
L-ArraySize	::=	*L-Extent* ($\times$*L-Extent*)*
L-Extent	::=	*Expr*
	|	*Expr* : *Expr*
	|	*Expr* # *Expr*
Paste	::=	(*Whitespace* | ;)$^+$

Matrix unpasting is an extension of variable declaration syntax as a shorthand for breaking a matrix into parts. On the left-hand side of a declaration, what looks like a matrix pasting of unbound variables is actually a declaration of

several new variables. This syntax serves to break the right-hand side into pieces and bind the pieces to the variables. Matrix unpastings are concise, eliminate several opportunities for fencepost errors, guarantee unaliased parts, and avoid overspecification of how the matrix should be taken apart.

The motivating example for matrix unpasting is cache-oblivious matrix multiplication. The general plan in a cache oblivious algorithm is to break the input apart on its largest dimension, and recursively attack the resulting smaller and more compact problems.

```
mm⟦nat m, nat n, nat p⟧(left : ℝ^{m×n}, right : ℝ^{n×p}, result : ℝ^{m×p}) : () = do
  case largest of
    1 ⇒ result_{0,0} += (left_{0,0} right_{0,0})
    m ⇒ [ lefttop
          leftbottom ] = left
        [ resulttop
          resultbottom ] = result
        t_1 = spawn do mm(lefttop, right, resulttop) end
        mm(leftbottom, right, resultbottom)
        t_1.wait()
    p ⇒ [ rightleft rightright ] = right
        [ resultleft resultright ] = result
      t_1 = spawn do mm(left, rightleft, resultleft) end
      mm(left, rightright, resultright)
      t_1.wait()
    n ⇒ [ leftleft leftright ] = left
        [ righttop
          rightbottom ] = right
        mm(leftleft, righttop, result)
        mm(leftright, rightbottom, result)
  end
end
```

In unpasting, the element syntax is slightly enhanced both to permit some specification of the split location and to receive information about the split that was performed. For example, perhaps only the upper left square of a matrix is interesting. The programmer can annotate bounds to the square unpasted element:

```
foo⟦nat m, nat n⟧(A : ℝ^{m×n}) : () = do
  if m < n then
    [ square_{m×m} rest ] = A
    ...
  elif m > n then
      [ square_{n×n}
        rest ] = A
      ...
  else (* A already square *)
      square = A
      ...
  end
end
```

The types of the elements of the newly declared matrix variables on the left-hand side of an unpasting are inferred (trivially) to be the type of the elements on the right-hand side.

If an unpasting into explicitly sized pieces does not exactly cover the right-hand-side matrix, an UnpastingException is thrown.

Each element of the left-hand-side of unpasting includes an optional *extent specification*. An extent specification $low \# num$ describes the indexing and the size of the given part of the matrix. The lower extent must be bound, either before the unpasting, or earlier (left-or-above) in the unpasting. For example, suppose that an algorithm chooses to break a matrix into 4 pieces, but retain the original indices for each piece:

$$
\begin{array}{l}
bar[\![\texttt{nat } p, \texttt{nat } q]\!](X : \mathbb{R}^{r_0 \# p \times c_0 \# q}) : () = \texttt{do} \\
\quad [\,\mathrm{A}_{r_0 \# m \quad \times c_0 \# n} \;\; \mathrm{B}_{r_0 \# m \quad \times c_0 + n \# q - n} \\
\quad\;\; \mathrm{C}_{r_0 + m \# p - m \times c_0 \# n} \;\; \mathrm{D}_{r_0 + m \# p - m \times c_0 + n \# q - n}\,] = X \\
\quad \ldots \\
\texttt{end}
\end{array}
$$

Unpasting does not directly support non-uniform decomposition, and does not provide any sort of constraint satisfaction between the extents of the parts. For example, the following decomposition is not legal because it constrains the split sizes to be equal with respect to unbound `nat` parameters:

$$
\begin{array}{l}
(* \text{ Not allowed! } *) \\
fubar[\![\texttt{nat } m, \texttt{nat } n]\!](X : \mathbb{R}^{m \times n}) : () = \texttt{do} \\
\quad (* \text{ p and q unbound } *) \\
\quad [\,\mathrm{A}_{p \times q} \;\; \mathrm{B}_{p \times q} \\
\quad\;\; \mathrm{C}_{p \times q} \;\; \mathrm{D}_{p \times q}\,] = X \\
\quad \ldots \\
\texttt{end}
\end{array}
$$

To get this effect, the programmer should compute the constrained values:

$$
\begin{array}{l}
fubar[\![\texttt{nat } m, \texttt{nat } n]\!](X : \mathbb{R}^{2m \times 2n}) : () = \texttt{do} \\
\quad [\,\mathrm{A}_{m \times n} \;\; \mathrm{B}_{m \times n} \\
\quad\;\; \mathrm{C}_{m \times n} \;\; \mathrm{D}_{m \times n}\,] = X \\
\quad \ldots \\
\texttt{end}
\end{array}
$$

Some non-uniform unpastings can be obtained with composition, which can be expressed either by repeated unpasting:

$$
\begin{array}{l}
unequalRows[\![\texttt{nat } m, \texttt{nat } n]\!](X : R^{4m \times 2n}) = \texttt{do} \\
\quad\;\; [\,c1_{4m \times n} \; c2_{4m \times n}\,] = X \\
\quad\;\; [\,\mathrm{A}_{m \times n} \\
\quad\;\;\; \mathrm{C}_{3m \times n}\,] = c1 \\
\quad\;\; [\,\mathrm{B}_{3m \times n} \\
\quad\;\;\; \mathrm{D}_{m \times n}\,] = c2 \\
\quad\;\; \ldots \\
\texttt{end}
\end{array}
$$

or simply by nesting matrices in the unpasting:

$$
\begin{array}{l}
unequalRows[\![\texttt{nat } m, \texttt{nat } n]\!](X : R^{2m \times 4n}) = \texttt{do} \\
\quad [[\,\mathrm{A}_{m \times n} \mathrm{B}_{m \times 3n}] \\
\quad\;\; [\,\mathrm{C}_{m \times 3n} \mathrm{D}_{m \times n}\,]\,] = X \\
\quad \ldots \\
\texttt{end}
\end{array}
$$

Chapter 7

Names

Names are used to refer to certain kinds of entities in a Fortress program. Names may be simple or qualified. A simple name is either an identifier or an operator. An operator may be an operator token, a special token corresponding to a special operator character. A qualified name consists of an API name followed by "`.`", followed by an identifier, where an API name consists of a sequence of identifiers separated by "`.`" tokens. Note that operators may not be qualified. Except in Section 7.3, we consider only simple names in this chapter.

Simple names are typically introduced by declarations, which bind the name to an entity. In some cases, the declaration is implicit. Every declaration has a scope, in which the declared name can be used to refer to the declared entity.

7.1 Namespaces

Fortress supports three namespaces, one for types, one for values, and one for labels. (If we consider the Fortress component system, there is another namespace for APIs.) These namespaces are logically disjoint: names in one namespace do not conflict with names in another.

Type declarations, of course, declare names in the type namespace. Function and variable declarations declare names in the value namespace. (This implies that object names are declared in both the type and value namespaces.) Labeled blocks declare names in the label namespace. Although they are not all type declarations, all the static-variable declarations declare names in the type namespace, as do dimension declarations. In addition, `unit` parameters, `nat` parameters, `int` parameters and `bool` parameters are also declared in the value namespace.

A reference to a name is resolved to the entity that the name refers to the namespace appropriate to the context in which the reference occurs. For example, a name refers to a label if and only if it occurs immediately following the special reserved word `exit`. It refers to a type if and only if it appears in a type context (described in Chapter 8). Otherwise, it refers to a value.

7.2 Reach and Scope of a Declaration

In this section, we define the *reach* and *scope* of a declaration, which determine where a declared name may be used to refer to the entity declared by the declaration. It is a static error for a reference to a name to occur at any point in a program at which the name is not in scope in the appropriate namespace (as defined below) unless the context is a value context and the name is the name of a field or method of a trait or object whose name is in scope in the type namespace.

We first define a declaration's *reach*. The reach of a labeled block is the block itself. A method declaration not in an object expression or declaration must be in the declaration of some trait T, and its reach is the declaration of T and any trait or object declarations or object expressions that extend T; that is, if the declaration of trait T contains a method declaration, and trait S extends trait T, then the reach of that method declaration includes the declaration of trait S. The reach of any other declaration is the smallest block strictly containing that declaration (i.e., not just the declaration itself). For example, the reach of a top-level declaration is the component containing that declaration, the reach of a field declaration is the enclosing object declaration or expression, the reach of a parameter declaration is the functional declaration or function expression in whose parameter list it occurs, and the reach of a local variable declaration is the smallest block in which that declaration occurs. We say that a declaration *reaches* any point within its reach.

It is a static error for two declarations with overlapping reaches to declare the same name other than `self` (even if the name is declared in different namespaces) unless one of the following conditions holds:

- both declarations are functional declarations with the same reach,
- both declarations are method declarations that occur in different trait declarations,
- one declaration is a field or keyword-parameter declaration whose reach is strictly contained in the reach of the other declaration, or
- one declaration is a method declaration that is provided by (i.e., occurs in or is inherited by) some trait or object declaration or object expression that is strictly contained in the reach of the other declaration.

If either of the first two conditions holds, or if one declaration is a field or method declaration that occurs in an object declaration or expression that inherits the other declaration (which therefore must be a method declaration), then the two declarations are *overloaded*, and subject to the restrictions on overloading (see Chapter 33).

If two declarations with overlapping reaches declare the same name in the same namespace, and the declarations are not overloaded, then at any point that their reaches overlap, one declaration *shadows* the other for that name in that namespace; we may omit the name and namespace when it is clear from context. Shadowing is permitted only in the following cases:

- In a trait or object declaration, any declaration in a block enclosing the declaration is shadowed if it declares a name of a field or method provided (declared or inherited) by the trait or object being declared.
- In a method declaration that does not give an explicit name other than `self` for the self parameter, any declaration of `self` (including implicit declarations) in a block enclosing the field or method declaration is shadowed.
- In the `ensures` clause of a contract, any declaration of $result$ in a block enclosing the `ensures` clause is shadowed.
- In a function or method declaration with keyword parameters, any declaration in a block enclosing the declaration is shadowed if it declares the name of any of the keyword parameters.

We say that a name is *in scope* in a namespace at any point in the program within the reach of a declaration that declares that name in that namespace unless one of the following conditions holds:

- the declaration is shadowed for the name in that namespace,
- the declaration is a variable declaration, the namespace is the value namespace, and the program point is in the initial-value expression of the declaration or an initial-value expression of another declaration that is in the smallest lexical block enclosing the declaration and lexically precedes the declaration in that block.

Note that the last condition applies to the method names declared by a wrapped field declaration.

We say that the *scope* of a declaration for a name in a namespace consists of those points at which the name is in scope for the namespace and the declaration is not shadowed for that name and that namespace. Again, when it is clear from context, we may omit the name and namespace.

7.3 Qualified Names

Fortress provides a component system in which the entities declared in a component are described by an API. A component may *import* APIs, allowing it to refer to these entities declared by the imported APIs. In some cases, references to these entities must be *qualified* by the API name. These qualified names can be used in any place that a simple name would be used had the entity been declared directly in the component rather than being imported. Note that qualified names are distinguished from simple names by the inclusion of a " . " token, so they never shadow, nor are they shadowed by, simple names. For further discussion on APIs and the component system, see Chapter 22.

Chapter 8

Types

Fortress provides several kinds of types: trait types, tuple types, arrow types, $\mathrm{BottomType}$, and other types provided in the Fortress standard libraries. Some types have names. Some types may be parameterized by types and values; we call these types *generic types*. Two types are identical if and only if they are the same kind and their names and arguments (if any) are identical. Types are related by several relationships as described in Section 8.1.

Syntactically, the positions in which a type may legally appear (*type context*) is determined by the nonterminal *TypeRef* in the Fortress grammar, defined in Appendix G.

8.1 Relationships between Types

Types in Fortress may be related by a subtyping relation, an exclusion relation, or a coercion.

A *subtyping* relation is reflexive, transitive, and antisymmetric, and is defined by the `extends` clause of trait declarations. Every expression has a *static type*. Every value has a *runtime type* (dynamic type). Fortress programs are checked before they are executed to ensure that if an expression e evaluates to a value v, the runtime type of v is a subtype of the static type of e. Sometimes we abuse terminology by saying that an expression has the runtime type of the value it evaluates to (see Section 4.2 for a discussion about evaluation of expressions). Thus, in the execution of a valid Fortress program, an expression's runtime type is always a subtype of its static type. We say that a value is *an instance of* its runtime type and of every supertype of its runtime type. Every type is a subtype of Object. For types T and U, we write $T \preceq U$ when T is a subtype of U, and $T \prec U$ when $T \preceq U$ and $T \neq U$.

Fortress defines an *exclusion* relation between types, which relates two disjoint types: no value can have a type that is a subtype of two types that exclude each other. The exclusion relation is irreflexive and symmetric, and is defined by the `excludes` and `comprises` clauses of trait declarations, and what is implied from these by the subtyping relation (including the fact that object trait types have no strict subtypes, and so exclude all types other than its supertypes). For example, suppose the following:

`trait` S `comprises` $\{U, V\}$ `end`
`trait` T `comprises` $\{V, W\}$ `end`
`object` U `extends` S `end`
`object` V `extends` $\{S, T\}$ `end`
`object` W `extends` T `end`

Because of the `comprises` clauses of S and T and the fact that U, V, and W are objects, S and T exclude each other. We write $T \lozenge U$ if T excludes U. If a type excludes another type, it excludes all its subtypes as well: $T \lozenge U \implies \forall T' \preceq T : T' \lozenge U$.

Fortress also allows *coercion* between types (see Chapter 17). A coercion from T to U is defined in the declaration of U. We write $T \to U$ if U defines a coercion from T. We say that T *can be coerced to* U, and write $T \rightsquigarrow U$, if U defines a coercion from T or any supertype of T: $T \rightsquigarrow U \iff \exists T' : T \preceq T' \wedge T' \to U$.

The Fortress type hierarchy is acyclic with respect to both subtyping and coercion relations except for the following:

- The trait Object is a single root of the type hierarchy and it forms a cycle as described in Chapter 23.
- There exists a bidirectional coercion between two tuple types if and only if they have the same sorted form.

8.2 Trait Types

Syntax:

TypeRef ::= *TraitType*

Traits are declared by trait declarations (described in Chapter 9). A trait has a *trait type* of the same name. A significant portion of Fortress types are trait types.

8.3 Object Trait Types

Named objects are declared by object declarations (described in Chapter 10) and anonymous objects are described by object expressions (described in Section 13.9). A named object has an *object trait type* of the same name and an anonymous object has an anonymous object trait type. An object trait type is a special kind of trait type. An object trait type extends all of the declared supertraits of the object. No other objects can have the object trait type and no trait type can extend an object trait type (i.e., an object trait type implicitly has an empty `comprises` clause).

8.4 Tuple Types

Syntax:

TypeRef ::= *TupleType*
TupleType ::= (*TypeRef* (, *TypeRef*)$^+$)
| ([*TypeRef* (, *TypeRef*)* ,] *TypeRef* ...)
| ([*TypeRef* (, *TypeRef*)* ,] [*TypeRef* ... ,] *Id* = *TypeRef* (, *Id* = *TypeRef*)*)

A tuple is an ordered sequence of keyword-value pairs. See Section 13.28 for a discussion of tuple expressions. A tuple type consists of a parenthesized, comma-separated list of element types where each element type is one of the following kinds:

- A plain type "T"
- A varargs type "$T \ldots$"
- A keyword-type pair "$identifier = T$"

The following restrictions apply: No two keyword-type pairs may have the same keyword. No keyword-type pair may precede a plain type. No varargs type may follow a keyword-type pair or precede a plain type. There must be at least one element type. If there is exactly one element type, it must be a varargs type or a keyword-type pair (because "(T)" is simply a type in parentheses, not a tuple type). Also, there can be at most one element type with varargs type.

An element type in tuple type X *corresponds* to one in tuple type Y if and only if:

- both are plain types in the same position,
- both are varargs types, or
- both are keyword-type pairs with the same keyword.

Every tuple type is a subtype of Object, and no other nontuple type. There is no other type that encompasses all tuples. Tuple types are covariant; a tuple type X is a subtype of tuple type Y if and only if:

- the correspondence between their element types is bijective;
- for each element type in X, the type in the element type is a subtype of the type in the corresponding element type in Y; and
- the keyword-type pairs in X and Y appear in the same order.

Note that, unlike record types in some other programming languages, the tuple type $(foo = P, bar = Q, baz = R)$ is not a subtype of $(foo = P, bar = Q)$, nor is (P, Q, R) a subtype of (P, Q). While $(\mathbb{Z}, \mathbb{Z})$ is not a subtype of $(\mathbb{Z} \ldots)$, there exists a coercion from the former to the latter (as described in Section 17.7).

For every tuple type X there is a tuple type X' that is the "sorted form" of the type, created by simply reordering the keyword-type pairs so that their keywords are in lexicographically ascending order. X' may be the same as X (as, for example, if X contains fewer than two keyword-type pairs). There is a coercion from tuple type X to tuple type Y if and only if X and Y have the same sorted form.

A tuple type excludes any nontuple type other than Object. Two tuple types exclude each other unless the correspondence between their element types is bijective. Two tuple types with a bijective correspondence between their element types exclude each other if either any type in an element type in one excludes the type in the corresponding element type in the other, or their keyword-type pairs do not appear in the same order.

Intersection of nonexclusive tuple types are defined elementwise; the intersection of nonexclusive tuple type X and Y is a tuple type with exactly corresponding elements, where the type in each element type is the intersection of the types in the corresponding element types of X and Y. Note that intersection of any exclusive types is $\mathrm{BottomType}$ as described in Section 8.6.

8.5 Arrow Types

Syntax:

TypeRef	::=	*ArrowType*
ArrowType	::=	*ArrowTypeRef* $\rightarrow$ *ArrowTypeRef* [*Throws*]
ArrowTypeRef	::=	*TypeRef* ($\times$ *TypeRef*)*
	|	*TypeRef* ^ *Number*

Functions can be passed as arguments and returned as values. See Chapter 12 for a discussion of functions. The types of function values are called *arrow types*. Every arrow type is a subtype of Object. Arrow types are not trait types. They cannot be extended by other trait types. Syntactically, an arrow type consists of the type of a parameter to the function followed by the token $\rightarrow$, followed by the type of a return value, and optionally a `throws` clause which specifies thrown checked exceptions. Here are some examples:

$(\mathbb{R}64, \mathbb{R}64) \rightarrow \mathbb{R}64$
$\mathbb{N} \rightarrow (\mathbb{N}, \mathbb{N})$ `throws` IOException
$(\mathrm{String}, \mathbb{N} \ldots, p = \mathrm{Printer}) \rightarrow \mathbb{N}$

Fortress supports alternative mathematical notations for arrow types whose parameter types or return types are tuple types:

- element types of a tuple type can be separated by the token $\times$ instead of by commas, with enclosing parentheses elided and
- element types of a tuple type that have the same type can be abbreviated using superscripts.

Here are some examples:

$\mathbb{R}64 \times \mathbb{R}64 \to \mathbb{R}64$
$\mathbb{N} \to \mathbb{N} \times \mathbb{N}$ `throws` IOException
$(\mathbb{N}, \mathbb{N}) \to \mathbb{N} \times \mathbb{N}$
$\mathbb{Z}^3 \to \mathbb{Z}$

Parameter types are contravariant but return types are covariant; arrow type "$A \to B$ `throws` C" is a subtype of arrow type "$D \to E$ `throws` F" if and only if:

- D is a subtype of A and
- B is a subtype of E and
- for all X in C, there exists Y in F such that X is a subtype of Y.

Coercion between arrow types are described in Section 17.7.

An arrow type excludes any nonarrow type other than Object. However, arrow types do not exclude other arrow types because of overloading as described in Chapter 33.

8.6 Bottom Type

Syntax:

TypeRef ::= $\mathrm{BottomType}$

Fortress provides a special *bottom type*, $\mathrm{BottomType}$, which is an uninhabited type. No value in Fortress has the bottom type; `throw` and `exit` expressions have the bottom type. The bottom type is a subtype of every type. Intersection of any exclusive types is the bottom type.

8.7 Types in the Fortress Standard Libraries

The Fortress standard libraries define simple standard types for literals such as $\mathrm{BooleanLiteral}[\![b]\!]$, $()$ (pronounced "void"), $\mathrm{Character}$, String, and $\mathrm{Numeral}[\![n, m, r, v]\!]$ for appropriate values of b, n, m, r, and v (See Section 13.1 for a discussion of Fortress literals). Moreover, there are several simple standard numeric types. These types are mutually exclusive; no value has more than one of them. Values of these types are immutable.

The numeric types share the common supertype Number. Fortress includes types for arbitrary-precision integers (of type $\mathbb{Z}$), their unsigned equivalents (of type $\mathbb{N}$), rational numbers (of type $\mathbb{Q}$), fixed-size representations for integers including the types $\mathbb{Z}8$, $\mathbb{Z}16$, $\mathbb{Z}32$, $\mathbb{Z}64$, $\mathbb{Z}128$, their unsigned equivalents $\mathbb{N}8$, $\mathbb{N}16$, $\mathbb{N}32$, $\mathbb{N}64$, $\mathbb{N}128$, floating-point numbers (described below), intervals (of type $\mathrm{Interval}[\![X]\!]$, abbreviated as $\langle\!|X|\!\rangle$, where X can be instantiated with any number type), and imaginary and complex numbers of fixed size (in rectangular form with types $\mathbb{C}n$ for $n = 16, 32, 64, 128, 256$ and polar form with type $\mathrm{Polar}[\![X]\!]$ where X can be instantiated with any real number type).

For floating-point numbers, Fortress supports types $\mathbb{R}32$ and $\mathbb{R}64$ to be 32 and 64-bit IEEE 754 floating-point numbers respectively, and defines two functions on types: $\mathrm{Double}[\![F]\!]$ is a floating-point type twice the size of the floating-point type F, and $\mathrm{Extended}[\![F]\!]$ is a floating-point type sufficiently larger than the floating-point type F to perform summations of "reasonable" size.[1]

[1] This formulation of floating-point types follows a proposal under consideration by the IEEE 754 committee.

The Fortress standard libraries also define other simple standard types such as Object, $\mathrm{Exception}$, $\mathrm{Boolean}$, and $\mathrm{BooleanInterval}$ as well as low-level binary data types such as $\mathrm{LinearSequence}$, $\mathrm{HeapSequence}$, and $\mathrm{BinaryWord}$. See Parts III and V for discussions of the Fortress standard libraries.

8.8 Intersection and Union Types

For every finite set of types, there is a type denoting a unique *intersection* of those types. The intersection of a set of types S is a subtype of every type $T \in S$ and of the intersection of every subset of S. There is also a type denoting a unique *union* of those types. The union of a set of types S is a supertype of every type $T \in S$ and of the union of every subset of S. Neither intersection types nor union types are first-class types; they are used solely for type inference (as described in Chapter 20) and they cannot be expressed directly in programs.

The intersection of a set of types S is equal to a named type U when any subtype of every type $T \in S$ and of the intersection of every subset of S is a subtype of U. Similarly, the union of a set of types S is equal to a named type U when any supertype of every type $T \in S$ and of the union of every subset of S is a supertype of U. For example:

```
trait S comprises {U, V} end
trait T comprises {V, W} end
trait U extends S excludes W end
trait V extends {S, T} end
trait W extends T end
```

because of the `comprises` clauses of S and T and the `excludes` clause of U, any subtype of both S and T must be a subtype of V. Thus, $V = S \cap T$.

Intersection types (denoted by $\cap$) possess the following properties:

- Commutativity: $T \cap U = U \cap T$.
- Associativity: $S \cap (T \cap U) = (S \cap T) \cap U$.
- Subsumption: If $S \preceq T$ then $S \cap T = S$.
- Preservation of shared subtypes: If $T \preceq S$ and $T \preceq U$ then $T \preceq S \cap U$.
- Preservation of supertype: If $S \preceq T$ then $\forall U.\ S \cap U \preceq T$.
- Distribution over union types: $S \cap (T \cup U) = (S \cap T) \cup (S \cap U)$.

Union types (denoted by $\cup$) possess the following properties:

- Commutativity: $T \cup U = U \cup T$.
- Associativity: $S \cup (T \cup U) = (S \cup T) \cup U$.
- Subsumption: If $S \preceq T$ then $S \cup T = T$.
- Preservation of shared supertypes: If $S \preceq T$ and $U \preceq T$ then $S \cup U \preceq T$.
- Preservation of subtype: If $T \preceq S$ then $\forall U.\ T \preceq S \cup U$.
- Distribution over intersection types: $S \cup (T \cap U) = (S \cup T) \cap (S \cup U)$.

8.9 Type Aliases

Syntax:

TypeAlias ::= `type` *Id* [*StaticParams*] = *TypeRef*

Fortress allows names to serve as aliases for more complex type instantiations. A *type alias* begins with the special reserved word `type` followed by the name of the alias type, followed by optional static parameters, followed by $=$, followed by the type it stands for. Parameterized type aliases are allowed but recursively defined type aliases are not. Here are some examples:

`type` $\mathrm{IntList} = \mathrm{List}[\![\mathbb{Z}64]\!]$
`type` $\mathrm{BinOp} = \mathrm{Float} \times \mathrm{Float} \to \mathrm{Float}$
`type` $\mathrm{SimpleFloat}[\![\texttt{nat}\ e, \texttt{nat}\ s]\!] = \mathrm{DetailedFloat}[\![\mathrm{Unity}, e, s, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true}]\!]$

All uses of type aliases are expanded before type checking. Type aliases do not define new types nor nominal equivalence relations among types.

Chapter 9

Traits

Traits are declared by trait declarations. Traits define new named types. A trait specifies a collection of *methods* (described in Section 9.2). One trait can extend others, which means that it inherits the methods from those traits, and that the type defined by that trait is a subtype of the types of traits it extends.

9.1 Trait Declarations

Syntax:

TraitDecl	::=	*TraitHeader* (*MdDecl* | *AbsFldDecl* | *PropertyDecl*)* `end`
TraitHeader	::=	*TraitMod** `trait` *Id* [*StaticParams*] [*Extends*] [*Excludes*] [*Comprises*] [*Where*]
Extends	::=	`extends` *TraitTypes*
Excludes	::=	`excludes` *TraitTypes*
Comprises	::=	`comprises` *MayTraitTypes*
TraitTypes	::=	*TraitType*
	|	{ *TraitTypeList* }
TraitTypeList	::=	*TraitType* (, *TraitType*)*
MayTraitTypes	::=	{}
	|	*TraitTypes*
TraitType	::=	*DottedId* [⟦*StaticArgList* ⟧]
	|	{ *TypeRef* $\mapsto$ *TypeRef* }
	|	⟨ *TypeRef* ⟩
	|	*TypeRef* [[*ArraySize*]]
	|	*TypeRef* [*MatrixSize*]
ArraySize	::=	*Extent* (, *Extent*)*
Extent	::=	*NatRef*
	|	*NatRef* # *NatRef*
MatrixSize	::=	*NatRef* ($\times$ *NatRef*)$^+$

Syntactically, a trait declaration starts with an optional sequence of modifiers followed by the special reserved word `trait`, followed by the name of the trait, an optional sequence of static parameters (described in Chapter 11), an optional set of *extended* traits, an optional set of *excluded* traits, an optional set of *comprises* on the trait, an optional `where` clause (described in Section 11.6), a list of method declarations, abstract field declarations, and property declarations (described in Section 19.6), and finally the special reserved word `end`.

Each of `extends`, `excludes`, and `comprises` clauses consist of the special reserved word `extends`, `excludes`, and `comprises` respectively followed by a set of trait references separated by commas and enclosed in braces '{' and

'}'. If such a clause contains only one trait, the enclosing braces may be elided. A trait reference is either a declared trait identifier or an abbreviated type for aggregate expressions (discussed in Section 13.28).

Every trait extends the trait Object. A trait with an `extends` clause extends every trait listed in its `extends` clause. If a trait T extends trait U, we call T a subtrait of U and U a supertrait of T. Extension is transitive; if T extends U it also extends all supertraits of U. Extension is also reflexive: T extends itself. The extension relation induced by a program is the smallest relation satisfying these conditions. This relation must form an acyclic hierarchy rooted at trait Object.

We say that trait T *strictly extends* trait U if and only if (*i*) T extends U and (*ii*) T is not U. We say that trait T *immediately extends* trait U if and only if (*i*) T strictly extends U and (*ii*) there is no trait V such that T strictly extends V and V strictly extends U. We call U an *immediate supertrait* of T and T an *immediate subtrait* of U.

A trait with an `excludes` clause excludes every trait listed in its `excludes` clause. If a trait T excludes a trait U, the two traits are mutually exclusive. No third trait can extend them both and neither can extend the other. A trait U can optionally have an `excludes` clause.

If a trait declaration of T includes a `comprises` clause, the trait must not be extended with immediate subtraits other than those that listed in its `comprises` clause. If a trait T has an empty `comprises` clause, no other traits can extend T.

For example, the following trait declaration:

```
trait Catalyst extends Object
  self.catalyze(reaction: Reaction): ()
end
```

declares a trait Catalyst with no modifiers, no static parameters, no `excludes` clauses, no `comprises` clauses, and no `where` clauses. Trait Catalyst extends a trait named Object. A single method (named *catalyze*) is declared, which has a parameter of type Reaction and the return type (). The special name `self` is explicitly declared as a parameter. See Section 9.2 for details about when `self` is implicitly declared, and to which entity it refers.

The following example trait:

```
trait Molecule comprises { OrganicMolecule, InorganicMolecule }
  mass(): Mass
end
```

comprises of two traits: OrganicMolecule and InorganicMolecule. Therefore, the following trait declaration is not allowed:

```
(* Not allowed! *)
trait ExclusiveMolecule extends Molecule end
```

Traits OrganicMolecule and InorganicMolecule may be exclusive:

```
trait OrganicMolecule extends Molecule excludes InorganicMolecule end
trait InorganicMolecule extends Molecule end
```

OrganicMolecule and InorganicMolecule exclude each other, even though only OrganicMolecule has an `excludes` clause. For example, the following trait declaration is not allowed:

```
(* Not allowed! *)
trait InclusiveMolecule extends { InorganicMolecule, OrganicMolecule } end
```

A trait is allowed to have multiple immediate supertraits. The following trait has two immediate supertraits:

```
trait Enzyme extends { OrganicMolecule, Catalyst } end
```

9.2 Method Declarations

Syntax:

MdDecl	::=	*AbsMdDecl*
	|	*MdDef*
AbsMdDecl	::=	[`abstract`] *MdMod** *MdHeader*
MdDef	::=	*MdMod** *MdHeader* $=$ *Expr*
	|	*Coercion*
MdMod	::=	`getter` | `setter` | *FnMod*
MdHeader	::=	[(*Id* | `self`) .]*Id* [*StaticParams*] ([*MdParams*]) [*IsType*] *FnClauses*
MdParams	::=	*MdParam*(, *MdParam*)*
	|	[*MdParam*(, *MdParam*)* ,] *Id* : *TypeRef* ...
	|	[*MdParam*(, *MdParam*)* ,] [*Id* : *TypeRef* ... ,] *MdParam* = *Expr* (, *MdParam* = *Expr*)*
MdParam	::=	*ParamId* [*IsType*]
	|	`self`
	|	*TypeRef*
ParamId	::=	*Id*
	|	_
IsType	::=	: *TypeRef*
FnClauses	::=	[*Throws*] [*Where*] [*Contract*]
Throws	::=	`throws` *MayTraitTypes*

A trait declaration contains a set of method declarations. Syntactically, a method declaration begins with an optional sequence of modifiers followed by the method's name optionally prefixed by a *self* parameter, optional static parameters (described in Chapter 11), the value parameter with its (optionally) declared type, an optional type of a return value, an optional declaration of thrown checked exceptions (discussed in Chapter 14), an optional `where` clause (discussed in Section 11.6), an optional contract for the method (discussed in Section 9.4), and finally an optional body expression preceded by the token $=$. A `throws` clause does not include naked type variables. Every element in a `throws` clause is a subtype of $\mathrm{CheckedException}$. A trait declaration may contain *coercions* discussed in Chapter 17.

Method declarations can include the following special modifiers:

`getter:` A method declaration with the modifier `getter` explicitly declares a getter method for a field, even in the absence of an actual field. If such a field exists, there is no implicit getter for the field. An explicitly declared getter method must take no arguments and return an appropriately typed result. A getter method must not throw any checked exception. Getter names may not overlap ordinary method names. A getter method must be invoked with the field access syntax:

$$expr.id$$

where id is the name of the getter method.

`setter:` A method declaration with the modifier `setter` explicitly declares a setter method for a field, even in the absence of an actual field. If such a field exists, there is no implicit setter for the field. An explicitly declared setter method must take a single argument—the value being set—and return $()$. A setter method must not throw any checked exception. Setter names may not overlap ordinary method names. A setter method must be invoked with the *assignment syntax*:

$$expr_1.id := expr_2$$

We say that a method declaration *occurs* in a trait declaration. A trait declaration *declares* a method declaration that occurs in that trait declaration. A trait declaration *inherits* method declarations from the declarations of its supertraits.

Note that a trait declaration inherits all method declarations declared by all of its supertraits–there's no real notion of overriding, just overloading (as discussed in Chapter 15). A trait declaration *provides* the method declarations that it declares or inherits.

There are two sorts of method declarations: *dotted method* declarations and *functional method* declarations. Syntactically, a dotted method declaration is identical to a function declaration, except that a special self parameter is provided immediately before the name of the method. When a method is invoked, the self parameter is bound to the object on which it is invoked. If no self parameter is provided explicitly, it is implicitly a parameter with name `self`. An explicit self parameter may be an identifier other than `self`, in which case `self` is not necessarily declared within that method.

A functional method declaration does not have a self parameter before the method name. Instead, it has a parameter named `self` at an arbitrary position in its parameter list. This parameter is not given a type and implicitly has the type of the enclosing declaration. Semantically, functional method declarations can be viewed as top-level functions. For example, the following overloaded functional methods f declared within a trait declaration A:

```
trait A
    f(self, t : T) = e1
    f(s : S, self) = e2
end
f(a, t)
```

may be rewritten as top-level functions as follows:

```
trait A
    internalF(t : T) = e1
    internalF(s : S) = e2
end
f1(a : A, t : T) = a.internalF(t)
f2(s : S, a : A) = a.internalF(s)
f1(a, t)
```

where $internalF$ is a freshly generated name. Functional method declarations may be overloaded with top-level function declarations. An abstract function declaration (described in Section 12.3) can be provided also for overloaded functional method declarations. See Chapter 15 for a discussion of overloaded functionals in Fortress.

A non-`self` self parameter can be used within nested object expressions (described in Section 13.9) to name the outer object in methods of the inner:

```
object
    m() = object
              notSelf.getOuterSelf() = self(* "self" declared in outer scope *)
              getInnerSelf() = self(* regular inner "self" *)
          end
end
```

When a method declaration includes a body expression, it is called a *method definition*. A method declaration that does not have its body expression is referred to as an *abstract method declaration*. An abstract method declaration may include the modifier `abstract`. An abstract method declaration may elide parameter names but parameter types cannot be omitted except for the self parameter.

Here is an example trait Enzyme which provides methods $mass$, $catalyze$, and $reactionSpeed$:

```
trait Enzyme extends { OrganicMolecule, Catalyst }
   reactionSpeed(): Speed
   catalyze(reaction) = reaction.accelerate(reactionSpeed())
```

```
end
```

Enzyme inherits the abstract method *mass* from OrganicMolecule, declares the abstract method *reactionSpeed*, and declares the concrete method *catalyze* which is inherited as an abstract method from its supertrait Catalyst.

9.3 Abstract Field Declarations

Syntax:

AbsFldDecl	::=	*AbsFldMod** *Id IsType*
AbsFldMod	::=	`hidden` | `settable` | `wrapped` | *UniversalMod*
UniversalMod	::=	`private` | `test`

Traits may also include abstract field declarations that are implicit declarations of abstract getter methods. Syntactically, an abstract field declaration consists of an optional sequence of modifiers followed by the field name, followed by the token `:`, and the type of the field.

By default, a field declaration implicitly declares a getter method for the field unless there is an explicit getter declared in the enclosing trait. An implicit getter method takes no arguments, has the same name as the field, and has a return type equal to the field type. When called, the implicit getter returns the value of the field when called.

Abstract field declarations can include the following special modifiers:

`hidden:` A field declaration with the modifier `hidden` has no implicit getter method.

`settable:` A field declaration with the modifier `settable` has an implicit setter method unless there is an explicit setter declared in the enclosing trait. An implicit setter method takes a parameter (with no default expression) whose type is the type of the field, and returns $()$. When called, the implicit setter rebinds the corresponding field to its argument. If a field declaration includes the modifier `settable` and `hidden`, only an abstract setter is declared. If a field declaration includes the modifier `hidden` without `settable`, it is a static error.

`wrapped:` If a field declaration of f has the modifier `wrapped` and the type of f is trait type T, and T is not a naked type variable, then the enclosing trait S implicitly includes "forwarding methods" for all methods in T that are also inherited from any supertrait of S. Each of these methods simply calls the corresponding method on the trait referred to by field f. If the trait declaration enclosing f explicitly declares a method m that conflicts with an implicitly declared forwarding method m', then the enclosing trait contains only method m, not m'. If the trait declaration enclosing f inherits a concrete method m that conflicts with an implicitly declared forwarding method m', then the enclosing trait contains only method m, not m'. Because wrapped fields do not change declarations of methods but change definitions of methods, they only affect implementations; APIs do not include wrapped fields.

For example, in the following declarations:

```
trait Dictionary⟦T⟧
  put(T): ()
  get(): T
end
trait WrappedDictionary⟦T⟧ extends Dictionary⟦T⟧
  wrapped val: Dictionary⟦T⟧
  get(): T
end
```

the parametric trait $\mathrm{WrappedDictionary}$ implicitly includes the following forwarding method:

$$put(x) = val.put(x)$$

If get were not explicitly declared in $\mathrm{WrappedDictionary}$, then $\mathrm{WrappedDictionary}$ would also include the forwarding method:

$$get() = val.get()$$

9.4 Method Contracts

Syntax:

Contract ::= [*Requires*] [*Ensures*] [*Invariant*]
Requires ::= `requires` *Expr*$^+$
Ensures ::= `ensures` (*Expr*$^+$ [`provided` *Expr*])$^+$
Invariant ::= `invariant` *Expr*$^+$

Method contracts consist of three optional clauses: a `requires` clause, an `ensures` clause, and an `invariant` clause. All three clauses are evaluated in the scope of the method body. See Section 12.4 for a discussion of each clause.

Method contracts are handled similarly to the manner described in [10]. In particular, substitutability under subtyping is preserved. For a call to a method m with receiver e, we use the term *static contract* of m to refer to a contract declared in the statically most applicable method declaration provided by the static type of e and the term *dynamic contract* of m to refer to a contract declared in the dynamically most applicable method declaration provided by the runtime type of e. Three exceptions may be thrown due to a method contract violation: $\mathrm{CallerViolation}$ is thrown when the `requires` clause of the static contract fails, $\mathrm{CalleeViolation}$ is thrown when the `ensures` or `invariant` clause of the dynamic contract fails, and $\mathrm{ContractHierarchyViolation}$ is thrown when the `requires` clause of the dynamic contract or the `ensures` or `invariant` clause of the static contract fails.

Evaluation of a call to a method m with receiver e proceeds as follows. First, e is evaluated to a value v with runtime type U. Let C and C' be the static and dynamic contracts of m, respectively. If the `requires` clause of C fails, a $\mathrm{CallerViolation}$ exception is thrown. Otherwise, if the `requires` clause of C' fails, a $\mathrm{ContractHierarchyViolation}$ exception is thrown. Otherwise, the `provided` subclauses of C and C' are evaluated. For every `provided` subclause that evaluates to $true$, the corresponding `ensures` subclause is recorded in a table E for later comparison. Similarly, the `invariant` clauses of C and C' are evaluated and the results are stored in E for later comparison. Then the body of m provided by U is evaluated. After evaluation of the body, all `ensures` subclauses of the dynamic contract recorded in E are checked to ensure that they evaluate to $true$, and all `invariant` clauses of the dynamic contract recorded in E are checked to ensure that they evaluate to values equal to the values they evaluated to before evaluation of the body. If any such check fails, a $\mathrm{CalleeViolation}$ exception is thrown. Otherwise, all `ensures` subclauses and `invariant` clauses of the static contract in E are checked. If any of these checks fails, a $\mathrm{ContractHierarchyViolation}$ exception is thrown.

9.5 Value Traits

Syntax:

TraitMod ::= `value` | *UniversalMod*

If a trait declaration has the modifier `value`, all subtraits of that trait must also have the modifier `value`, and all objects extending that trait are required to be value objects (described in Section 10.3). If a field declaration of a value trait has the modifier `settable`, the return type of its implicit setter method is the value trait type. If a value trait has

an explicit setter method, the setter must be an abstract method and its return type must be the value trait type. See Section 10.3 for a discussion of updating fields of value objects.

Chapter 10

Objects

An object is a *value object*, a *reference object*, or a *function object*: It is a function object if it has an arrow type, a reference object if it has an object trait type that is not declared with the `value` modifier (see Section 10.3), and a value object otherwise (i.e., if it has a tuple type, the type (), or an object trait type declared with the `value` modifier).

Value objects cannot have mutable fields, and they are completely determined by their type, environment and their fields: Value objects with the same type, environment and fields are indistinguishable. Thus, an implementation may freely copy value objects. Most objects with simple standard types, such as booleans, numeric literals, IEEE floating-point numbers, and integers are value objects. In contrast, reference objects are thought to "reside in memory", and are identified by an *object reference*. A new object reference is created whenever a reference object is constructed, so that reference objects constructed separately are always distinct. Reference objects include arbitrary-precision numbers and aggregates such as arrays, lists and sets. Function objects are immutable and have no fields. Identity is not well-defined for function objects, and attempting to check whether two functions are equivalent returns an approximate result. Section 10.4 describes object equivalence in further detail.

10.1 Object Declarations

Syntax:

ObjectDecl	::=	*ObjectHeader* (*MdDef* | *FldDef* | *PropertyDecl*)* `end`
ObjectHeader	::=	*ObjectMod** `object` *Id* [*StaticParams*] [([*ObjectParams*])] [*Extends*] *FnClauses*
ObjectMod	::=	*TraitMod*
ObjectParams	::=	*ObjectParam*(, *ObjectParam*)*
	|	[*ObjectParam*(, *ObjectParam*)* ,] *ObjectVarargs*
	|	[*ObjectParam*(, *ObjectParam*)* ,] [*ObjectVarargs* ,] *ObjectKeyword* (, *ObjectKeyword*)*
ObjectVarargs	::=	`transient` *Id* : *TypeRef* ...
ObjectKeyword	::=	*ObjectParam* = *Expr*
ObjectParam	::=	*FldMod** *PlainParam*
	|	`transient` *PlainParam*
FnClauses	::=	[*Throws*] [*Where*] [*Contract*]
Throws	::=	`throws` *MayTraitTypes*

Object declarations declare both object values and object trait types. Object declarations extend a set of traits from which they inherit methods. An object declaration inherits the concrete methods of its supertraits and must include a definition for every method declared but not defined by its supertraits. Especially, an object declaration must not include `abstract` methods (discussed in Section 22.3); it must define all `abstract` methods inherited from its supertraits. It is also allowed to define overloaded declarations of concrete methods inherited from its supertraits.

Syntactically, an object declaration begins with an optional sequence of modifiers followed by the special reserved word `object`, followed by the identifier of the object, optional static parameters (described in Chapter 11), optional value parameters, optional traits the object extends, an optional declaration of thrown checked exceptions (discussed in Chapter 14), an optional `where` clause (discussed in Section 11.6), an optional contract for the object (discussed in Section 12.4), a list of method declarations, field declarations, and property declarations (described in Section 19.6), and finally the special reserved word `end`. If an object declaration has no `extends` clause, the object implicitly extends only trait Object. A `throws` clause does not include naked type variables. Every element in a `throws` clause is a subtype of $\mathrm{CheckedException}$. If an object declaration has a contract, the contract is evaluated as function contracts (described in Section 12.4) when the object is created.

There are two kinds of object declarations: singleton object declarations and parametric object declarations. A singleton object declaration does not have any static or value parameter; it declares a sole, stand-alone object. There are two kinds of parametric object declarations: statically parametric objects and dynamically parametric objects. Statically parametric objects are parameterized by static parameters and dynamically parametric objects are parameterized by value parameters (possibly with static parameters). A dynamically parametric object declaration includes a constructor declaration and every call to the constructor of such an object with the same argument yields a new object. A statically parametric object declaration does not include a constructor declaration and every instantiation of such an object with the same argument yields the same singleton object. Initialization of parametric objects is entirely demand-driven as described in Section 22.6.

Each value parameter of a parameterized object declaration may be preceded by a sequence of field modifiers or the special modifier `transient`: A value parameter preceded by the modifier `transient` doesn't correspond to a field in an instantiation of the object. `transient` parameters are not in scope of the object's method declarations.

Fields can be also explicitly defined within a parameterized object declaration as within a singleton object declaration. All fields of an object are initialized before that object is made available to subsequent computations. Syntactically, method declarations in object declarations are identical to method declarations in trait declarations.

For example, the following empty list object extending trait List:

```
object Empty extends { List }
  first() = throw Error
  rest() = throw Error
  cons(x) = Cons(x, self)
  append(xs) = xs
end
```

has no fields and four methods.

Here is an example of a parameterized Cons object extending trait $\mathrm{List}[\![T]\!]$:

```
object Cons[\![T]\!](first : T, rest : List[\![T]\!])
  extends List[\![T]\!]
  cons(x) = Cons(x, self)
  append(xs) = Cons(first, rest.append(xs))
end
```

Note that this declaration implicitly introduces the "factory" function $\mathrm{Cons}[\![T]\!](\mathit{first} : T, \mathit{rest} : \mathrm{List}[\![T]\!]) : \mathrm{Cons}[\![T]\!]$ which is used in the body of the object declaration to define the cons and append methods. Multiple factory functions can be defined by overloading a parametric object with functions. For example: $\mathrm{Cons}(\mathit{first} : T) = \mathrm{Cons}(\mathit{first}, \mathrm{Empty})$.

10.2 Field Declarations

Syntax:

FldDef ::= *FldMod** *Id* [*IsType*] ($=$ | $:=$) *Expr*
FldMod ::= `var` | *AbsFldMod*

Fields are variables local to an object. They must not be referred to outside their enclosing object declarations. Field declarations in an object declaration are syntactically identical to top-level variable declarations (described in Section 6.2), with the same meanings attached to the form of variable declarations except that they have a different set of modifiers.

10.3 Value Objects

An object declaration with the modifier `value` declares a value object that is called in many languages a *primitive* value. The object trait type declared by a value object implicitly has the modifier `value`.

The fields of a `value` object are immutable; they cannot be changed directly or it is a static error. However, Fortress allows value objects to have settable fields as an abbreviation for constructing a new value object with a different value for one field. If a value object has a setter method or a subscripted assignment operator method (described in Section 34.7), then the return type of the method must be the value object trait type instead of $()$. When such a method is invoked, the receiver must itself be assignable, and the value returned by the method is assigned to the receiver.

For example, here is a value object $\mathrm{Complex}$ number:

```
value object Complex(settable real : Double, settable imaginary : Double = 0)
  opr +(self, other : Complex) = Complex(real + other.real, imaginary + other.imaginary)
end
```

When a mutable variable z:

```
var z : Complex = Complex(0)
```

updates its $imaginary$ field, the following syntax:

$$z.imaginary := v$$

can be used as an abbreviation for:

$$z := \mathrm{Complex}(z.real, v)$$

So the setter for the field $imaginary$ in $\mathrm{Complex}$ would do the work of constructing and returning $\mathrm{Complex}(z.real, v)$, and the assignment:

$$z.imaginary := v$$

would be construed to mean:

$$z := z.imaginary(v)$$

Note that modifying a settable field directly within the value object is not allowed. For example, the following:

$$imaginary := 3$$

within the $\mathrm{Complex}$ object means:

$$\texttt{self}.imaginary := 3$$

and because `self` is not mutable, the assignment is disallowed.

10.4 Object Equivalence

The trait Object defines the object equivalence operator $\equiv$. This operator is automatically defined for all objects; it is a static error for the programmer to override it. The $\equiv$ operator is used to decide whether its two arguments refer to "the same object" in the strictest sense possible. If the arguments have different dynamic types—including the instantiations of all static parameters—the result is always $false$. If both arguments are value objects with the same type, then the result is $true$ if and only if corresponding fields of the objects are themselves equivalent as defined by this operator; in particular, two binary words are strictly equivalent if and only if they contain the same bit pattern. If both arguments are object references, then the result is $true$ if and only if the two object references refer to the identical reference object (in implementation terms, occupying the same memory locations in the heap). If both arguments are functions, the result is $true$ only if the functions behave identically for any choice of type-correct arguments. Even if two functions behave identically, the fortress implementation is free to return $false$ when they are compared for object equivalence.

Chapter 11

Static Parameters

Trait, object, and functional declarations may be parameterized with *static parameters*. Static parameters are *static variables* listed in white square brackets ⟦ and ⟧ immediately after the name of a trait, object, or functional and they are in scope of the entire body of the declaration. Static parameters may be instantiated with static expressions discussed in Section 13.27. In this chapter, we describe the forms that these static parameters can take.

Syntax:

StaticParams ::= ⟦*StaticParamList*⟧
StaticParamList ::= *StaticParam* (, *StaticParam*)*

11.1 Type Parameters

Syntax:

StaticParam ::= *Id* [*Extends*] [`absorbs unit`]

Static parameters may include one or more type parameters. Syntactically, a type parameter consists of an identifier followed by an optional `extends` clause, followed by an optional "`absorbs unit`" clause (described in Section 35.4). If a type parameter does not have an `extends` clause, it has an implicit "`extends` Object" clause.

Type parameters are instantiated with types such as traits, tuple types, and arrow types (See Chapter 8 for a discussion of Fortress types). We use the term *naked type variable* to refer to an occurrence of a type variable as a stand-alone type (rather than as a parameter to another type). Type parameters can appear in any context that an ordinary type can appear, except that a naked type variable must not appear in the `extends` clause of a trait or object declaration nor as the type of a `wrapped` field (discussed in Section 10.2).

Here is a parameterized trait List:

```
trait List⟦T⟧
  first(): T
  rest(): List⟦T⟧
  cons(T): List⟦T⟧
  append(List⟦T⟧): List⟦T⟧
end
```

11.2 Nat and Int Parameters

Syntax:

StaticParam ::= `nat` *Id*
| `int` *Id*

Static parameters may include one or more `nat` and `int` parameters. Syntactically, a `nat` parameter consists of the special reserved word `nat` followed by an identifier. An `int` parameter consists of the special reserved word `int` followed by an identifier. These parameters are instantiated at runtime with numeric values. A `nat` parameter may be used to instantiate other `nat` parameters, or to appear in any context that a variable of type $\mathbb{N}$ can appear, except that it cannot be assigned to. An `int` parameter may be used to instantiate other `int` parameters, or to appear in any context that a variable of type $\mathbb{Z}$ can appear, except that it cannot be assigned to.

For example, the following function f:

$f[\![\texttt{nat}\ n]\!](x : \mathrm{Length}^{2n}) : \mathrm{Length}^{n} = sqrt(x)$

declares a `nat` parameter n, which appears in both the parameter type and return type of f.

11.3 Bool Parameters

Syntax:

StaticParam ::= `bool` *Id*

Static parameters may include one or more `bool` parameters. Syntactically, a `bool` parameter consists of the special reserved word `bool` followed by an identifier. These parameters are instantiated at runtime with boolean values. They may be used to instantiate other `bool` parameters, or to appear in any context that a variable of type $\mathrm{Boolean}$ can appear, except that they cannot be assigned to.

For example, the following `coercion` declared in the trait $\mathrm{Boolean}$:

```
trait Boolean
  coercion [[bool b]](x: BooleanLiteral[[b]])
end
```

declares a `bool` parameter b, which appears in the parameter type. See Chapter 24 for a full declaration of $\mathrm{Boolean}$.

11.4 Dimension and Unit Parameters

Syntax:

StaticParam ::= `dim` *Id*
| `unit` *Id* [: *DimRef*] [`absorbs unit`]

Static parameters may include one or more `dim` and `unit` parameters. Syntactically, a `dim` parameter begins with the special reserved word `dim` followed by an identifier. A `unit` parameter begins with the special reserved word `unit` followed by an identifier, optionally followed by the token : and a dimension, and the unit is thereby restricted to be a unit of the specified dimension. A `unit` parameter may include the clause " `absorbs unit` "; the meaning of this is described in Section 35.4. A `dim` parameter is allowed to appear in any context that a dimension can appear. A `unit` parameter is allowed to appear in any context that a unit can appear.

For example, here is a function that is parameterized with a unit:

$sqrt[\![\texttt{unit}\ U]\!](x: \mathbb{R}64\ U^2): \mathbb{R}64\ U = numericalsqrt(x/U^2)\ U$

11.5 Operator and Identifier Parameters

Syntax:

StaticParam ::= `opr` *Op*
| `ident` *Id*

Static parameters may include one or more operator symbols and identifiers denoting method names. Syntactically, an operator parameter begins with the special reserved word `opr` followed by an operator symbol. An identifier parameter begins with the special reserved word `ident` followed by an identifier.

Unlike other static parameters, operator and identifier parameters may be used in both type context and value context. The following example operator parameter $\odot$:

```
trait UnaryOperator⟦T extends UnaryOperator⟦T, ⊙⟧, opr ⊙⟧
  opr ⊙(self): T
end
```

is declared as a static parameter of UnaryOperator, instantiated as a static argument, and declared as an operator method.

Operator and identifier parameters may be freely intermixed with other static parameters. For example, the following trait HasLeftZeroes:

```
trait HasLeftZeroes⟦T extends HasLeftZeroes⟦T, ⊙, isLeftZero⟧, opr ⊙, ident isLeftZero⟧
    extends { BinaryOperator⟦T, ⊙⟧ }
  isLeftZero(): Boolean
  property ∀(a: T, b: T) a.isLeftZero() →: ((a ⊙ b) = a)
end
```

is parameterized with a type parameter T, an operator parameter $\odot$, and an identifier parameter $isLeftZero$. Many interesting examples are described in Section 37.3.

11.6 Where Clauses

Syntax:

Where ::= `where` { *WhereClauseList* }
WhereClauseList ::= *WhereClause* (, *WhereClause*)*
WhereClause ::= *Id Extends*
| *TypeAlias*
| *NatConstranint*
| *IntConstranint*
| *BoolConstraint*
| *UnitConstraint*
| *TypeRef* `coerces` *TypeRef*
| *TypeRef* `widens` *TypeRef*

Static parameters may have constraints placed on them in a `where` clause. A `where` clause begins with the special reserved word `where`, followed by a sequence of static parameter constraints enclosed in braces { and }, and separated by commas.

A `where` clause may introduce new static variables, i.e., identifiers for types and other static entities that may not be static parameters. We use the term `where`*-clause variables* to refer to static variables that are not also static parameters. The `where`-clause variables must be bound in a `where` clause.

A static parameter constraint is one of the following forms:

- a trait constraint consisting of the identifier of a naked type variable, followed by the special reserved word `extends` followed by a set of trait references which may include naked type variables,
- a type alias (described in Section 8.9),
- an arithmetic constraint,
- a boolean constraint,
- a unit equality constraint,
- a `coerces` constraint (described in Section 17.2), or
- a `widens` constraint (described in Section 17.2).

A `where` clause may include mutually recursive constraints. All static variables in a trait, object, or functional declaration must occur either as a static parameter or as a `where`-clause variable. Appendix A.2 describes a Fortress core calculus with `where` clauses.

Trait declarations are allowed to extend other instantiations of themselves. For example, we can write:

```
trait C⟦S⟧ extends C⟦T⟧
  where {S extends T, T extends Object}
end
```

In this declaration, for every subtype S of T, $C[\![S]\!]$ is a subtype of $C[\![T]\!]$. Effectively, we have expressed the fact that the static parameter S of C is covariant.

Trait declarations need not have any static parameters in order to have a `where` clause. For example, the following trait declaration is legal:

```
trait C extends D⟦T⟧
  where {T extends Object}
end
```

In this declaration, trait C is a subtrait of *every* instantiation of parametric trait D. Thus, trait C has all of the methods of every instantiation of D. By thinking of the declaration this way, we can see what restrictions we need to impose on the trait C in order for it to be sensible. If trait C inherits a method declaration that refers to T, it really contains infinitely many methods (one for each instantiation of T). However, instantiations of the `where`-clause variables are not explicit from the program text as static parameters are. It must be possible to infer which method is referred to at the call site. If there is not enough information to infer which method is called, type checking rejects the program and requires more type information from the programmer. Programmers always can provide more type information by using type ascription as described in Section 13.30.

Object or functional declarations may include `where` clauses. Here is an example declaration of an Empty list:

```
object Empty extends List⟦T⟧ where {T extends Object}
  first() = throw Error
  rest() = throw Error
  cons(x) = Cons(x, self)
  append(xs) = xs
end
```

where Cons is declared in Section 10.1.

Chapter 12

Functions

Functions are values that have arrow types described in Section 8.5. Each function takes exactly one argument, which may be a tuple, and returns exactly one result, which may be a tuple. A function may be declared as top level or local as described in Section 6.1. Fortress allows functions to be *overloaded* (as described in Chapter 15); there may be multiple function declarations with the same function name in a single lexical scope. Functions can be passed as arguments and returned as values. Single variables may be bound to functions including overloaded functions.

12.1 Function Declarations

Syntax:

FnDecl	::=	*AbsFnDecl*
	|	*FnDef*
FnDef	::=	*FnMod** *FnHeader* = *Expr*
FnMod	::=	`atomic` | `io` | *UniversalMod*
FnHeader	::=	*Id* [*StaticParams*] *ValParam* [*IsType*] *FnClauses*
ValParam	::=	*ParamId*
	|	([*ValParams*])
ParamId	::=	*Id*
	|	_
ValParams	::=	*PlainParam*(, *PlainParam*)*
	|	[*PlainParam*(, *PlainParam*)* ,] *Id* : *TypeRef* ...
	|	[*PlainParam*(, *PlainParam*)* ,] [*Id* : *TypeRef* ... ,] *PlainParam* = *Expr* (, *PlainParam* = *Expr*)*
PlainParam	::=	*ParamId* [*IsType*]
	|	*TypeRef*
FnClauses	::=	[*Throws*] [*Where*] [*Contract*]
Throws	::=	`throws` *MayTraitTypes*

Syntactically, a function declaration consists of an optional sequence of modifiers followed by the name of the function, optional static parameters (described in Chapter 11), the value parameter with its (optionally) declared type, an optional type of a return value, an optional declaration of thrown checked exceptions (discussed in Chapter 14), an optional `where` clause (discussed in Section 11.6), an optional contract for the function (discussed in Section 12.4), and finally an optional body expression preceded by the token $=$. A `throws` clause does not include naked type variables. Every element in a `throws` clause is a subtype of $\mathrm{CheckedException}$. When a function declaration includes a body expression, it is called a *function definition*. Function declarations can be mutually recursive.

Function declarations can include the following special modifiers:

`atomic:` A function with the modifier `atomic` acts as if its entire body were surrounded in an `atomic` expression discussed in Section 13.23.

`io:` Functions that perform externally visible input/output actions are said to be `io` functions. An `io` function must not be invoked from a non-`io` function.

A function takes exactly one argument, which may be a tuple. When a function takes a tuple argument, we abuse terminology by saying that the function takes multiple arguments. Value parameters cannot be mutated inside the function body.

A function's value parameter consists of a parenthesized, comma-separated list of bindings where each binding is one of:

- A plain binding "$identifier$" or "$identifier : T$"
- A varargs binding "$identifier : T \ldots$"
- A keyword binding "$identifier = e$" or "$identifier : T = e$"

When the parameter is a single plain binding without a declared type, enclosing parentheses may be elided. The following restrictions apply: No two bindings may have the same identifier. No keyword binding may precede a plain binding. No varargs binding may follow a keyword binding or precede a plain binding. Note that it is permitted to have a single plain binding, or to have no bindings. The latter case, "()", is considered equivalent to a single plain binding of the ignored identifier "_" of type (), that is, "$(\_ : ())$". Also, there can be at most one varargs binding.

A parameter declared by keyword binding is called a *keyword parameter*; a keyword parameter must be declared with a *default* expression, which is used when no argument is bound to the parameter explicitly. Syntactically, the default expression is specified after an $=$ sign. The default expression of a parameter x of function f is evaluated each time the function is called without a value provided for x at the call site. All parameters occurring to the left of x are in scope of its default expression. All parameters following x must include default expressions as well; x is in scope of their default expressions and the body of the function. When an argument is passed explicitly for a keyword parameter, that argument must be passed as a *keyword argument*. (See Section 12.2.) If no type is declared for a keyword parameter, the type is inferred from the static type of its default expression.

A parameter declared by varargs binding is called a *varargs parameter*; it is used to pass a variable number of arguments to a function as a single heap sequence. The type of a varargs parameter is $\mathrm{HeapSequence}[\![T]\!]$ where T is the type mentioned in (or inferred for) that binding. See Section 40.3 for a discussion of $\mathrm{HeapSequence}$. Note that the type of a varargs parameter cannot be omitted. If a function does not have a varargs parameter then the number of arguments is fixed by the function's type. Note that a varargs parameter is not allowed to have a default expression.

For example, here is a simple polymorphic function for creating lists:

$\mathrm{List}[\![T$ `extends` $\mathrm{Object}, \mathtt{nat}\ \mathit{length}]\!](\mathit{rest} : T[\mathit{length}]) =$ `do`
  `if` $\mathit{length} = 0$ `then` Empty
  `else` $\mathrm{Cons}(\mathit{rest}_0, \mathrm{List}(\mathit{rest}_{1:(\mathit{length}-1)}))$
  `end`
`end`

The following function:

$$\mathit{swap}(x : \mathrm{Object}, y : \mathrm{Object}) : (\mathrm{Object}, \mathrm{Object}) = (y, x)$$

has no static parameters, throws no checked exceptions, and has no contract. It takes a tuple of two elements of type Object and returns a tuple of two values. Namely, it returns its arguments in reverse order. If the return type is elided, it is inferred to be the static type of the body. The following declaration of swap has the same return type as the above declaration:

$swap(x : \text{Object}, y : \text{Object}) = (y, x)$

Similarly, function parameter type can often be inferred from the body of the function. When a type can be inferred for a parameter from the body of the function, that parameter type need not be declared explicitly. Thus, the following declaration of $swap$ has the same parameter type and return type as the above declarations:

$swap(x, y) = (y, x)$

See Chapter 20 for a discussion of type inference in Fortress.

The following function $wrap$:

$wrap(xs, ys = xs) = [xs\ ys]$

returns an array containing its parameters. If a value for only the parameter xs is given to $wrap$ at a call site, the value of xs is bound to ys as well, and an array that contains xs as both of its indices is returned.

12.2 Function Applications

Fortress provides overloaded functions (as described in Chapter 15); there may be multiple function declarations with the same function name in a single lexical scope. Thus, we need to determine which function declaration are applicable to a function application.

If a function's argument type is $()$, then function declarations with the following forms of parameter lists are considered to be applicable:

- $()$ which means the same thing as $(\_ : ())$
- $(x : ())$ which is something programmers don't ordinarily write
- $(x : T \ldots)$

In the last case, x is bound to an empty $\text{HeapSequence}[\![T]\!]$.

If a function's argument type A is neither $()$ nor a tuple type, then function declarations with the following forms of parameter lists are considered to be applicable:

- $(x : T)$ where A is a subtype of T
- $(x : T \ldots)$ where A is a subtype of T

In the last case, x is bound to a $\text{HeapSequence}[\![T]\!]$ of length 1, containing the actual argument value.

If a function's argument type A is a tuple type, then function declarations with the following forms of parameter lists are considered to be applicable:

- $(x : T)$ where A is a subtype of T
- $(x : T \ldots)$ where A is a subtype of T
- a parameter list with no varargs binding, provided that
  - type A has exactly as many plain types as the parameter list has plain bindings, and
  - for every keyword-type pair (described in Section 8.4) in A, the parameter list has a binding with the same keyword, and
  - for every element type in A, the type in the element type is a subtype of the type of the corresponding binding in the parameter list.

- a parameter list with a varargs binding, provided that
  - type A has at least as many plain types as the parameter list has plain bindings, and
  - for every keyword-type pair in A, the parameter list has a binding with the same keyword, and
  - for every element type in A, the type in the element type is a subtype of the type of the corresponding binding in the parameter list—but if there is no corresponding binding, then the type in the element type must be a subtype of the type in the varargs binding.

In the latter case, the parameter named by the identifier in the varargs binding is bound to a $\mathrm{HeapSequence}[\![T]\!]$ that contains, in order, all the values of the tuple that did not correspond to plain bindings, followed by all the values in the varargs $\mathrm{HeapSequence}$ of the tuple, if any.

When an argument is passed explicitly for a keyword parameter, that argument must be passed as a *keyword argument*. Syntactically, a keyword argument is a keyword-value pair "$identifier = e$". Keyword parameters not explicitly bound are bound to their default values. If a parameter that has no default value is not explicitly bound to an argument, it is a static error. Because a keyword-value pair shares a syntax with an *equality expression*, we provide rules for disambiguation in Section 13.28.1.

When a function is called (See Section 13.6 for a discussion of function call expressions), explicit arguments are evaluated in parallel, keyword parameters not explicitly bound are bound to their default values sequentially, and the body of the function is evaluated in a new environment, extending the environment in which it is defined with all parameters bound to their arguments.

If the application of a function f ends by calling another function g, tail-call optimization must be applied. Storage used by the new environments constructed for the application of f must be reclaimed.

Here are some examples:

$sqrt(x)$
$\arctan(y, x)$
$makeColor(red = 5, green = 3, blue = 43)$
$processString(s, start = 5, finish = 43)$

If the function's argument is not a tuple, then the argument need not be parenthesized:

$sqrt\, 2$
$\sin x$
$\log \log n$

Here are a few varargs examples:

$f(x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z} \ldots) = \langle x, y, \langle q \mid q \leftarrow z \rangle \rangle$

$f(1, 2)$	*returns*	$\langle 1, 2, \langle\rangle\rangle$
$f(1, 2, 3, 4)$	*returns*	$\langle 1, 2, \langle 3, 4\rangle\rangle$
$f(1, 2, [3\ 4] \ldots)$	*returns*	$\langle 1, 2, \langle 3, 4\rangle\rangle$
$f(1, 2, 3, 4, [5\ 6] \ldots)$	*returns*	$\langle 1, 2, \langle 3, 4, 5, 6\rangle\rangle$
$f(1, 2, 3, 4, 17 \# 3 \ldots)$	*returns*	$\langle 1, 2, \langle 3, 4, 17, 18, 19\rangle\rangle$
$f(1, [3\ 4] \ldots)$	*declaration not applicable*	

12.3 Abstract Function Declarations

Syntax:

AbsFnDecl ::= *FnMod** *FnHeader*
| *Name* : *ArrowType*

A function declaration may be separated from its definition. An *abstract function declaration* can be provided for overloaded function definitions. When the parameter type of an abstract function declaration includes a type that is declared with a `comprises` clause, it is a static error if the corresponding function definitions do not cover every immediate subtype of the type.

Syntactically, an abstract function declaration is a function declaration without a body. Parameter names may be elided but parameter types cannot be omitted. Additionally, when a function's type is not parameterized, Fortress provides an alternative mathematical notation for an abstract function declaration: function name followed by the token `:`, followed by an arrow type.

For example, after the following abstract function declaration:

$$printMolecule(\text{Molecule}) : ()$$

where trait Molecule is defined as follows:

`trait` Molecule `comprises` $\{\text{OrganicMolecule}, \text{InorganicMolecule}\}$ `end`

the programmer could write:

$$printMolecule(molecule : \text{Molecule}) = \ldots$$

or could write:

$$printMolecule(molecule : \text{OrganicMolecule}) = \ldots$$
$$printMolecule(molecule : \text{InorganicMolecule}) = \ldots$$

For the latter, the programmer must provide a definition for every immediate subtype of Molecule, or it is a static error.

12.4 Function Contracts

Syntax:

Contract	::=	[*Requires*] [*Ensures*] [*Invariant*]
Requires	::=	`requires` *Expr*$^+$
Ensures	::=	`ensures` (*Expr*$^+$ [`provided` *Expr*])$^+$
Invariant	::=	`invariant` *Expr*$^+$

Function contracts consist of three optional clauses: a `requires` clause, an `ensures` clause, and an `invariant` clause. All three clauses are evaluated in the scope of the function body.

The `requires` clause consists of a sequence of expressions of type Boolean. The `requires` clause is evaluated during a function call before the body of the function. If any expression in a `requires` clause does not evaluate to $true$, a CallerViolation exception is thrown.

The `ensures` clause consists of a sequence of `ensures` subclauses. Each such subclause consists of a sequence of expressions of type Boolean, optionally followed by a `provided` subclause. A `provided` subclause begins with the special reserved word `provided` followed by an expression of type Boolean. For each subclause in the `ensures` clause of a contract, the `provided` subclause is evaluated immediately after the `requires` clause during a function call (before the function body is evaluated). If a `provided` subclause evaluates to $true$, then the expressions preceding this `provided` subclause are evaluated after the function body is evaluated. If any expression evaluated after function evaluation does not evaluate to $true$, a CalleeViolation exception is thrown. The expressions preceding the `provided` subclause can refer to the return value of the function. A $result$ variable is implicitly bound to a return value of the function and is in scope of the expressions preceding the `provided` subclause. The implicitly declared $result$ shadows any other declaration with the same name in scope.

The `invariant` clause consists of a sequence of expressions of *any type*. These expressions are evaluated before and after a function call. For each expression e in this sequence, if the value of e when evaluated before the function call is not equal to the value of e after the function call, a $\mathrm{CalleeViolation}$ exception is thrown.

Here are some examples:

$factorial(n : \mathbb{Z}64)$ `requires` $n \geq 0 =$
  `if` $n = 0$ `then` 1
  `else` $n\, factorial(n-1)$
  `end`

$mangle(input : \mathrm{List})$ `ensures` $sorted(result)$ `provided` $sorted(input) =$
  `if` $input \neq \mathrm{Empty}$
  `then` $mangle(first(input))$
    $mangle(rest(input))$
  `end`

Overloaded function contracts are handled similarly with method contracts described in Section 9.4. In particular, substitutability is preserved: the statically most applicable function to a call should be substitutable with the dynamically most applicable function to the call. For a call of function f, we use the term *static contract* of f to refer to a contract declared in the statically most applicable function declaration and the term *dynamic contract* of f to refer to a contract declared in the dynamically most applicable function declaration. Three exceptions may be thrown due to an overloaded function contract violation: $\mathrm{CallerViolation}$ is thrown when the `requires` clause of the static contract fails, $\mathrm{CalleeViolation}$ is thrown when the `ensures` or `invariant` clause of the dynamic contract fails, and $\mathrm{ContractOverloadingViolation}$ is thrown when the `requires` clause of the dynamic contract or the `ensures` or `invariant` clause of the static contract fails.

Evaluation of a call of function f proceeds as follows. Let C and C' be the static and dynamic contracts of f, respectively. If the `requires` clause of C fails, a $\mathrm{CallerViolation}$ exception is thrown. Otherwise, if the `requires` clause of C' fails, a $\mathrm{ContractOverloadingViolation}$ exception is thrown. Otherwise, the `provided` subclauses of C and C' are evaluated. For every `provided` subclause that evaluates to $true$, the corresponding `ensures` subclause is recorded in a table E for later comparison. Similarly, the `invariant` clauses of C and C' are evaluated and the results are stored in E for later comparison. Then the body of the dynamically most applicable function declaration of f is evaluated. After evaluation of the body, all `ensures` subclauses of the dynamic contract recorded in E are checked to ensure that they evaluate to $true$, and all `invariant` clauses of the dynamic contract recorded in E are checked to ensure that they evaluate to values equal to the values they evaluated to before evaluation of the body. If any such check fails, a $\mathrm{CalleeViolation}$ exception is thrown. Otherwise, all `ensures` subclauses and `invariant` clauses of the static contract in E are checked. If any of these checks fails, a $\mathrm{ContractOverloadingViolation}$ exception is thrown.

Chapter 13

Expressions

Fortress is an expression-oriented language. Syntactically, the positions in which an expression may legally appear (*value context*) is determined by the nonterminal *Expr* in the Fortress grammar, defined in Appendix G.

13.1 Literals

Syntax:

Value ::= *Literal*

Fortress provides boolean literals, $()$ literal, character literals, string literals, and numeric literals. Literals are values; they do not require evaluation.

The literal *false* has type $\mathrm{BooleanLiteral}[\![\mathit{false}]\!]$. The literal *true* has type $\mathrm{BooleanLiteral}[\![\mathit{true}]\!]$.

The literal $()$ is the only value with type $()$. Whether any given occurrence of $()$ refers to the value $()$ or to the type $()$ is determined by context.

A character literal has type $\mathrm{Character}$. Each character literal consists of an abstract character in Unicode 5.0 [25], enclosed in single quotation marks (for example, '`a`', '`A`', '`$`', 'α', '$\oplus$'). For convenience, the single quotes may be either true typographical "curly" single quotation marks or a pair of ordinary apostrophe characters (for example, `'a'`, `'A'`, `'$'`, 'α', '$\oplus$'). See Section 5.9 for a description of how names of characters may be used rather than actual characters within character literals, for example '`APOSTROPHE`' and '`GREEK_CAPITAL_LETTER_GAMMA`'.

A string literal has type String. Each string literal is a sequence of Unicode 5.0 characters enclosed in double quotation marks (for example, "`Hello, world!`" or "$\pi\, r^2$"). For convenience, the double quotes may be either true typographical "curly" double quotation marks or a pair of "neutral" double-quote characters (for example, `"Hello, world!"` or "$\pi\, r^2$"). Section 5.10 also describes how names of characters may be used rather than actual characters within string literals. One may also use the escape sequences `\b` and `\t` and `\n` and `\f` and `\r` as described in [5].

Numeric literals in Fortress are referred to as *numerals*, corresponding to various expressible numbers. Numerals may be either *simple* or *compound* (as described in Section 5.13).

A numeral containing only digits (let n be the number of digits) has type $\mathrm{NaturalNumeral}[\![n, 10, v]\!]$ where v is the value of the numeral interpreted in radix ten. If the numeral has no leading zeros, or is the literal 0, then it also has type $\mathrm{Literal}[\![v]\!]$.

A numeral containing only digits (let n be the number of digits) then an underscore, then a radix indicator (let r be the radix) has type $\mathrm{NaturalNumeralWithExplicitRadix}[\![n, r, v]\!]$ where v is the value of the n-digit numeral interpreted in radix r.

A numeral containing only digits (let n be the total number of digits) and a radix point (let m be the number of digits after the radix point) has type $\mathrm{RadixPointNumeral}[\![n, m, 10, v]\!]$ where v is the value of the numeral, with the radix point deleted, interpreted in radix ten.

A numeral containing only digits (let n be the total number of digits) and a radix point (let m be the number of digits after the radix point), then an underscore, then a radix indicator (let r be the radix) has the following type:

$\mathrm{RadixPointNumeralWithExplicitRadix}[\![n, m, r, v]\!]$

where v is the value of the n-digit numeral, with the radix point deleted, interpreted in radix r.

Every numeral also has type $\mathrm{Numeral}[\![n, m, r, v]\!]$ for appropriate values of n, m, r, and v.

Numerals are not directly converted to any of the number types because, as in common mathematical usage, we expect them to be polymorphic. For example, consider the numeral $3.14159265358979323 84$; converting it immediately to a floating-point number may lose precision. If that numeral is used in an expression involving floating-point intervals, it would be better to convert it directly to an interval. Therefore, numerals have their own types as described above. This approach allows library designers to decide how numerals should interact with other types of objects by defining coercion operations (see Section 17.1 for an explanation of coercion in Fortress). The Fortress standard libraries define coercions from numerals to integers (for simple numerals) and rational numbers (for compound numerals).

In Fortress, dividing two integers using the / operator produces a *rational number*; this is true regardless of whether the integers are of type $\mathbb{Z}$ (or `ZZ`), $\mathbb{Z}64$ (or `ZZ64`), $\mathbb{N}32$ (or `NN32`), or whatever. Addition, subtraction, multiplication, and division of rationals are always exact; thus values such as $1/3$ are represented exactly in Fortress.

Numerals containing a radix point are actually rational literals; thus 3.125 has the rational value $3125/1000$. The quotient of two integer literals is a constant expression (described in Section 13.27) whose value is rational. Similarly, a sequence of digits with a radix point followed by the symbol $\times$ and an integer literal raised to an integer literal, such as 6.0221415×10^{23} is a constant expression whose value is rational. If such constants are mentioned as part of a floating-point computation, the compiler performs the rational arithmetic exactly and then converts the result to a floating-point value, thus incurring at most one floating-point rounding error. But in general rational computations may also be performed at run time, not just at compile time.

A rational number can be thought of as a pair of integers p and q that have been reduced to "standard form in lowest terms"; that is, $q > 0$ and there is no nonzero integer k such that $\frac{p}{k}$ and $\frac{q}{k}$ are integers and $\left|\frac{p}{k}\right| + \left|\frac{q}{k}\right| < |p| + |q|$. The type $\mathbb{Q}$ includes all such rational numbers.

The type $\mathbb{Q}^*$ relaxes the requirement $q > 0$ to $q \geq 0$ and includes two extra values, $1/0$ and $-1/0$ (sometimes called "the infinite rational" and "the indefinite rational"). The advantage of $\mathbb{Q}^*$ is that it is closed under the rational operations $+$, $-$, $\times$, and $/$. If a value of type $\mathbb{Q}^*$ is assigned to a variable of type $\mathbb{Q}$, a $\mathrm{DivideByZeroException}$ is thrown at run time if the value is $1/0$ or $-1/0$. The type $\mathbb{Q}^\#$ includes all of $1/0$, $-1/0$, and $0/0$. In ASCII, $\mathbb{Q}$, $\mathbb{Q}^*$, and $\mathbb{Q}^\#$ are written as `QQ`, `QQ_star`, and `QQ_splat`, respectively. See Section 38.1 for definitions of $\mathbb{Q}$, $\mathbb{Q}^*$, and $\mathbb{Q}^\#$.

13.1.1 Pi

The object named π (or `pi`) may be used to represent the ratio of the circumference of a circle to its diameter rather than a specific floating-point value or interval value. In Fortress, π has type $\mathrm{RationalValueTimesPi}[\![\mathit{false}, 1, 1]\!]$. When used in a floating-point computation, it becomes a floating-point value of the appropriate precision; when used in an interval computation, it becomes an interval of the appropriate precision.

13.1.2 Infinity and Zero

The object named ∞ has type $\mathrm{ExtendedIntegerValue}[\![\mathit{true}, 0, \mathit{true}]\!]$. One can negate ∞ to get a negative infinity.

Negating the literal 0 produces a special negative-zero object, which refuses to participate in compile-time constant arithmetic (discussed in Section 13.27). It has type $\mathrm{NegativeZero}$. The main thing it is good for is coercion to a floating-point number (discussed in Chapter 17). (Negating any other zero-valued expression simply produces zero.)

13.2 Identifier References

Syntax:

Expr	::=	*DottedName*[⟦*StaticArgList* ⟧]
	|	`self`
DottedName	::=	*DottedId*
	|	`opr` *Op*

A name that is not an operator appearing in an expression context is called an *identifier reference*. It evaluates to the value of the name in the enclosing scope in the value namespace. The type of an identifier reference is the declared type of the name. See Chapter 7 for a discussion of names. An identifier reference performs a memory read operation. Note in particular that if a name is not in scope, it is a static error (as described in Section 7.2).

An identifier reference which denotes a polymorphic function may include explicit type arguments (described in Chapter 12) but most identifier references do not include them; the type arguments are statically inferred from the context of the method invocation (as described in Chapter 20). For example, $identity⟦\mathrm{String}⟧$ is an identifier reference with an explicit type argument where the function $identity$ is defined as follows:

$$identity⟦T⟧(x:T):T = x$$

The special name `self` is declared as a parameter of a method. When the method is invocated, its receiver is bound to the `self` parameter; the value of `self` is the receiver. The type of `self` is the type of the trait or object being declared by the innermost enclosing trait or object declaration or object expression. See Section 9.2 for details about `self` parameters.

13.3 Dotted Field Accesses

Syntax:

Expr	::=	*Expr* . *Id*

An expression consisting of a single subexpression (called the *receiver expression*), followed by '`.`', followed by a name, not immediately followed by a parenthesis, is a *field access*. If the receiver expression denotes an object (called the *receiver*), the field access is evaluated to a call to a getter mapped from that name by the receiver. The type of the field access is the return type of its getter. The static type of the receiver indicates whether a getter mapped from that name is provided by the denoted object. If a getter is not provided, it is a static error. See Section 9.2 for a discussion of getters.

13.4 Dotted Method Invocations

Syntax:

Expr	::=	*Expr* . *Id*[⟦*StaticArgList* ⟧]`(`[*ExprList*]`)`
	|	*TraitType* . `coercion`[⟦*StaticArgList* ⟧]`(`*Expr*`)`

A *dotted method invocation* consists of a subexpression (called the receiver expression), followed by '.', followed by an identifier, an optional list of type arguments (described in Chapter 12) and a subexpression (called the *argument expression*). Unlike in function calls (described in Section 13.6), the argument expression must be parenthesized, even if it is not a tuple. There must be no whitespace on either side of the '.', and there must be no whitespace on the left-hand side of the left parenthesis of the argument expression. The receiver expression evaluates to the receiver of the invocation (bound to the self parameter (discussed in Section 9.2) of the method). A *coercion* invocation (discussed in Chapter 17) has a similar syntax to a dotted method invocation.

The subexpressions of a method invocation are evaluated *in parallel*; evaluation steps of the subexpressions can be interleaved, and even reordered, to form an evaluation of the method invocation. See Section 4.4 for a discussion of the semantics of their concurrent evaluation. A method invocation may include explicit instantiations of type parameters but most method invocations do not include them; the type arguments are statically inferred from the context of the method invocation (as described in Chapter 20). After the subexpressions of a dotted method invocation are evaluated to values, the body of the method is evaluated with the parameter of the method bound to the value of the argument expression. The value and the type of a dotted method invocation are the value and the type of the method body.

We say that methods or functions (collectively called as *functionals*) may be *applied to* (also "*invoked on*" or "*called with*") an argument. We use "call", "invocation", and "application" interchangeably.

Here are some examples:

$myString.toUppercase()$
$myString.replace(\text{“}\texttt{foo}\text{”}, \text{“}\texttt{few}\text{”})$
$\text{SolarSystem}.variation((\pi/2\,\text{radian})/452\ million\ year)$
$myNum.add(otherNum)$ $\quad(*$ NOT myNum.add otherNum $*)$

13.5 Naked Method Invocations

Syntax:

Expr ::= *Id Expr*

Method invocations that are not prefixed by receivers are *naked method invocations*. A naked method invocation is either a functional method call (See Section 9.2 for a discussion of functional methods) or a method invocation within a trait or object that provides the method declaration. Syntactically, a naked method invocation is same as a function call except that the method name is used instead of an arbitrary expression denoting the applied method. Like function calls, an argument expression need not be parenthesized unless it is a tuple. After the argument expression is evaluated to a value, the body of the method is evaluated with the parameter of the method bound to the value of the argument expression. The value and the type of a naked method invocation are the value and the type of the method body.

13.6 Function Calls

Syntax:

Expr ::= *Expr Expr*

A *function call* consists of two subexpressions: an expression denoting the applied function and an argument expression. The argument expression and the expression denoting the applied function are evaluated *in parallel*: evaluation steps of the subexpressions can be interleaved, and even reordered when forming an evaluation of the function call. See Section 4.4 for a description of the semantics of parallel evaluation. As with languages such as Scheme and the Java Programming Language, function calls in Fortress are call-by-value. An argument expression is evaluated to a value before the function is applied. After the subexpressions of a function call are evaluated to values, the body of

the function is evaluated with the parameter of the function bound to the value of the argument expression. The value and the type of a function call are the value and the type of the function body.

Here are some examples:

$sqrt(x)$
$\arctan(y, x)$

If the function's argument is not a tuple, then the argument need not be parenthesized:

$sqrt\, 2$
$\sin x$
$\log \log n$

13.7 Function Expressions

Syntax:

Value ::= `fn` *ValParam* [*IsType*] [*Throws*] $\Rightarrow$ *Expr*

Function expressions denote function values; they do not require evaluation. Syntactically, they start with the special reserved word `fn` followed by a parameter, optional return type, optional `throws` clause, $\Rightarrow$, and finally an expression. The type of a function expression is an arrow type consisting of the function's parameter type followed by the token $\rightarrow$, followed by the function's return type, and the function's optional `throws` clause. Unlike declared functions (described in Chapter 12), function expressions are not allowed to include static parameters nor `where` clauses (described in Chapter 11).

Here is a simple example:

`fn` $(x : \mathrm{Double}) \Rightarrow$ `if` $x < 0$ `then` $-x$ `else` x `end`

13.8 Operator Applications

Syntax:

Expr ::= *Op Expr*
| *Expr Op* [*Expr*]
Value ::= *LeftEncloser ExprList RightEncloser*

To support a rich mathematical notation, Fortress allows most Unicode characters that are specified to be mathematical operators to be used as operators in Fortress expressions, as well as various tokens described in Chapter 16. Most of the operators can be used as prefix, infix, postfix, or nofix operators as described in Section 16.3; the fixity of an operator is determined syntactically, and the same operator may have definitions for multiple fixities.

Syntactically, an operator application consists of an operator and its argument expressions. If the operator is a prefix operator, it is followed by its argument expression. If the operator is an infix operator, its two argument expressions come both sides of the operator. If the operator is a postfix operator, it comes right after its argument expression. Like function calls, argument expressions are evaluated *in parallel*. After evaluating argument expressions to values, the body of the operator definition is evaluated with the parameters of the operator bound to the values of the argument expressions. The value and the type of an operator application are the value and the type of the operator body.

Here are some examples:

$(-b + sqrt(b^2 - 4ac))/2a$
$n^n e^{(-n)} sqrt(2\pi n)$
$a_k b_{n-k}$
$x_1 y_2 - x_2 y_1$
$1/2gt^2$
$n(n+1)/2$
$(j+k)!/(j!k!)$
$1/3\ \ 3/5\ \ 5/7\ \ 7/9\ \ 9/11$
$17.3\,\mathrm{meter/second}$
$17.3\,\mathrm{m/s}$
$u \cdot (v \times w)$
$(A \cup B)$ `INTERSECT` C
$(A \cup B) \cap C$
$i < j \leq k \wedge p \prec q$
$print($“`The answers are `” $(p+q)$ “ `and `” $(p-q))$
$\langle 2, 3, 4, 5 \rangle$

13.9 Object Expressions

Syntax:

Val ::= `object` [*Extends*] (*FldDef* | *MdDef*)* `end`

Object expressions denote object values; they do not require evaluation. Syntactically, they start with the special reserved word `object`, followed by an optional `extends` clause, field declarations, method declarations, and finally the special reserved word `end`. The type of an object expression is an anonymous object trait type that extends the traits listed in the `extends` clause of the object expression. The object trait type does not include the methods introduced by the object expression (i.e., those methods not provided by any supertraits of the object expression). Every evaluation of an object expression has the same anonymous object trait type. Each object trait type is associated with a program location; any two object expressions with the same `extends` clause have different object trait types.

Unlike object declarations (described in Chapter 10), object expressions are not allowed to include modifiers nor value parameters nor static parameters nor `where` clauses (described in Chapter 11). While object declarations must not include any free static variable (i.e., all static variables in an object declaration must occur either as a static parameter or as a `where`-clause variable), object expressions may include free static variables.

For example, the following object expression:

$f[\![T]\!](x:T) =$ `object` $f:T = x$ `end`

has a static variable T that is not its static parameter nor its `where`-clause variable.

The following example expression evaluates to a new object extending trait List:

```
object extends { List }
  first() = throw Error
  rest() = throw Error
  cons(x) = Cons(x, self)
  append(xs) = xs
end
```

13.10 Assignments

Syntax:

Expr ::= *Expr AssignOp Expr*
AssignOp ::= := | *Op*=

An assignment expression consists of a left-hand side indicating one or more variables or a subscripted expression (as described in Section 34.7) to be updated, an assignment token, and a right-hand-side expression.

The assignment token may be ‘:=’, to indicate ordinary assignment; or may be any operator (other than ‘:’ or ‘=’ or ‘<’ or ‘>’) followed by ‘=’ with no intervening whitespace, to indicate compound (updating) assignment. A compound assignment is a syntactic sugar; for example, $x \mathrel{+}= e$ is a shorthand for $x := x + e$. An assignment expression evaluates its right-hand-side expression and binds its left-hand side to the value of the right-hand-side. An assignment expression performs a memory write operation.

A left-hand side of an assignment expression may be a single variable, a subscripted expression, or n variables using tuple notation. If tuple notation is used, then the right-hand side must be an expression which ultimately evaluates to a tuple of length n or a function application that returns a tuple of n values. Variables updated in assignment expressions must be already declared. The value of an assignment expression is $()$.

Here are some examples:

$x := f(0)$
$c_{ij} := c_{ij} + a_{ik}b_{kj}$
$(a, b, c) := (b, c, a)$ (∗ Permute a, b, and c ∗)
$x \mathrel{+}= 1$
$(x, y) \mathrel{+}= (\delta_{\mathrm{x}}, \delta_{\mathrm{y}})$
$myBag = myBag \cup newItems$
$myBag \cup = newItems$

13.10.1 Definite Assignment

References to uninitialized variables are statically forbidden. As with the Java Programming Language, this static constraint is ensured with a specific conservative flow analysis. In essence, an initialization of a variable must occur on every possible execution path to each reference to a variable. Variable initialization is performed in a local scope analogously to the rules for top-level initialization of simple components, defined in Section 22.6.

13.11 Do Expressions

Syntax:

Flow ::= *Do*
Do ::= `do` *BlockElem** `end`
BlockElem ::= *Expr*[, *GeneratorList*]
| *LocalVarFnDecl*

A `do` expression consists of the special reserved word `do`, a series of expressions, a generated expressions (described in Section 13.11.1) local variable declarations, or local function declarations (*block expression*), and the special reserved word `end`. The last of the block expression must not be a local declaration. A `do` expression evaluates its subexpressions and local declarations in order. The value and type of a `do` expression is the value and type of the last expression in the block expression. Each `do` expression introduces a new scope. Some compound expressions have clauses that are implicitly block expressions.

Here are examples of function declarations whose bodies are `do` expressions:

```
f(x : ℝ64) = do
  (sin(x) + 1)^2
end

foo(x : ℝ64) = do
  y = x
  z = 2x
  y + z
end

mySum(i : ℤ64) : ℤ64 = do
  acc : ℤ64 := 0
  for j ← 0 : i do
    acc := acc + j
  end
  acc
end
```

13.11.1 Generated Expressions

If a subexpression of a `do` expression has type $()$, the expression may be followed by a '`,`' and a generator list (described in Section 13.17). When a generator list is provided, generators produce values and bind the values to the identifiers that are used in the preceding expression. Most generators, unlike the *sequential* generator, may execute each evaluation of the assignment in a separate implicit thread.

13.11.2 Distinguishing a Local Declaration from an Equality Expression

Because a local declaration shares a syntax with an *equality expression*, we provide rules for disambiguation:

- If an expression of the form "$e = e$" occurs as a proper subexpression in any non-block expression, it is an equality expression.
- If such an expression occurs as an immediate subexpression of a block expression, it is a local declaration. Adding parentheses makes the expression an equality expression.

13.12 Parallel Do Expressions

Syntax:

Do ::= `do` *BlockElem*$^+$ `also` *Do*
| `at` *Expr Do*

A series of blocks may be run in parallel using the `also do` construct. Any number of contiguous blocks may be joined together by the special reserved word `also`. Each block is run in a separate implicit thread; these threads together form a group. A thread can be placed in a particular region by using an `at` expression as described in Section 32.7.

For example:

```
treeSum(t : TreeLeaf) = 0
treeSum(t : TreeNode) = do
   var accum := 0
   do
      accum += treeSum(t.left)
   also do
      accum += treeSum(t.right)
   also do
      accum += t.datum
   end
   accum
end
```

13.13 Label and Exit

Syntax:

Flow ::= `label` *Id Expr*$^+$ `end` *Id*
| `exit` [*Id*] [`with` *Expr*]

Block expressions may be labeled with an identifier. Syntactically, a `label` expression begins with the special reserved word `label` followed by an identifier, inner expressions (*label block*), the special reserved word `end`, and finally the same identifier. A `label` expression evaluates its inner expressions in order and any inner `exit` expression can exit the label block. Syntactically, an `exit` expression begins with the special reserved word `exit` followed by an optional identifier of the *targeted label block* with an optional value (*exit value*), which consists of the special reserved word `with` followed by an expression. If an `exit` expression does not have a `with` clause, it has an implicit exit value `()`. If an `exit` expression does not exist within a `label` expression, the value of the `label` expression is the value of the last expression of the label block. If an `exit` expression exists, the expression completes abruptly and attempts to transfer control to the end of the targeted label block. The targeted label block evaluates to the exit value of the `exit` expression. The type of a `label` expression is a union of the type of the last expression of its label block and the types of any `exit` values. The type of an `exit` expression is $\mathrm{BottomType}$.

If one or more `try` expressions are nested between an `exit` expression and the targeted label block, the `finally` clauses of these expressions are run in order, from innermost to outermost. Only when every intervening `finally` clause has completed normally does the targeted block complete normally. If any `finally` clause completes abruptly by throwing an exception, the `exit` expression fails to exit, the `label` expression completes abruptly, and the exception is propagated.

Here is a simple example:

```
label I95
   if goingTo(Sun)
   then exit I95 with x32B
   else  x32A
   end
end I95
```

The expression `exit` I_{95} `with` $x32B$ completes abruptly and attempts to transfer control to the end of the targeted label block `label` I_{95}. The targeted label block completes normally with value $x32B$.

13.14 While Loops

Syntax:

Flow ::= `while` *Expr Do*

A `while` loop consists of a *condition expression* of type $\mathrm{Boolean}$ followed by the special reserved word `do`, a series of expressions, local variable declarations, or local function declarations, and the special reserved word `end`. It evaluates the condition expression and the body expressions repeatedly until the value of the condition expression is false. The value of a `while` loop is $()$. The body expressions form a block expression and has the various properties of block expressions (described in Section 13.11).

13.15 For Loops

Syntax:

Flow ::= `for` *GeneratorList Do*

A `for` loop consists of the special reserved word `for` followed by a generator list (discussed in Section 13.17), followed by the special reserved word `do`, a series of expressions, local variable declarations, or local function declarations, and the special reserved word `end`. `for` loops are implicitly parallel. Parallelism in `for` loops is specified by the generators used (see Section 13.17). Most generators, unlike the $sequential$ generator, may execute each iteration of the loop body expression in a separate implicit thread. For each iteration, generators produce values and bind the values to the identifiers that are used in the loop body. The value of a `for` loop is $()$. The body expressions form a block expression and have the various properties of block expressions (described in Section 13.11).

13.15.1 Reduction Variables

To perform computations as locally as possible, and avoid the need to synchronize relatively simple `for` loops, Fortress gives special treatment to *reductions*. We say that an operator $\odot$ is a *reduction operator* for type T with an identity id if T is a subtype of $\mathrm{Monoid}[\![T, \odot, id]\!]$, which implies that $\odot$ is an associative binary infix operator of T (see Section 37.4 for details about the Monoid trait). A loop body may contain as many of the following assignment expressions using reduction operators as desired:

$l := l \odot expr$
$l := expr \odot l$
$l \odot= expr$

Reductions restrict a set of valid executions of `for` loops to get additional benefits such as less synchronizations with other threads. We say that a variable l is a *reduction variable* reduced using the reduction operator $\odot$ for a particular `for` loop if it satisfies the following conditions:

- Every assignment to l within the loop body uses the same reduction operator $\odot$, and the value of l is not otherwise read or written.
- The variable l is not a free variable of a `fn` expression or a method in an `object` expression which occurs in the loop body.
- The variable l is not an object field.

Other threads which simultaneously reference a reduction variable while a loop is running will see the value of the variable before the loop begins. At the end of the loop body, the original variable value before the loop and the final variable values from each execution of the loop body are combined together using the reduction operator, in some arbitrarily-determined order.

Several common mathematical operators are defined to be reduction operators in the Fortress standard libraries. These include $+$, $\cdot$, $\wedge$, $\vee$, and $\underline{\vee}$. If a type T extends $\mathrm{Group}[\![T, +, -, id]\!]$ (see Section 37.4 for details about the Group trait) then reduction expressions of the form:

$x \mathrel{-}= y$

are transformed into:

$x \mathrel{+}= x.id - y$

Note that since there are no guarantees on the order of execution of loop iterations, there are also no guarantees on the order of reduction.

The semantics of reductions enables implementation strategies such as OpenMP [22]: A reduction variable l is assigned $l.id$ at the beginning of each iteration. The original variable value may be read ahead of time, resulting in the loss of parallel updates to the variable which occur in other threads while the loop is running. Note that because this implementation strategy does not read reduction variables in the loop body, the actual implementation of reduction may vary substantially from the execution.

In the following example, sum is a reduction variable:

```
arraySum[\nat x\](a : ℝ64[x]) : ℝ64 = do
    sum : ℝ64 := 0
    for i ← a.indices do
        sum := sum + a_i
    end
    sum
end
```

13.16 Ranges

Syntax:

Range	::=	[*Expr*] : [*Expr*][: [*Expr*]]
	|	*Expr* # *Expr*

A *range expression* is used to create a special kind of $\mathrm{Generator}$ for a set of integers, called a Range, useful for indexing an array or controlling a `for` loop. Generators in general are discussed further in Section 13.17.

An *explicit range* is self-contained and completely describes a set of integers. Assume that a, b, and c are expressions that produce integer values.

- The range $a : b$ is the set of $n = \max(0, b - a + 1)$ integers $\{a, a + 1, a + 2, \ldots, b - 2, b - 1, b\}$. This is a *nonstrided* range. If a and b are both static expressions (described in Section 13.27), then it is a *static range* of type $\mathrm{StaticRange}[\![a, n, 1]\!]$ and therefore also a *range of static size*) of type $\mathrm{RangeOfStaticSize}[\![n]\!]$.
- The range $a : b : c$ is the set of $n = \max\left(0, \left\lfloor \frac{b-a+c}{c} \right\rfloor\right)$ integers $\{a, a + c, a + 2c, \ldots, a + \left\lfloor \frac{b-a}{c} \right\rfloor c\}$, unless c is zero, in which case it throws an exception. (If c is a static expression, then it is a static error if c is zero.) This is a *strided* range. If a, b, and c are all static expressions, then it is a *static range* of type $\mathrm{StaticRange}[\![a, n, c]\!]$ and therefore also a *range of static size*) of type $\mathrm{RangeOfStaticSize}[\![n]\!]$.
- The range $a \# n$ is the set of n integers $\{a, a+1, a+2, \ldots, a+n-3, a+n-2, a+n-1\}$, unless n is negative, in which case it throws an exception. (If n is a static expression, then it is a static error if n is negative.) This is a *nonstrided* range. If a and n are both static expressions, then it is a *static range* of type $\mathrm{StaticRange}[\![a, n, 1]\!]$ If b is a static expression, then it is a *range of static size* of type $\mathrm{RangeOfStaticSize}[\![n]\!]$, even if a is not a static expression.

An *implicit range* may be used only in certain contexts, such as array subscripts, that can supply implicit information. Suppose an implicit range is used as a subscript for an axis of array for which the lower bound is l and the upper bound is u.

- The implicit range $:$ is treated as $l:u$.
- The implicit range $::c$ is treated as $l:u:c$.
- The implicit range $:b$ is treated as $l:b$.
- The implicit range $:b:c$ is treated as $l:b:c$.
- The implicit range $a:$ is treated as $a:u$.
- The implicit range $a::c$ is treated as $a:u:c$.

One may test whether an integer is in a range by using the operator $\in$:

`if` $j \in a:b$ `then` *print* "`win`" `end`

Ranges may be compared as if they were sets of integers by using $\subset$ (`SUBSET`) and $\subseteq$ (`SUBSETEQ`) and $=$ and $\supseteq$ (`SUPSETEQ`) and $\supset$ (`SUPSET`).

Nonstrided ranges may be intersected using the operator $\cap$ (`INTERSECTION`).

The size of a range (the number of integers in the set) may be found by using the set-cardinality operator $|\ldots|$. For example, the value of $|3:7|$ is 5 and the value of $|1:100:2|$ is 50.

Note that a range is very different from an interval with integer endpoints. The range $3:5$ contains only the values 3, 4, and 5, whereas the interval $[3,5]$ contains all real numbers x such that $3 \le x \le 5$.

13.17 Generators

Syntax:

GeneratorList	::=	*Generator* (, *Generator*)*
Generator	::=	*Id* $\leftarrow$ *Expr*
	|	(*Id* , *IdList*) $\leftarrow$ *Expr*
	|	*Expr*
IdList	::=	*Id* (, *Id*)*

Fortress makes extensive use of comma-separated *generator lists* to express parallel iteration. Generator lists occur in generated expressions (described in Section 13.11.1), `for` loops (described in Section 13.15), sums and big operators (described in Section 13.18), and comprehensions (described in Section 13.29). We refer to these collectively as *expressions with generators*. Every expression with generators contains a *body expression* (for an assignment this expression is the assignment itself) which is evaluated for each combination of values bound in the generator list.

An element of a generator list is either a *generator binding* or a boolean expression. A generator binding consists of one or more comma-separated identifiers followed by the token $\leftarrow$, followed by a subexpression (called the *generator expression*). A generator expression evaluates to an object whose type is $\mathrm{Generator}$. For each iteration, a generator object produces a value or a tuple of values. These values are bound to the identifiers to the left of the arrow, which are in scope of subsequent generator list elements and of the body of the construct containing the generator list.

A boolean expression in a generator list is interpreted as a *filter*. An iteration is performed only if the result of the filter expression is true. If the filter is false, subsequent expressions in the generator list will not be evaluated. However, other than this restriction, there is no implied order of evaluation of the generator expressions or the boolean expressions in a generator list. Thus, for example, if a boolean expression in the middle of the list evaluates to $true$, then generator expressions to its right in the list may be evaluated before generator expressions to its left.

The body of each iteration is run in its own implicit thread. The expressions in the generator list can each be considered to run in a separate implicit thread. Together these implicit threads form a thread group.

Some common $\mathrm{Generator}$s include:

$l : u$	Any range expression
$a.indices$	The index set of an array a
$\{0, 1, 2, 3\}$	The elements of an aggregate expression
$sequential(g)$	A sequential version of generator g

The generator $sequential(g)$ forces the iterations using distinct values from g to be performed in order. Every generator has an associated *natural order* which is the order obtained by $sequential$. For example, a sequential `for` loop starting at 1 and going to n can be written as follows:

```
for i ← sequential(1 : n) do
  ...
end
```

Given a multidimensional array, the $indices$ generator returns a tuple of values, which can be bound by a tuple of variables to the left of the arrow:

$(i, j) \leftarrow my2DArray.indices$

The parallelism of a loop on this generator follows the spatial distribution (discussed in Section 32.5) of $my2DArray$ as closely as possible.

The order of nesting of generators need not imply anything about the relative order of nesting of iterations. In most cases, multiple generators can be considered equivalent to multiple nested loops. However, the compiler will make an effort to choose the best possible iteration order it can for a multiple-generator loop; there may be no such guarantee for nested loops. Thus loops with multiple generators are preferable in general. Note that the early termination behavior of nested looping is subtly different from a single multi-generator loop; see Section 32.6.

13.18 Summations and Other Reduction Expressions

Syntax:

Flow	::=	*Accumulator* `[[` *GeneratorList* `]]` *Expr*
Accumulator	::=	$\sum$ | $\prod$ | `BIG` *Op*

A *reduction expression* begins with a big operator such as $\sum$ or $\prod$ followed by an optional generator list (described in Section 13.17), followed by a subexpression. A complete list of these operators are described in Section 16.8.1. When a generator list is provided, generators produce values and bind the values to the identifiers that are used in the subexpression. Most generators, unlike the $sequential$ generator, may execute each evaluation of the subexpression in a separate implicit thread. The value of a reduction expression is the result of the operation over the values of the subexpressions. The type of a reduction expression is the return type of the big operator used.

A reduction expression with a generator list:

$\sum[v_1 \leftarrow g_1, v_2 \leftarrow g_2, \ldots]e$

is equivalent to the following code:

```
do
  result = 0
  for v1 ← g1, v2 ← g2, ... do
    result += e
  end
```

$result$
`end`

where $result$ is a fresh variable. A reduction expression without a generator list:

$\sum g$

is equivalent to the following:

$\sum[x \leftarrow g]x$

Note that reduction expressions without generator lists can be used to conveniently sum any aggregate expression (described in Section 13.28), since every aggregate expression is a generator.

13.19 If Expressions

Syntax:

Flow ::= `if` *Expr* `then` *Expr*$^+$ (`elif` *Expr* `then` *Expr*$^+$)* [*Else*] `end`
| (`if` *Expr* `then` *Expr*$^+$ (`elif` *Expr* `then` *Expr*$^+$)* *Else* [`end`])
Else ::= `else` *Expr*$^+$

An `if` expression consists of the special reserved word `if` followed by a condition expression of type $\mathrm{Boolean}$, followed by the special reserved word `then`, a sequence of expressions, an optional sequence of `elif` clauses (each consisting of the special reserved word `elif` followed by a condition expression, the special reserved word `then`, and a sequence of expressions), an optional `else` clause (consisting of the special reserved word `else` followed by a sequence of expressions), and finally the special reserved word `end`. Each clause forms a block expression and has the various properties of block expressions (described in Section 13.11). An `if` expression first evaluates its condition expression. If the condition expression evaluates to true, the `then` clause is evaluated. Otherwise, the next to the `then` clause is evaluated. An `elif` clause evaluates its condition expression first and proceeds similarly to an `if` expression. The type of an `if` expression is the union of the types of all right-hand sides of the clauses. If there is no `else` clause in an `if` expression, then the last expression in every clause must evaluate to $()$. The special reserved word `end` may be elided if the `if` expression is immediately enclosed by parentheses. In such a case, an `else` clause is required.

For example,

```
if x ∈ {0, 1, 2} then 0
elif x ∈ {3, 4, 5} then 3
else 6 end
```

13.20 Case Expressions

Syntax:

Flow ::= `case` *Expr* [*Op*] `of` (*Expr* ⇒ *Expr*$^+$)$^+$ [*Else*] `end`

A `case` expression begins with the special reserved word `case` followed by a condition expression, followed by an optional operator, the special reserved word `of`, a sequence of case clauses (each consisting of a *guarding expression* followed by the token ⇒, followed by a sequence of expressions), an optional `else` clause (consisting of the special reserved word `else` followed by a sequence of expressions), and finally the special reserved word `end`.

A `case` expression evaluates its condition expression and checks each case clause to determine which case clause matches. To find a matched case clause, the guarding expression of each case clause is evaluated in order and compared

to the value of the condition expression. The two values are compared according to an optional operator specified. If the operator is omitted, it defaults to $=$ or $\in$. If the condition expression has type Generator or if the guarding expression does not, then the default operator is $=$; otherwise, it is $\in$. It is a static error if the specified operator is not defined for these types or if the operator's return type is not $\mathrm{Boolean}$.

The right-hand side of the first matched clause (and only that clause) is evaluated. If no matched clause is found, a $\mathrm{MatchFailure}$ exception is thrown. The right-hand side of each clause forms a block expression and has the various properties of block expressions (described in Section 13.11). The optional `else` clause always matches. The value of a `case` expression is the value of the right-hand side of the matched clause. The type of a `case` expression is the union type of the types of all right-hand sides of the case clauses.

For example, the following `case` expression specifies the operator $\in$:

```
case planet ∈ of
  { Mercury, Venus, Earth, Mars } ⇒ "inner"
  { Jupiter, Saturn, Uranus, Neptune, Pluto } ⇒ "outer"
  else ⇒ "remote"
end
```

but the following does not:

```
case 2 + 2 of
  4 ⇒ "it really is 4"
  5 : 7 ⇒ "we were wrong again"
end
```

13.21 Extremum Expressions

Syntax:

Flow ::= `case (largest | smallest)` [*Op*] `of` (*Expr* ⇒ *Expr*$^+$)$^+$ `end`

An extremum expression uses the same syntax as a `case` expression (described in Section 13.20) except that the special reserved word `largest` or `smallest` is used where a `case` expression would have a condition expression and an extremum expression does not have an optional `else` clause.

All guarding expressions of an extremum expression is evaluated *in parallel*. See Section 4.4 for a discussion of parallel evaluation. To find the largest (or smallest) quantity, the values of the guarding expressions are compared in parallel according to an optional operator specified. If the operator is omitted, it defaults to `CMP`. The union of the types of all the candidate expressions must be a subtype of $\mathrm{TotalOrderOperators}[\![T, <, \leq, \geq, >, \texttt{CMP}]\!]$ for some T, $<$, $\leq$, $\geq$, and $>$, if a default operator is used (see Section 37.2 for details about the $\mathrm{TotalOrderOperators}$ trait). If an explicit operator is used, the explicit operator replaces `CMP` in $\mathrm{TotalOrderOperators}[\![T, <, \leq, \geq, >, \texttt{CMP}]\!]$.

The right-hand side of the clause with the largest (smallest) guarding expression (and only that clause) is evaluated. If more than one guarding expressions are tied for largest (smallest), the leftmost clause is evaluated. The right-hand side of each clause forms a block expression and has the various properties of block expressions (described in Section 13.11). The value of an extremum expression is the value of the right-hand side of the matched clause. The type of an extremum expression is the union type of the types of all right-hand sides of the clauses.

For example, the following code:

```
case largest of
  1 mile ⇒ "miles are larger"
  1 kilometer ⇒ "we were wrong again"
end
```

evaluates to "`miles are larger`". A more interesting example is described in Section 6.5.

13.22 Typecase Expressions

Syntax:

Flow	::=	`typecase` *TypecaseBindings* `in` (*TypecaseTypeRefs* $\Rightarrow$ *Expr*$^+$)$^+$ [*Else*] `end`
TypecaseBindings	::=	*Id*
	|	*Binding*
	|	(*BindingList*)
Binding	::=	*Id* $=$ *Expr*
BindingList	::=	*Binding* (, *Binding*)*
TypecaseTypeRefs	::=	*TypeRef*
	|	(*TypeRefList*)

A `typecase` expression begins with the special reserved word `typecase` followed by a sequence of bindings (either an identifier or a sequence of an identifier followed by the token $=$, followed by an expression), followed by the special reserved word `in`, a sequence of typecase clauses (each consisting of a sequence of *guarding types* followed by the token $\Rightarrow$, followed by a sequence of expressions), an optional `else` clause (consisting of the special reserved word `else` followed by a sequence of expressions), and finally the special reserved word `end`.

A `typecase` expression evaluates its bindings and checks each typecase clause to determine which typecase clause matches. A single identifier x (where x is a valid local identifier) as the binding of a `typecase` expression is a shorthand for $x = x$. Each subexpression in the bindings is evaluated and its value is bound to the corresponding identifier. To find a matched typecase clause, the guarding types of each typecase clause are compared to the types of the identifiers bound in the bindings in order. The right-hand side of the first matched clause (and only that clause) is evaluated. If no matched clause is found, a $\mathrm{MatchFailure}$ exception is thrown. The right-hand side of each clause forms a block expression and has the various properties of block expressions (described in Section 13.11). The optional `else` clause always matches. The value of a `typecase` expression is the value of the right-hand side of the matched clause. The type of a `typecase` expression is the union type of the types of all right-hand sides of the typecase clauses.

For example:

```
typecase x = myLoser.myField in
    String ⇒ x.append("foo")
    Number ⇒ x + 3
    Object ⇒ yogiBerraAutograph
end
```

Note that "x" has a different type in each clause.

13.23 Atomic Expressions

Syntax:

Flow	::=	`atomic` *Expr*
	|	`tryatomic` *Expr*

As Fortress is a parallel language, an executing Fortress program consists of a set of threads (See Section 4.4 for a discussion of parallelism in Fortress). In multithreaded programs, it is often convenient for a thread to evaluate some expressions *atomically*. For this purpose, Fortress provides `atomic` expressions.

An `atomic` expression consists of the special reserved word `atomic` followed by a *body expression*. Evaluating an `atomic` expression is simply evaluating the body expression. All reads and all writes which occur as part of this evaluation will appear to occur simultaneously in a single atomic step with respect to *any* action performed by any thread which is dynamically outside. This is specified in detail in Chapter 21. The value and type of an `atomic` expression are the value and type of its body expression.

A `tryatomic` expression consists of the special reserved word `tryatomic` followed by an expression. See Section 32.3 for a discussion of `tryatomic` expressions.

A function or method with the modifier `atomic` acts as if its entire body were surrounded in an `atomic` expression. However, it is a static error if an API declares a functional f with the modifier `atomic` but a component implementing the API defines f whose body is an `atomic` expression without the modifier. Input and output (including functionals with the modifier `io`) cannot be performed within an `atomic` expression. Thus, a functional must not have both `atomic` and `io` modifiers.

When the body of an `atomic` expression completes abruptly, the `atomic` expression completes abruptly in the same way. If it completes abruptly by exiting to an enclosing `label` expression, writes within the block are retained and become visible to other threads. If it completes abruptly by throwing an uncaught exception, all writes to objects allocated before the `atomic` expression began evaluation are discarded. Writes to newly allocated objects are retained. Any variable reverts to the value it held before evaluation of the `atomic` expression began. Thus, the only values retained from the abruptly completed `atomic` expression will be reachable from the exception object through a chain of newly allocated objects.

Atomic expressions may be nested arbitrarily; the above semantics imply that an inner `atomic` expression is atomic with respect to evaluations which occur dynamically outside the inner `atomic` expression but dynamically inside an enclosing `atomic`.

Implicit threads may be created dynamically within an `atomic` expression. These implicit threads will complete before the `atomic` expression itself does so. The implicit threads may run in parallel, and will see one another's writes; they may synchronize with one another using nested `atomic` expressions.

A spawned thread created in an `atomic` expression conceptually begins execution after the `atomic` expression, so long as an exception is not thrown by the `atomic` expression. Thus the body of the spawned thread is dynamically outside the `atomic` expression in which it was created. It is legal to either spawn a thread (discussed in Section 13.24) or to synchronize with it using the val or $wait$ methods during the course of an `atomic` expression, but it is illegal to do both to the same thread. Doing so will cause the method call to throw the $\mathrm{AtomicSpawnSynchronization}$ exception.

Note that `atomic` expressions may be evaluated in parallel with other expressions. An `atomic` expression experiences *conflict* when another thread attempts to read or write a memory location which is accessed by the `atomic` expression. The evaluation of such an expression must be partially serialized with the conflicting memory operation (which might be another `atomic` expression). The exact mechanism by which this occurs will vary; the necessary serialization is provided by the implementation. In general, the evaluation of a conflicting `atomic` expression may be abandoned, forcing the effects of execution to be discarded and execution to be retried. The longer an `atomic` expression evaluates and the more memory it touches the greater the chance of conflict and the larger the bottleneck a conflict may impose.

For example, the following code uses a shared counter atomically:

```
arraySum⟦N extends Additive, nat x⟧(a : N[x]) : N = do
  sum : N := 0
  for i ← a.indices do
    atomic sum += a_i
  end
  sum
end
```

The loop body reads a_i and sum, then adds them and writes the result back to sum; this will appear to occur atomically with respect to all other threads (including the other iterations of the loop body). Note in particular that the `atomic` expression will appear atomic with respect to reads and writes not in `atomic` expressions of a_i and sum.

13.24 Spawn Expressions

Syntax:

Flow ::= `spawn` *Expr*

A thread can be created by a `spawn` expression. A `spawn` expression consists of the special reserved word `spawn` followed by an expression. A `spawn` expression spawns a thread which evaluates its subexpression in parallel with any succeeding evaluation. The value of a `spawn` expression is the spawned thread and the type of the expression is the Thread trait.

The Thread trait has the following methods:

- The val method returns the value computed by the subexpression of the `spawn` expression. If the thread has not yet completed execution, the invocation of val blocks until it has done so.
- The $wait$ method waits for a thread to complete, but does not return a value.
- The $ready$ method returns $true$ if a thread has completed, and returns $false$ otherwise.
- The $stop$ method attempts to terminate a thread as described in Section 32.6.

In the absence of sufficient parallel resources, an attempt is made to run the subexpression of the `spawn` expression before continuing succeeding evaluation (so long as we have not specified a region for evaluation as described in Section 32.7, and we are not currently evaluating an `atomic` expression as described in Section 13.23). We can imagine that it is actually the *rest* of the evaluation *after* the parallel block which is spawned off in parallel. This is a subtle technical point, but makes the sequential execution of parallel code simpler to understand, and avoids subtle problems with the asymptotic stack usage of parallel code [19, 11].

13.25 Throw Expressions

Syntax:

Flow ::= `throw` *Expr*

A `throw` expression consists of the special reserved word `throw` followed by a subexpression. The subexpression must have the type $\mathrm{Exception}$ (see Chapter 14). A `throw` expression evaluates its subexpression to an exception value and throws the exception value; the expression completes abruptly and has $\mathrm{BottomType}$.

The type $\mathrm{Exception}$ has exactly two direct mutually exclusive subtypes, $\mathrm{CheckedException}$ and $\mathrm{UncheckedException}$. Every $\mathrm{CheckedException}$ that is thrown must be caught or forbidden by an enclosing `try` expression (see Section 13.26), or it must be declared in the `throws` clause of an enclosing functional declaration (see Section 12.1). Similarly, every $\mathrm{CheckedException}$ declared to be thrown in the static type of a functional called must be either caught or forbidden by an enclosing `try` expression, or declared in the `throws` clause of an enclosing functional declaration.

13.26 Try Expressions

Syntax:

Flow ::= `try` *Expr*$^+$ [`catch` *Id* (*TraitType* $\Rightarrow$ *Expr*$^+$)$^+$] [`forbid` *TraitTypes*] [`finally` *Expr*$^+$] `end`

A `try` expression starts with the special reserved word `try` followed by a sequence of expressions (the `try` *block*), followed by an optional `catch` clause, an optional `forbid` clause, an optional `finally` clause, and finally the special reserved word `end`. A `catch` clause consists of the special reserved word `catch` followed by an identifier, followed by a sequence of subclauses (each consisting of an exception type followed by the token $\Rightarrow$ followed by a sequence of expressions). A `forbid` clause consists of the special reserved word `forbid` followed by a set of exception types. A `finally` clause consists of the special reserved word `finally` followed by a sequence of expressions. Note that the `try` block and the clauses form block expressions and have the various properties of block expressions (described in Section 13.11).

The expressions in the `try` block are first evaluated in order until they have all completed normally, or until one of them completes abruptly. If the `try` block completes normally, the *provisional* value of the `try` expression is the value of the last expression in the `try` block. In this case, and in case of exiting to an enclosing `label` expression, the `catch` and `forbid` clauses are ignored.

If an expression in the `try` block completes abruptly by throwing an exception, the exception value is bound to the identifier specified in the `catch` clause, and the type of the exception is matched against the subclauses of the `catch` clause in turn, exactly as in a `typecase` expression (Section 13.22). The right-hand-side sequence of expressions of the first matching subclause is evaluated. If it completes normally, its value is the provisional value of the `try` expression. If the `catch` clause completes abruptly, the `try` expression completes abruptly. If a thrown exception is not matched by the `catch` clause (or this clause is omitted), but it is a subtype of the exception type listed in a `forbid` clause, a new $\mathrm{ForbiddenException}$ is created with the thrown exception as its argument and thrown. The exception thrown by the `try` block is *chained* to the $\mathrm{ForbiddenException}$ as described in Section 14.3.

If an exception thrown from a `try` block is matched by both `catch` and `forbid` clauses, the exception is caught by the `catch` clause. If an exception thrown from a `try` block is not matched by any `catch` or `forbid` clause, the `try` expression completes abruptly.

The `finally` clause is evaluated after completion of the `try` block and any `catch` or `forbid` clause. The expressions in the `finally` clause are evaluated in order until they have all completed normally, or until one of them completes abruptly. In the latter case, the `try` expression completes abruptly exactly as the subexpression in the `finally` clause does.

If the `finally` clause completes normally, and the `try` block or the `catch` clause completes normally, then the `try` expression completes normally with the provisional value of the `try` expression. Otherwise, the `try` expression completes abruptly as specified above.

For example, the following `try` expression:

```
try
   inp = read(file)
   write(inp, newFile)
forbid IOException
end
```

is equivalent to:

```
try
   inp = read(file)
   write(inp, newFile)
catch e
```

```
    IOException ⇒ throw ForbiddenException(e)
end
```

The following example ensures that *file* is closed properly even if an IO error occurs:

```
try
    open(file)
    inp = read(file)
    write(inp, newFile)
catch e
    IOException ⇒ throw ForbiddenException(e)
finally
    close(file)
end
```

13.27 Static Expressions

Static expressions denote *static values*. Given instantiations of all static parameters (described in Chapter 11) in scope of a static expression, the value of the static expression can be determined statically. Static expressions can be used as instantiations of static parameters. We define the set of static expressions by first defining the types of static expressions, and distinguishing static values from the closely related literal values. We then describe the expressions that evaluate to the various kinds of static values.

13.27.1 Types of Static Expressions

There are three groups of traits that describe literals and static expressions:

1. The *literal* traits, which describe boolean, character, string, and numerals. For example, the literal $true$ has trait $\mathrm{BooleanLiteral}[\![true]\!]$ and a character literal has trait $\mathrm{Character}$. See Section 13.1 for a discussion of Fortress literals.

2. The *constant* traits, which describe values denoted by expressions composed from literals and operators; the type of a constant expression encodes the value of the expression. For example, the type of a constant expression $3 + 5$, $\mathrm{IntegerConstant}[\![false, 3 + 5]\!]$, encodes the value of the expression.

3. The *static* traits, which describe values denoted by expressions composed from literals, operators, and `bool`, `nat`, and `int` parameters; here the type does not encode the value of the expression, but the value of the expression can nevertheless be known statically if specific values are specified for the static parameters. Also, in situations where the type of an expression composed solely from literals and operators nevertheless cannot be described by a constant trait, then a static trait may be used to describe it instead. For example, a static expression $2(3 + m)$ where m is a `nat` parameter has trait $\mathrm{NaturalStatic}$.

The only operation on literals that produces a new literal (as opposed to a constant) is concatenation by using the operator $\|$. One may concatenate mixed string and character literals, producing a string literal. For example, "`foo`" $\|$ "`bar`" yields "`foobar`". One may also concatenate two natural numerals of the same radix, producing a new natural numeral of that radix. For example, $\texttt{deadc}_{16} \| \texttt{0de}_{16}$ yields $\texttt{deadc0de}_{16}$.

Every literal trait extends an appropriate constant trait, and every constant trait extends an appropriate static trait. So every literal is also a constant expression, and every constant expression is a static expression.

13.27.2 Static Expressions and Values

Static parameters are static expressions. A `nat` parameter denotes a value that has type $\mathrm{NaturalStatic}$ (which extends $\mathrm{IntegerStatic}$). An `int` parameter denotes a value that has type $\mathrm{IntegerStatic}$. A `bool` parameter denotes a value that has type $\mathrm{BooleanStatic}$.

Boolean static expressions may be combined using the operators $\wedge$, $\vee$, $\oplus$, $\equiv$, $\leftrightarrow$, `NAND`, `NOR`, $=$, $\neq$, and $\rightarrow$ (See Appendix F for a discussion of Fortress operators.) to produce other static expressions denoting boolean static expressions. If both operands are boolean constant expressions, then the result is also a boolean constant expression.

Character static expressions may be compared using the operators $<$, $\leq$, $\geq$, $>$, $=$, and $\neq$ to produce boolean static expressions. If both operands are character constant expressions, then the result is a boolean constant expression.

Numeric static expressions may be compared using the operators $<$, $\leq$, $\geq$, $>$, $=$, and $\neq$ to produce boolean static expressions. If both operands are numeric constant expressions, then the result is a boolean constant expression.

Numeric static expressions may be combined using the operators and functions $+$, $-$, $\times$, $\cdot$, $/$, `!`, `MIN`, `MAX`, $\sqrt{}$, *floor*, *ceiling*, *hyperfloor*, *hyperceiling*, $\gcd$, lcm, $\sin$, $\cos$, $\tan$, $\arcsin$, $\arccos$, and $\arctan$ to produce new numeric static expressions. If the result is indeed a numeric static expression, and both operands are numeric constant expressions, then the result is also a numeric constant expression (except under certain circumstances—for example, $sqrt\ 5$ denotes a numeric constant expression, and so does $(1 + sqrt\ 5)/2$, but $sqrt((1 + sqrt\ 5)/2)$ does not, because it is too complicated).

Where things get too complicated, a static expression evaluation backs off, widest-need processing steps in at a later stage and chooses an appropriate precision of floating-point arithmetic.

13.28 Aggregate Expressions

Syntax:

Value ::= *Aggregate*

Aggregate expressions evaluate to values that are themselves homogeneous collections of values. Evaluation of the subexpressions of an aggregate expression is performed *in parallel*. Evaluation steps of the subexpressions can be interleaved and even reordered to form an evaluation of the aggregates expression. See Section 4.4 for a discussion of parallel evaluation.

Syntax for aggregate expressions are provided in the Fortress standard libraries for sets, maps, lists, tuples, matrices, vectors, and arrays.

Set Expressions:

Syntax:

Aggregate ::= { [*ExprList*] }
ExprList ::= *Expr* (, *Expr*)*

Set element expressions are enclosed in braces and separated by commas. The type of a set expression is $\mathrm{Set}[\![T]\!]$, where T is the union type of the types of all element expressions of the set expression.

Set containment is checked with the operator $\in$ and the subset relationship is checked with the operator $\subseteq$. For example:

$$3 \in \{0, 1, 2, 3, 4, 5\}$$

evaluates to $true$ and

$$\{0, 1, 2\} \subseteq \{0, 3, 2\}$$

evaluates to $false$.

Map Expressions:

Syntax:

Aggregate ::= { [*EntryList*] }
EntryList ::= *Entry* (, *Entry*)*
Entry ::= *Expr* ↦ *Expr*

Map entries are enclosed in curly braces, separated by commas, and matching pairs are separated by ↦. The type of a map expression is $\mathrm{Map}[\![S, T]\!]$ where S is the union type of the types of all left-hand-side expressions of the map entries, and T is the union type of the types of all right-hand-side expressions of the map entries. This type can be abbreviated as $\{S \mapsto T\}$.

A map m is indexed by placing an element in the domain of m enclosed in brackets immediately after an expression evaluating to m. Thus, the index is rendered as a subscript. For example, if:

$$m = \{\texttt{'a'} \mapsto 0, \texttt{'b'} \mapsto 1, \texttt{'c'} \mapsto 2\}$$

then $m_{\texttt{'b'}}$ evaluates to 1. In contrast, $m_{\texttt{'x'}}$ throws a $\mathrm{NotFound}$ exception, as `'x'` is not an index of m.

List Expressions:

Syntax:

Aggregate ::= ⟨ [*ExprList*] ⟩

List element expressions are enclosed in angle brackets ⟨ and ⟩ and are separated by commas. The type of a list expression is $\mathrm{List}[\![T]\!]$ where T is the union type of the types of all element expressions. This type can be abbreviated as $\langle T \rangle$.

A list l is indexed by placing an index enclosed in square brackets immediately after an expression evaluating to l. Thus, the index is rendered as a subscript. Lists are always indexed from 0. For example:

$$\langle 3, 2, 1, 0 \rangle_2$$

evaluates to 1.

Array Expressions

Syntax:

Aggregate ::= [(*Expr* | ;)*]

Array element expressions are enclosed in brackets. Element expressions along a row are separated only by whitespace. Two dimensional array expressions are written by separating rows with newlines or semicolons. If a semicolon appears, whitespace before and after the semicolon is ignored. The parts of higher-dimensional array expressions are separated by repeated-semicolons, where the dimensionality of the result is equal to one plus the number of repeated semicolons. The type of a k-dimensional array expression is $\mathrm{Array}[\![T]\!][n_0, \cdots, n_{k-1}]$, where T is the union type of the types of the element expressions and $n_0, ..., n_{k-1}$ are the sizes of the array in each dimension. This type can be abbreviated as $T[n_0, \cdots, n_{k-1}]$.

A k-dimensional array A is indexed by placing a sequence of k indices enclosed in brackets, and separated by commas, after an expression evaluating to A. Thus, the index is rendered as a subscript. By default arrays are indexed from 0. The horizontal dimension of an array is the last dimension mentioned in the array index. For example:

$A = [1\ 2\ 3; 4\ 5\ 6; 7\ 8\ 9]$

then $A_{1,0}$ evaluates to 4.

An array of two dimensions whose elements are a subtype of Number can be coerced to a matrix. Matrix types are written $\mathrm{Matrix}[\![T]\!][n_0 \times \ldots \times n_{k-1}]$, where $k > 1$. A type of this form can be abbreviated as $T^{n_0 \times \ldots \times n_{k-1}}$. Matrices are indexed in the same manner as arrays.

A one-dimensional array whose elements are a subtype of Number can be coerced to a vector. Vector types are written $\mathrm{Vector}[\![T]\!][n]$. A type of this form can be abbreviated as T^n, unless T is declared in the enclosing scope to be a physical dimension or unit.

The element expressions in an array expression may be either scalars or array expressions themselves. If an element is an array expression, it is "flattened" (pasted) into the enclosing expression. This pasting works because arrays never contain other arrays as elements. The elements along a row (or column) must have the same number of columns (or rows), though two elements in different rows (columns) need not have the same number of columns (rows). See Section 6.5 for a discussion of matrix unpasting.

The following four examples are all equivalent:

```
[3 4          [3 4 ;        [  3 4          [3 4 ; 5 6]
 5 6]          5 6 ]
                             ; 5 6 ]
```

Here is a $3 \times 3 \times 3 \times 2$ matrix example:

```
[1 0 0
 0 1 0
 0 0 1 ; ; 0 1 0
           1 0 1
           0 1 0 ; ; 1 0 1
                     0 1 0
                     1 0 1
 ; ; ;
 1 0 0
 0 1 0
 0 0 1 ; ; 0 1 0
           1 0 1
           0 1 0 ; ; 1 0 1
                     0 1 0
                     1 0 1]
```

Tuple Expressions:

Syntax:

Aggregate ::= (*Expr*(, *Expr*)$^+$)
| ([*Expr*(, *Expr*)* ,] *Expr* ...)
| ([*Expr*(, *Expr*)* ,] [*Expr* ... ,] *Id* = *Expr* (, *Id* = *Expr*)*)

Tuple element expressions are enclosed in parentheses and separated by commas. Each element is one of:

- A plain expression "e"
- A varargs expression "$e\ldots$"
- A keyword-value pair "$identifier = e$"

The following restrictions apply: No two keyword-value pairs may have the same keyword. No keyword-value pair may precede a plain expression. No varargs expression may follow a keyword-value pair or precede a plain expression. There must be at least one item. If there is exactly one item, it must be a varargs expression or a keyword-value pair (because an expression "(e)" is simply a parenthesized expression, not a tuple.) Also, there can be at most one item with a varargs expression. In a varargs expression, the expression "e" must be of type $\mathrm{OrderedGenerator}[\![T]\!]$ for some T.

The type of a tuple expression is a tuple type (as discussed in Section 8.4), which may be described by taking the tuple expression and replacing each element expression with its type, except that a varargs expression having type $\mathrm{OrderedGenerator}[\![T]\!]$ is replaced by $T\ldots$ (for the most specific T possible). The element expressions are all evaluated before the tuple is constructed, and if there is a varargs expression then the generator is used to construct a $\mathrm{HeapSequence}$ (described in Section 40.3) containing all the generated values; this $\mathrm{HeapSequence}$ then becomes an element of the tuple. Tuples are value objects. There are no explicit deconstructors for tuples except multiple variable declarations as discussed in Section 6.3.

13.28.1 Distinguishing a Keyword-Value Pair from an Equality Expression

Because a keyword-value pair shares a syntax with an *equality expression*, we provide rules for disambiguation:

- If an expression "$identifier = e$" has no parentheses around it, then it is an equality expression unless it is part of a tuple expression with more than one element expression.
- If the expression is in immediately surrounding parentheses with no other expression in the parentheses, then it is an equality expression unless the parenthesized expression is part of a juxtaposition sequence and is to be used as an argument to a function, in which case the parenthesized expression is a tuple expression.
- Adding parentheses makes the expression an equality expression.
- In the rare situations where "$(identifier = e)$" is treated as an equality expression and it must be a tuple expression, "$tuple(identifier = e)$" makes it a tuple expression where "$tuple$" is an identity function defined in the Fortress standard libraries (as described in Section 13.32.4).

13.29 Comprehensions

Syntax:

Comprehension	::=	{ *Expr* | *GeneratorList* }
	|	{ *Expr* $\mapsto$ *Expr* | *GeneratorList* }
	|	⟨ *Expr* | *GeneratorList* ⟩
	|	[(*ArrayComprehensionLeft* | *GeneratorList*)$^+$]
ArrayComprehensionLeft	::=	*Id* $\mapsto$ *Expr*
	|	(*Id* , *IdList*) $\mapsto$ *Expr*

Fortress provides *comprehension* syntax for several aggregate expressions (described in Section 13.28). Generators (described in Section 13.17) produce values and bind the values to the identifiers that are used in the left-hand side of the token | (*left-hand body*). Most generators, unlike the $sequential$ generator, may execute each evaluation of the left-hand body in a separate implicit thread. Comprehensions evaluate to aggregate values and have corresponding aggregate types.

A set comprehension is enclosed in braces, with a left-hand body separated by the token | from a generator list. For example, the comprehension:

$$\{\, x^2 \mid x \leftarrow \{0, 1, 2, 3, 4, 5\},\ x \texttt{ MOD } 2 = 0 \,\}$$

evaluates to the set

$\{0, 4, 16\}$

Map comprehensions are like set comprehensions, except that the left-hand body must be of the form $e_1 \mapsto e_2$. An exception is thrown if e_1 produces the same value but e_2 a different value on more than one iteration of the generator list. For example:

$\{x^2 \mapsto x^3 \mid x \leftarrow \{0, 1, 2, 3, 4, 5\},\ x \texttt{ MOD } 2 = 0\}$

evaluates to the map

$\{0 \mapsto 0, 4 \mapsto 8, 16 \mapsto 64\}$

List comprehensions are like set comprehensions, except that they are syntactically enclosed in angle brackets. For example:

$\langle x^2 \mid x \leftarrow \{0, 1, 2, 3, 4, 5\},\ x \texttt{ MOD } 2 = 0\rangle$

evaluates to the list

$\langle 0, 4, 16\rangle$

Array comprehensions are like set comprehensions, except that they are syntactically enclosed in brackets, and the left-hand body must be of the form `(`$index_1$`, `$index_2$`, ..., `$index_n$`)` $\mapsto$ e. Moreover an array comprehension may have multiple clauses separated by semicolons or line breaks. Each clause conceptually corresponds to an independent loop. Clauses are run in order. The result is an n-dimensional array. For example:

$$\begin{aligned} a = [&(x, y, 1) \mapsto 0.0 \mid x \leftarrow 1 : xSize, y \leftarrow 1 : ySize \\ &(1, y, z) \mapsto 0.0 \mid y \leftarrow 1 : ySize, z \leftarrow 2 : zSize \\ &(x, 1, z) \mapsto 0.0 \mid x \leftarrow 2 : xSize, z \leftarrow 2 : zSize \\ &(x, y, z) \mapsto x + y \cdot z \mid x \leftarrow 2 : xSize, y \leftarrow 2 : ySize, z \leftarrow 2 : zSize\,] \end{aligned}$$

13.30 Type Ascription

Syntax:

Expr ::= *Expr* `as` *TypeRef*

An expression consisting of a single subexpression, followed by the special reserved word `as`, followed by a type, is a *type ascription*. The value of the expression is the value of the subexpression. The static type of the expression is the ascripted type. The type of the subexpression must be a subtype of the ascripted type. A type ascription does not affect the dynamic type of the value the expression evaluates to (unlike a type assumption described in Section 13.31). It is usually for type inference discussed in Chapter 20; when it is impossible to infer a type for an expression, the programmer can provide type information for the expression using a type ascription.

13.31 Type Assumption

Syntax:

Expr ::= *Expr* `asif` *TypeRef*

An expression consisting of a single subexpression, followed by the special reserved word `asif`, followed by a type, is a *type assumption*. The value of the expression is the value of the subexpression. The static type of the expression is the given type. The type of the subexpression must be a subtype of the given type. A type assumption converts

the dynamic type of the value the expression evaluates to (unlike a type ascription described in Section 13.30). It is usually for accessing methods provided by supertraits. When multiple supertraits provide different methods with the same name, a subtrait may access a particular method from one of the supertraits using a type assumption. The static type of the supertrait indicates whether a definition of the method is provided by the supertrait. If the concrete method is not provided exist, it is a static error. The keyword `super` in the Java Programming Language is an example of a type assumption.

13.32 Expression-like Functions

For convenience, the Fortress standard libraries provide functions such as $cast$ and $instanceOf$ that are often provided by other programming languages.

13.32.1 Casting

Although there is no "casting" operator (equivalent to casts in the Java Programming Language) built into Fortress, the effect of a cast can be provided by the following function:

$cast[\![T]\!](x) : T =$
  `typecase` x `in`
    $T \Rightarrow x$
    `else` $\Rightarrow$ `throw` $\mathrm{CastException}$
  `end`

The function converts the type of its argument to a given type. If the static type of the argument is not a subtype of the given type, a $\mathrm{CastException}$ is thrown. For convenience, the function $cast$ is included in the Fortress standard libraries.

13.32.2 Instanceof Testing

Although there is no "instanceof" operator (equivalent to instanceof testing in the Java Programming Language) built into Fortress, the effect of an instanceof testing can be provided by the following function:

$instanceOf[\![T]\!](x) : \mathrm{Boolean} =$
  `typecase` x `in`
    $T \Rightarrow true$
    `else` $\Rightarrow$ $false$
  `end`

The function tests whether its argument has a given type and returns a boolean value. For convenience, the function $instanceOf$ is included in the Fortress standard libraries.

13.32.3 Ignoring Values

For convenience, the function $ignore$ (equivalent to the `ignore` function in the Objective Caml programming language) is included in the Fortress standard libraries:

$ignore(x) = ()$

The function discards the value of its argument and returns $()$. For example, the following:

$ignore(f\ x)$

is equivalent to:

$f\ x; ()$

13.32.4 Enforcing Tuples

An identity function $tuple$ is defined in the Fortress standard libraries to make "$(identifier = e)$" a tuple expression (as discussed in Section 13.28.1):

$tuple(x) = x$

The function returns its argument as a tuple expression.

Chapter 14

Exceptions

Exceptions are values that can be thrown and caught, via `throw` expressions (described in Section 13.25) and `catch` clauses of `try` expressions (described in Section 13.26). When a `throw` expression "`throw` e" is evaluated, the subexpression e is evaluated to an exception. The static type of e must be a subtype of $\mathrm{Exception}$. Then the `throw` expression tries to transfer control to its *dynamically containing block* (described in Chapter 4), from the innermost outward, until either (*i*) an enclosing `try` expression is reached, with a `catch` clause matching a type of the thrown exception, or (*ii*) the outermost dynamically containing block is reached.

If a matching `catch` clause is reached, the right-hand side of the first matching subclause is evaluated. If no matching `catch` clause is found before the outermost dynamically containing block is reached, the outermost dynamically containing block completes abruptly whose cause is the thrown exception.

If an enclosing `try` expression of a `throw` expression includes a `finally` clause, and the `try` expression completes abruptly, the `finally` clause is evaluated before control is transferred to the dynamically containing block.

14.1 Causes of Exceptions

Every exception is thrown for one of the following reasons:

1. A `throw` expression is evaluated.
2. An implementation resource is exceeded (e.g., an attempt is made to allocate beyond the set of available locations).

14.2 Types of Exceptions

All exceptions have type $\mathrm{Exception}$ declared as follows:

```
trait Exception comprises { CheckedException, UncheckedException }
  settable message: Maybe[[String]]
  settable chain: Maybe[[Exception]]
  printStackTrace(): ()
end
```

Every exception has either type $\mathrm{CheckedException}$ or $\mathrm{UncheckedException}$:

```
trait CheckedException
  extends { Exception }
  excludes { UncheckedException }
end

trait UncheckedException
  extends { Exception }
  excludes { CheckedException }
end
```

A functional declaration (described in Section 9.2 and Section 12.1) includes an optional `throws` clause in its header listing the CheckedExceptions (also written *checked exceptions*) that can be thrown by invocation of the functional. If a `throws` clause is not explicitly provided, the `throws` clause of the functional declaration is empty. The body of a functional is statically checked to ensure that no checked exceptions are thrown by any subexpression of the functional body other than those listed in the `throws` clause. This static check is performed by examining each `throw` expression and functional invocation I, determining the static type of the functional f invoked in I, and determining the `throws` clause of f. (If f is polymorphic, or occurs in a polymorphic context, instantiations of type variables free in the `throws` clause of f are substituted for formal type variables). For each checked exception thrown in I, the enclosing expressions of I are checked for a matching `catch` clause. The set A of all checked exceptions thrown by all invocations without a matching `catch` clause in the functional body is accumulated and compared against the `throws` clause of the enclosing functional declaration. If an exception that is not a subtype of an exception listed in the `throws` clause occurs in A, it is a static error.

A similar analysis is performed on top-level variable declarations. If it is determined that their initialization expressions can throw a checked exception, it is a static error.

14.3 Information of Exceptions

Every exception has optional fields: a message and a chained exception. These fields are default to Nothing as follows:

```
trait Exception comprises { CheckedException, UncheckedException }
  getter message(): Maybe⟦String⟧ = Nothing
  setter message(String) : ()
  getter chain(): Maybe⟦Exception⟧ = Nothing
  setter chain(Exception) : ()
  printStackTrace(): ()
end
```

where an optional value v is either Nothing or Just(v) as declared in Section 31.2. The *chain* field can be set at most once. If it is set more than once, an InvalidChainException is thrown. It is generally set when the exception is created.

When an exception is created, the execution stack of its thread at the time of the exception creation is captured in the exception. The invocation of *printStackTrace* prints the captured stack trace. There is no way to update the captured stack trace. If a programmer wants to catch a thrown exception and rethrow it, and capture the stack trace at the time of the second throwing of the exception, the programmer has to create a new exception (perhaps with the original exception as its *chain* field).

When an exception is thrown, its *message* and *chain* fields may be set. For example, if a checked exception is caught in a `catch` clause, and the `catch` clause in turn throws an unchecked exception, the unchecked exception can be chained so that an examination of the unchecked exception reveals information about the original exception. For example:

```
read(fileName) =
  try
    readFile(fileName)
  catch e
    IOException ⇒ throw Error("This code can't handle IOExceptions", e)
  end
```

where the *message* and *chain* fields of Error are set to "`This code can't handle IOExceptions`" and IOException respectively.

By default, a `forbid` clause in a `try` expression throws a new ForbiddenException by ***chaining*** the exception thrown by the `try` block in the `try` expression that is a subtype of the exception type listed in the `forbid` clause. For example, the following *read* function:

```
read(fileName) =
  try
    readFile(fileName)
  forbid IOException
  end
```

is equivalent to:

```
read(fileName) =
  try
    readFile(fileName)
  catch e
    IOException ⇒throw ForbiddenException(e)
  end
```

where the *chain* of ForbiddenException is set to IOException.

Chapter 15

Overloading and Multiple Dispatch

Fortress allows functions and methods (collectively called as *functionals*) to be *overloaded*; that is, there may be multiple declarations for the same functional name visible in a single scope (which may include inherited method declarations), and several of them may be applicable to any particular functional call.

Calls to overloaded functionals are checked via a preliminary static dispatch on the static type of the argument, followed by dynamic dispatch on the runtime type of the argument. Fortress imposes restrictions (described in Chapter 33) on overloaded functional declarations that ensure there exists a unique most specific declaration for every call. Thus, it is unambiguous at run time which declaration should be applied at run time.

In this chapter, we describe how to determine which declarations are *applicable* to a particular functional call, and when several are applicable, how to select among them. Section 15.1 introduces some terminology and notation. In Section 15.2, we show how to determine which declarations are applicable to a *named functional call* (a function call described in Section 13.6 or a naked method invocation described in Section 13.5) when all declarations have only ordinary parameters (without varargs or keyword parameters). We discuss how to handle dotted method calls (described in Section 13.4) in Section 15.3, and declarations with varargs and keyword parameters in Section 15.4. Determining which declaration is applied, if several are applicable, is discussed in Section 15.5.

15.1 Terminology and Notation

When there are two or more function declarations of the same name within a single lexical scope, we say that the function name is *overloaded* within that lexical scope; we also say that each of the function declarations is overloaded, and that any pair of the function declarations are *mutually overloaded*. Top-level function declarations in a component are permitted to be overloaded with function declarations imported from APIs (using `import` $functionNames$ `from` $apiName$). Likewise, it is permitted to have two or more method declarations (declared or inherited) of the same method name within a single trait or object declaration; we say that the method name is *overloaded* within that trait or object declaration, and we also say that each of the method declarations is overloaded, and that any pair of the method declarations are *mutually overloaded*.

We assume throughout this chapter that all static variables have been instantiated or inferred. Although there may be multiple declarations with the same functional name, it is a static error for their static parameters to differ (up to α-equivalence), or for one declaration to have static parameters and another to not have them. Hence, static parameters do not enter into the determination of which declarations are applicable, so we ignore them for most of this chapter.

Recall from Chapter 8 that we write $T \preceq U$ when T is a subtype of U, and $T \prec U$ when $T \preceq U$ and $T \neq U$.

15.2 Applicability to Named Functional Calls

In this section, we show how to determine which declarations are applicable to a named functional call when all declarations have only ordinary parameters (i.e., neither varargs nor keyword parameters).

For the purpose of defining applicability, a named functional call can be characterized by the name of the functional and its argument type. Recall that a functional has a single parameter, which may be a tuple (a dotted method has a receiver as well). We abuse notation by using *static call* $f(A)$ to refer to a named functional call with name f and whose argument has static type A, and *dynamic call* $f(X)$ to refer to a named functional call with name f and whose argument, when evaluated, has dynamic type X. (Note that if the type system is sound—and we certainly hope that it is!—then $X \preceq A$ for any call to f.) We use the term *call* $f(C)$ to refer to static and dynamic calls collectively.

We also use *function declaration* $f(P) : U$ to refer to a function declaration with function name f and whose parameter type is P and return type is U.

For method declarations, we must take into account the self parameter, which we do as follows:

A *dotted method declaration* $P_0.f(P) : U$ is a dotted method declaration with name f, where P_0 is the trait or object type in which the declaration appears, P is the parameter type, and U is the return type. (Note that despite the suggestive notation, a dotted method declaration need not explicitly list its self parameter.)

A *functional method declaration* $f(P) : U$ *with self parameter at* i is a functional method declaration with name f, with a parameter that has `self` in the ith position, parameter type P, and return type U. Note that the static type of the self parameter is the trait or object trait type in which the declaration $f(P) : U$ occurs. In the following, we will use P_i to refer to the ith element of P.

We elide the return type of a declaration, writing $f(P)$ and $P_0.f(P)$, when the return type is not relevant to the discussion.

A declaration $f(P)$ is *applicable* to a call $f(C)$ if the call is in the scope of the declaration and $C \preceq P$. (See Chapter 7 for the definition of scope.)

Note that a named functional call $f(C)$ may invoke a dotted method declaration if the declaration is provided by the trait or object enclosing the call. To account for this we rewrite a named functional call $f(C)$ to $C_0.f(C)$ where the type C_0 is declared by the trait or object declaration immediately enclosing the call if there are no declarations applicable to $f(C)$. We then use the rule for applicability to dotted method calls (described in Section 15.3) to determine which declarations are applicable to $C_0.f(C)$.

15.3 Applicability to Dotted Method Calls

Dotted method applications can be characterized similarly to named functional applications, except that, analogously to dotted method declarations, we use A_0 to denote the static type of the receiver object, and, as for named functional calls, A to denote the static type of the argument of a static dotted method call; we use X_0 and X similarly for dynamic dotted method calls. We write $A_0.f(A)$ and $X_0.f(X)$ to refer to the static and dynamic calls respectively. A dotted method call $C_0.f(C)$ refers to static and dynamic calls collectively.

A dotted method declaration $P_0.f(P)$ is *applicable* to a dotted method call $C_0.f(C)$ if $C_0 \preceq P_0$ and $C \preceq P$.

15.4 Applicability for Functionals with Varargs and Keyword Parameters

The basic idea for handling varargs and keyword parameters is that we can think of a functional declaration that has such parameters as though it were (possibly infinitely) many declarations, one for each set of arguments it may

be called with. In other words, we expand these declarations so that there exists a declaration for each number of arguments that can be passed to it.

A declaration with k keyword parameters corresponds to 2^k declarations which cannot be called with elided arguments, one for each subset of the keyword parameters. For example, the following declaration:

$f(x = 5, y = 6, z = 7) : \mathbb{Z}$

would be expanded into:

$f(x = 5, y = 6, z = 7) : \mathbb{Z}$
$f(x = 5, y = 6) : \mathbb{Z}$
$f(x = 5, z = 7) : \mathbb{Z}$
$f(y = 6, z = 7) : \mathbb{Z}$
$f(x = 5) : \mathbb{Z}$
$f(y = 6) : \mathbb{Z}$
$f(z = 7) : \mathbb{Z}$
$f() : \mathbb{Z}$

Note that even though expanded declarations still have keyword parameters, they cannot be called with elided arguments any more. A declaration with keyword parameters is applicable to a call if any one of the expanded declarations is applicable.

A declaration with a varargs parameter corresponds to an infinite number of declarations, one for every number of arguments that may be passed to the varargs parameter. In practice, we can bound that number by the maximum number of arguments that the functional is called with anywhere in the program (in other words, a given program will contain only a finite number of calls with different numbers of arguments). The expansion described here is a conceptual one to simplify the description of the semantics; we do not expect any real implementation to actually expand these declarations at compile time. For example, the following declaration:

$f(x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z} \ldots) : \mathbb{Z}$

would be expanded into:

$f(x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z} \ldots) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}, z_2 : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}, z_2 : \mathbb{Z}, z_3 : \mathbb{Z}) : \mathbb{Z}$
$\ldots$

Notice that the expansion includes the original declaration. This declaration is retained to account for the case when a tuple expression with a varargs expression is passed as an argument to a call; even though this declaration still has a varargs parameter, it's called with a fixed number of arguments. A declaration with a varargs parameter is applicable to a call if any one of the expanded declarations is applicable.

15.5 Overloading Resolution

Several declarations may be applicable to a given functional call. Therefore, it is necessary to determine which declaration is dispatched to. The basic principle is that, for any functional call, we wish to identify a unique declaration that is the most specific among all declarations applicable to the call at run time. If there is no such declaration, then the call is *undefined*, which is a static error. If there are two or more such declarations, no one of which is more specific than all the others, the call is said to be *ambiguous*, which is also a static error. As discussed in Chapter 33, it is a static error for overloaded declarations to admit ambiguous calls at run time, whether such calls actually appear in the program or not.

If several declarations are applicable to a particular call, we determine which is most specific by using the subtype relation to compare parameter types. Formally, a declaration $f(P)$ is more specific than a declaration $f(Q)$ if $P \prec Q$. Similarly, a declaration $P_0.f(P)$ is more specific than a declaration $Q_0.f(Q)$ if $(P_0, P) \prec (Q_0, Q)$.

Chapter 16

Operators

Operators are like functions or methods; operator declarations are described in Chapter 34 and operator applications are described in Section 13.8. Just as functions or methods may be overloaded (see Chapter 15 for a discussion of overloading), so operators may have overloaded declarations, of the same or differing fixities. Calls to overloaded operators are resolved first via the fixity of the operators based on the context of the calls. Then, among the applicable declarations with that fixity, the most specific declaration is chosen.

Most operators can be used as prefix, infix, postfix, or nofix operators as described in Section 16.3; the fixity of an operator is determined syntactically, and the same operator may have declarations for multiple fixities. A simple example is that '$-$' may be either infix or prefix, as is conventional. As another example, the Fortress standard libraries define '!' to be a postfix operator that computes factorial when applied to integers. These operators may not be used as enclosing operators.

Several pairs of operators can be used as enclosing operators. Any number of '|' (vertical line) can be used as both infix operators and enclosing operators.

Some operators are always postfix: a '^' followed by any ordinary operator (with no intervening whitespace) is considered to be a superscripted postfix operator. For example, '$\hat{}*$' and '$\hat{}+$' and '$\hat{}?$' are available for use as part of the syntax of extended regular expressions. As a very special case, '$\hat{}T$' is also considered to be a superscripted postfix operator, typically used to signify matrix transposition.

Finally, there are special operators such as juxtaposition and operators on dimensions and units. Juxtaposition may be a function application, a numeral concatenation, or an infix operator in Fortress. When the left-hand-side expression is a function, juxtaposition performs function application; when the left-hand-side expression is a number, juxtaposition conventionally performs multiplication; when the left-hand-side expression is a string, juxtaposition conventionally performs string concatenation. Fortress provides several operators on dimensions and units as described in Chapter 18.

16.1 Operator Names

To support a rich mathematical notation, Fortress allows most Unicode characters that are specified to be mathematical operators to be used as operators in Fortress expressions, as well as these characters and character combinations:

```
!   @   #   $   %   *   +   -   =   |   :   <   >   /   ?   ^   ~
->   -->   =>   ==>   <=   >=   =/=   **   !!   ||   |||
<<   <<<   >>   >>>   <->   <-/-   -/->   <=>   ===
```

In addition, a token that is made up of a mixture of uppercase letters and underscores (but no digits), does not begin or end with an underscore, and contains at least two different letters is also considered to be an operator:

`MAX` `MIN` `SQRT` `TIMES`

The above operators are rendered as: `MAX` `MIN` $\sqrt{}$ $\times$. Some of these uppercase tokens are considered to be equivalent to single Unicode characters, but even those that are not can still be used as operators. (See Appendix F for a detailed description of operator names in Fortress.)

16.2 Operator Precedence

Fortress specifies that certain operators have higher precedence than certain other operators, so that one need not use parentheses in all cases where operators are mixed in an expression. (See Appendix F for a detailed description of operator precedence in Fortress.) However, Fortress does not follow the practice of other programming languages in simply assigning an integer to each operator and then saying that the precedence of any two operators can be compared by comparing their assigned integers. Instead, Fortress relies on defining traditional groups of operators based on their meaning and shape, and specifies specific precedence relationships between some of these groups. If there is no specific precedence relationship between two operators, then parentheses must be used. For example, Fortress does not accept the expression $a + b \cup c$; one must write either $(a + b) \cup c$ or $a + (b \cup c)$. (Whether or not the result then makes any sense depends on what definitions have been made for the $+$ and $\cup$ operators—see Chapter 34.)

Here are the basic principles of operator precedence in Fortress:

- Member selection (`.`) and method invocation ($.name(\ldots)$) are not operators. They have higher precedence than any operator listed below.
- Subscripting (`[ ]`), superscripting (`^`), and postfix operators have higher precedence than any operator listed below; within this group, these operations are left-associative (performed left-to-right).
- *Tight juxtaposition*, that is, juxtaposition without intervening whitespace, has higher precedence than any operator listed below. The associativity of tight juxtaposition is type-dependent; see Section 16.7.
- Next, *tight fractions*, that is, the use of the operator '$/$' with no whitespace on either side, have higher precedence than any operator listed below. The tight-fraction operator has no precedence compared with itself, so it is not permitted to be used more than once in a tight fraction without use of parentheses.
- *Loose juxtaposition*, that is, juxtaposition with intervening whitespace, has higher precedence than any operator listed below. The associativity of loose juxtaposition is type-dependent and is different from that for tight juxtaposition; see Section 16.7. Note that *lopsided juxtaposition* (having whitespace on one side but not the other) is a static error as described in Section 16.3.
- Prefix operators have higher precedence than any operator listed below.
- The infix operators are partitioned into certain traditional groups, as explained below. They have higher precedence than any operator listed below.
- The equal symbol '$=$' in binding context, the assignment operator '$:=$', and compound assignment operators ($+=$, $-=$, $\wedge=$, $\vee=$, $\cap=$, $\cup=$, and so on as described in Section 13.8) have lower precedence than any operator listed above. Note that compound assignment operators themselves are not operator names.

The infix binary operators are divided into four general categories: arithmetic, relational, boolean, and other. The arithmetic operators are further categorized as multiplication/division/intersection, addition/subtraction/union, and other. The relational operators are further categorized as equivalence, inequivalence, chaining, and other. The boolean operators are further categorized as conjunctive, disjunctive, and other.

The arithmetic and relational operators are further divided into groups based on shape:

- "ordinary" operators: $+ - \cdot \times / \pm \mp \oplus \ominus \odot \otimes \oslash \boxplus \boxminus \boxdot \boxtimes$ $< \leq \geq > \ll \lll \ggg \gg \not< \not\leq \not\geq \not>$ etc.

  The arithmetic operations in this group are further subdivided into "plain" ($+ - \cdot \times / \pm \mp$ etc.), "circled" ($\oplus \ominus \odot \otimes \oslash$ etc.), "boxed" ($\boxplus \boxminus \boxdot \boxtimes$ etc.), and so on; any of these groups may be used with the plain relational operators ($< \leq \geq > \ll \lll \ggg \gg \not< \not\leq \not\geq \not>$ etc.), but the groups may not be mixed.

- "rounded horseshoe" or "set" operators: $\cap \Cap \cup \Cup \uplus \subset \subseteq \supseteq \supset \Subset \Supset \not\subset \not\subseteq \not\supseteq \not\supset$ etc.

- "square horseshoe" operators: $\sqcap \sqcup \sqsubset \sqsubseteq \sqsupseteq \sqsupset \not\sqsubseteq \not\sqsupseteq$ etc.

- "curly" operators: $\curlywedge \curlyvee \prec \preceq \succeq \succ \not\prec \not\preceq \not\succeq \not\succ$ etc.

- "triangular" relations: $\lhd \trianglelefteq \trianglerighteq \rhd \not\lhd \not\trianglelefteq \not\trianglerighteq \not\rhd$ etc.

- "chickenfoot" relations: $<\!\!\text{-}\!\!>$ etc.

The principles of precedence for binary operators are then as follows:

- A multiplication or division or intersection operator has higher precedence than any addition or subtraction or union operator that is in the same shape group.

- Certain addition and subtraction operators come in pairs, such as $+$ and $-$, or $\oplus$ and $\ominus$, which are considered to have the same precedence and so may be mixed within an expression and are grouped left-associatively. These addition-subtraction pairs are the *only* cases where two different operators are considered to have the same precedence.

- An arithmetic operator has higher precedence than any equivalence or inequivalence operator.

- An arithmetic operator has higher precedence than any relational operator that is in the same shape group.

- A relational operator has higher precedence than any boolean operator.

- A conjunctive boolean operator has higher precedence than any disjunctive boolean operator.

While the rules of precedence are complicated, they are intended to be both unsurprising and conservative. Note that operator precedence in Fortress is not always transitive; for example, while $+$ has higher precedence than $<$ (so you can write $a + b < c$ without parentheses), and $<$ has higher precedence than $\vee$ (so you can write $a < b \vee c < d$ without parentheses), it is *not* true that $+$ has higher precedence than $\vee$—the expression $a \vee b + c$ is not permitted, and one must instead write $(a \vee b) + c$ or $a \vee (b + c)$.

Another point is that the various multiplication and division operators do *not* have "the same precedence"; they may not be mixed freely with each other. For example, one cannot write $u \cdot v \times w$; one must write $(u \cdot v) \times w$ or (more likely) $u \cdot (v \times w)$. Similarly, one cannot write $a \cdot b \,/\, c \cdot d$; but juxtaposition does bind more tightly than a loose (whitespace-surrounded) division slash, so one is allowed to write $a\,b \,/\, c\,d$, and this means the same as $(a\,b)/(c\,d)$. On the other hand, loose juxtaposition binds less tightly than a tight division slash, so that $a\,b/c\,d$ means the same as $a\,(b/c)\,d$. On the other other hand, tight juxtaposition binds more tightly than tight division, so that $(n+1)/(n+2)(n+3)$ means the same as $(n+1)/((n+2)(n+3))$.

There are two additional rules intended to catch misleading code: it is a static error for an operand of a tight infix or prefix operator to be a loose juxtaposition, and it is a static error if the rules of precedence determine that a use of infix operator a has higher precedence than a use of infix operator b, but that particular use of a is loose and that particular use of b is tight. Thus, for example, the expression $\sin\ x\ +\ y$ is permitted, but $\sin\ x + y$ is not permitted. Similarly, the expression $a \cdot b + c$ is permitted, as are $a{\cdot}b + c$ and $a{\cdot}b{+}c$, but $a \cdot b{+}c$ is not permitted. (The rule detects only the presence or absence of whitespace, not the amount of whitespace, so $a \ \cdot b + c$ is permitted. You have to draw the line somewhere.)

When in doubt, just use parentheses. If there's a problem, the compiler will (probably) let you know.

16.3 Operator Fixity

Most operators in Fortress can be used variously as prefix, postfix, infix, nofix, or multifix operators. (See Section 16.4 for a discussion of how infix operators may be chained or treated as multifix operators.) Some operators can be used in pairs as enclosing (bracketing) operators—see Section 16.5. The Fortress language dictates only the rules of syntax; whether an operator has a meaning when used in a particular way depends only on whether there is a definition in the program for that operator when used in that particular way (see Chapter 34).

The fixity of a non-enclosing operator is determined by context. To the left of such an operator we may find (1) a *primary* expression (described below), (2) another operator, or (3) a comma, semicolon, or left encloser. To the right we may find (1) a primary expression, (2) another operator, (3) a comma, semicolon, or right encloser, or (4) a line break. A primary expression is an identifier, a literal, an expression enclosed by matching enclosers, a field selection, or an expression followed by a postfix operator. Considered in all combinations, this makes twelve possibilities. In some cases one must also consider whether or not whitespace separates the operator from what lies on either side. The rules of operator fixity are specified by Figure 16.1, where the center column indicates the fixity that results from the left and right context specified by the other columns.

left context	**whitespace**	**operator fixity**	**whitespace**	**right context**
primary	yes	**infix**	yes	primary
	yes	**error** (infix)	no	
	no	**postfix**	yes	
	no	**infix**	no	
primary	yes	**infix**	yes	operator
	yes	**error** (infix)	no	
	no	**postfix**	yes	
	no	**infix**	no	
primary	yes	**error** (postfix)		, ; right encloser
	no	**postfix**		
primary	yes	**infix**		line break
	no	**postfix**		
operator		**prefix**		primary
operator		**prefix**		operator
operator		**error** (nofix)		, ; right encloser
operator		**error** (nofix)		line break
, ; left encloser		**prefix**		primary
, ; left encloser		**prefix**		operator
, ; left encloser		**nofix**		, ; right encloser
, ; left encloser		**error** (prefix)		line break

Figure 16.1: Operator Fixity (I)

A case described in the center column of the table as an **error** is a static error; for such cases, the fixity mentioned in parentheses is the recommended treatment of the operator for the purpose of attempting to continuing the parse in search of other errors.

The table may seem complicated, but it all boils down to a couple of practical rules of thumb:

1. *Any* operator can be prefix, postfix, infix, or nofix.
2. An infix operator can be *loose* (having whitespace on both sides) or *tight* (having whitespace on neither side), but it mustn't be *lopsided* (having whitespace on one side but not the other).
3. A postfix operator should have no whitespace before it and should be followed (possibly after some whitespace) by a comma, semicolon, right encloser, or line break.

left context	whitespace	operator fixity	whitespace	right context
primary	yes yes no no	**infix** **left encloser** **right encloser** **infix**	yes no yes no	primary
primary	yes yes no no	**infix** **left encloser** **right encloser** **infix**	yes no yes no	operator
primary	yes no	**error** (right encloser) **right encloser**		, ; right encloser
primary	yes no	**infix** **right encloser**		line break
operator		**error** (left encloser) **left encloser**	yes no	primary
operator		**error** (left encloser) **left encloser**	yes no	operator
operator		**error** (nofix)		, ; right encloser
operator		**error** (nofix)		line break
, ; left encloser		**left encloser**		primary
, ; left encloser		**left encloser**		operator
, ; left encloser		**nofix**		, ; right encloser
, ; left encloser		**error** (left encloser)		line break

Figure 16.2: Operator Fixity (II)

16.4 Chained and Multifix Operators

Certain infix mathematical operators that are traditionally regarded as *relational* operators, delivering boolean results, may be *chained*. For example, an expression such as $A \subseteq B \subset C \subseteq D$ is treated as being equivalent to $(A \subseteq B) \wedge (B \subset C) \wedge (C \subseteq D)$ except that the expressions B and C are evaluated only once (which matters only if they have side effects such as writes or input/output actions). Fortress restricts such chaining to operators of the same kind and having the same sense of monotonicity; for example, neither $A \subseteq B \leq C$ nor $A \subseteq B \supset C$ is permitted. Equivalence operators may be mixed into a chain; for example, one may write $A \subseteq B = C \subseteq D$. This transformation is done before type checking. In particular, it is done even though these operators do not return boolean values, and the resulting expression is checked for type correctness. (See Section F.4 for a detailed description of which operators may be chained.)

Any infix operator that does not chain may be treated as *multifix*. If $n-1$ occurrences of the same operator separate n operands where $n \geq 3$, then the compiler first checks to see whether there is a definition for that operator that will accept n arguments. If so, that definition is used; if not, then the operator is treated as left-associative and the compiler looks for a two-argument definition for the operator to use for each occurrence. As an example, the cartesian product $S_1 \times S_2 \times \cdots \times S_n$ of n sets may usefully be defined as a multifix operator, but ordinary addition $p + q + r + s$ is normally treated as $((p + q) + r) + s$.

16.5 Enclosing Operators

These operators are always used in pairs as enclosing operators:

(/ /) (\ \)

```
[   ]        [/   /]                        [*   *]
{   }        {/   /}        {\   \}         {*   *}
              </   />        <\   \>
             <</   />>      <<\   \>>
```

(ASCII encodings are shown here; they all correspond to particular single Unicode characters.) There are other pairs as well, such as ⌊ ⌋ and ⌈ ⌉. Note that the pairs ( ) and [\ \] (also known as ⟦ ⟧) are not operators; they play special roles in the syntax of Fortress, and their behavior cannot be redefined by a library. The bracket pairs that may be used as enclosing operators are described in Section F.1.

Any number of '|' (vertical line) may also be used in pairs as enclosing operators but there is a trick to it, because on the face of it you can't tell whether any given occurrence is a left encloser or a right encloser. Again, context is used to decide, this time according to Figure 16.2.

This is very similar to Figure 16.1 in Section 16.3; a rough rule of thumb is that if an ordinary operator would be considered a prefix operator, then one of these will be considered a left encloser; and if an ordinary operator would be considered a postfix operator, then one of these will be considered a right encloser.

In this manner, one may use |...| for absolute values and ||...|| for matrix norms.

16.6 Conditional Operators

If a binary operator other than ':' is immediately followed by a ':' then it is *conditional*: evaluation of the right-hand operand cannot begin until evaluation of the left-hand operand has completed, and whether or not the right-hand operand is evaluated may depend on the value of the left-hand operand. If the left-hand operand throws an exception, then the right-hand operand is not evaluated.

The Fortress standard libraries define two conditional operators on boolean values, ∧: and ∨: (see Section 16.8.15).

See Section 34.8 for a discussion of how conditional operators are implemented.

16.7 Juxtaposition

Fortress provides several kinds of juxtaposition: juxtaposition may be a function call, a numeral concatenation, or a special infix operator. See Section 25.1 for an example declaration of a `juxtaposition` operator.

When two expressions are juxtaposed, the juxtaposition is interpreted as follows: if the left-hand-side expression is a function, juxtaposition performs function application; if the left-hand-side expression is a number and the right-hand-side expression is also a number, juxtaposition performs numeral concatenation; otherwise, juxtaposition performs the `juxtaposition` operator application.

The manner in which a juxtaposition of three or more items should be associated requires type information and awareness of whitespace. (This is an inherent property of customary mathematical notation, which Fortress is designed to emulate where feasible.) Therefore a Fortress compiler must produce a provisional parse in which such multi-element juxtapositions are held in abeyance, then perform a type analysis on each element and use that information to rewrite the n-ary juxtaposition into a tree of binary juxtapositions.

All we need to know is whether the static type of each element of a juxtaposition is an arrow type. There are actually three legitimate possibilities for each element of a juxtaposition: (a) it has an arrow type, in which case it is considered to be a function element; (b) it has a type that is not an arrow type, in which case it is considered to be a non-function element; (c) it is an identifier that has no visible declaration, in which case it is considered to be a function element (and everything will work out okay if it turns out to be the name of an appropriate functional method).

The rules below are designed to forbid certain forms of notational ambiguity that can arise if the name of a functional method happens to be used also as the name of a variable. For example, suppose that trait T has a functional method of one parameter named n; then in the code

```
do
  a: T = t
  n: ℤ = 14
  z = na
end
```

it might not be clear whether the intended meaning was to invoke the functional method n on a or to multiply a by 14. The rules specify that such a situation is a static error.

The rules for reassociating a loose juxtaposition are as follows:

- First the loose juxtaposition is broken into nonempty chunks; wherever there is a non-function element followed by a function element, the latter begins a new chunk. Thus a chunk consists of some number (possibly zero) of functions followed by some number (possibly zero) of non-functions.
- It is a static error if any non-function element in a chunk is an unparenthesized identifier f and is followed by another non-function element whose type is such that f can be applied to that latter element as a functional method.
- The non-functions in each chunk, if any, are replaced by a single element consisting of the non-functions grouped left-associatively into binary juxtapositions.
- What remains in each chunk is then grouped right-associatively.
- It is a static error if an element of the original juxtaposition was the last element in its chunk before reassociation, the chunk was not the last chunk (and therefore the element in question is a non-function element), the element was an unparenthesized identifier f, and the type of the following chunk after reassociation is such that f can be applied to that following chunk as a functional method.
- Finally, the sequence of rewritten chunks is grouped left-associatively.

(Notice that no analysis of the types of newly constructed chunks is needed during this process.)

Here is an example: $n\ (n+1)\ \sin\ 3\ n\ x\ \log\ \log\ x$. Assuming that $\sin$ and $\log$ name functions in the usual manner and that n, $(n+1)$, and x are not functions, this loose juxtaposition splits into three chunks: $n\ (n+1)$ and $\sin\ 3\ n\ x$ and $\log\ \log\ x$. The first chunk has only two elements and needs no further reassociation. In the second chunk, the non-functions $3\ n\ x$ are replaced by $((3\ n)\ x)$. In the third chunk, there is only one non-function, so that remains unchanged; the chunk is the right-associated to form $(\log\ (\log\ x))$. Finally, the three chunks are left-associated, to produce the final interpretation $((n\ (n+1))\ (\sin\ ((3\ n)\ x)))\ (\log\ (\log\ x))$. Now the original juxtaposition has been reduced to binary juxtaposition expressions.

The rules for reassociating a tight juxtaposition follow a different strategy:

- If the tight juxtaposition contains no function element, or if only the last element is a function, go on to the next step. Otherwise, consider the leftmost function element and examine the element that follows it. If that latter element is not parenthesized, it is a static error; otherwise, replace the two elements with a single element consisting of a new juxtaposition of the two elements (in the same order), and perform a type analysis on this new juxtaposition. (At this point, it is a static error if this new juxtaposition is preceded in the overall juxtaposition by a non-function element that is an unparenthesized identifier f, and the type of the new juxtaposition is such that f can be applied to it as a functional method.) Then repeat this step on the original juxtaposition (which is now one element shorter).
- The overall juxtaposition now either is a single element or consists entirely of non-function elements. It is a static error the overall juxtaposition now contains a non-function element that is an unparenthesized identifier f, and the type of the following element is such that f can be applied to it as a functional method.

- Left-associate the remaining elements of the juxtaposition.

(Note that this process requires type analysis of newly created chunks along the way.)

Here is an (admittedly contrived) example: $reduce(f)(a)(x+1)sqrt(x+2)$. Suppose that $reduce$ is a curried function that accepts a function f and returns a function that can be applied to an array a (the idea is to use the function f, which ought to take two arguments, to combine the elements of the array to produce an accumulated result).

The leftmost function is $reduce$, and the following element (f) is parenthesized, so the two elements are replaced with one: $(reduce(f))(a)(x+1)sqrt(x+2)$. Now type analysis determines that the element $(reduce(f))$ is a function.

The leftmost function is $(reduce(f))$, and the following element (a) is parenthesized, so the two elements are replaced with one: $((reduce(f))(a))(x+1)sqrt(x+2)$. Now type analysis determines that the element $((reduce(f))(a))$ is not a function.

The leftmost function is $(sqrt)$, and the following element $(x+2)$ is parenthesized, so the two elements are replaced with one: $((reduce(f))(a))(x+1)(sqrt(x+2))$. Now type analysis determines that the element $(sqrt(x+2))$ is not a function.

There are no functions remaining in the juxtaposition, so the remaining elements are left-associated:

$$(((reduce(f))(a))(x+1))(sqrt(x+2))$$

Now the original juxtaposition has been reduced to binary juxtaposition expressions.

16.8 Overview of Operators in the Fortress Standard Libraries

This section provides a high-level overview of the operators in the Fortress standard libraries. See Appendix F for the detailed rules for the operators provided by the Fortress standard libraries.

16.8.1 Prefix Operators

For all standard numeric types, the prefix operator $+$ simply returns its argument and the prefix operator $-$ returns the negative of its argument.

The operator $\neg$ is the logical NOT operator on boolean values and boolean intervals.

The operator $\dashv$ computes the bitwise NOT of an integer.

Big operators such as $\sum$ begin a *reduction expression* (Section 13.18). The big operators include $\sum$ (summation) and $\prod$ (product), along with $\bigcap$, $\bigcup$, $\bigwedge$, $\bigvee$, $\underline{\bigvee}$, $\bigoplus$, $\bigotimes$, $\biguplus$, $\boxplus$, $\boxtimes$, `MAX`, `MIN`, and so on.

16.8.2 Postfix Operators

The operator ! computes factorial; the operator !! computes double factorials. They may be applied to a value of any integral type and produces a result of the same type.

When applied to a floating-point value x, $x!$ computes $\Gamma(1+x)$, where Γ is the Euler gamma function.

16.8.3 Enclosing Operators

When used as left and right enclosing operators, $\lvert \quad \rvert$ computes the absolute value or magnitude of any number, and is also used to compute the number of elements in an aggregate, for example the cardinality of a set or the length of a list. Similarly, $\| \quad \|$ is used to compute the norm of a vector or matrix.

The floor operator $\lfloor \quad \rfloor$ and ceiling operator $\lceil \quad \rceil$ may be applied to any standard integer, rational, or real value (their behavior is trivial when applied to integers, of course). The operators hyperfloor $\llbracket x \rrbracket = 2^{\lfloor \log_2 x \rfloor}$, hyperceiling $\llbracket x \rrbracket = 2^{\lceil \log_2 x \rceil}$, hyperhyperfloor $\lll x \ggg = 2^{\llbracket \log_2 x \rrbracket}$, and hyperhyperceiling $\lll x \ggg = 2^{\llbracket \log_2 x \rrbracket}$ are also available.

16.8.4 Exponentiation

Given two expressions e and e' denoting numeric quantities v and v' that are not vectors or matrices, the expression $e^{e'}$ denotes the quantity obtained by raising v to the power v'. This operation is defined in the usual way on numerals.

Given an expression e denoting a vector and an expression e' denoting a value of type $\mathbb{Z}$, the expression $e^{e'}$ denotes repeated vector multiplication of e by itself e' times.

Given an expression e denoting a square matrix and an expression e' denoting a value of type $\mathbb{Z}$, the expression $e^{e'}$ denotes repeated matrix multiplication of e by itself e' times.

16.8.5 Superscript Operators

The superscript operator $\hat{} T$ transposes a matrix. It also converts a column vector to a row vector or a row vector to a column vector.

16.8.6 Subscript Operators

Subscripting of arrays and other aggregates is written using square brackets:

`a[i]`	*is displayed as*	a_i	ith element of one-dimensional array a
`m[i,j]`	*is displayed as*	m_{ij}	i, jth element of two-dimensional matrix m
`space[i,j,k]`	*is displayed as*	$space_{ijk}$	i, j, kth element of three-dimensional array $space$
`a[3] := 4`	*is displayed as*	$a_3 := 4$	assign 4 to the third element of mutable array a
`m["foo"]`	*is displayed as*	m“`foo`”	fetch the entry associated with string “`foo`” from map m

16.8.7 Multiplication, Division, Modulo, and Remainder Operators

For most integer, rational, floating-point, complex, and interval expressions, multiplication can be expressed using any of ‘$\cdot$’ or ‘$\times$’ or simply juxtaposition. There are, however, two subtle points to watch out for. First, juxtaposition of numerals is treated as literal concatenation rather than multiplication; in this way one can use spaces to separate groups of digits, for example $1\ 234\ 567.890\ 12$ rather than 123457.89012. Second, the $\times$ operator is used to express the shape of matrices, so if expressions using multiplication are used in expressing the shape of a matrix, it may be necessary to avoid the use of ’$\times$’ to express multiplication, or to use parentheses.

For integer, rational, floating-point, complex, and interval expressions, division is expressed by $/$. When the operator $/$ is used to divide one integer by another, the result is rational. The operator $\div$ performs truncating integer division: $m \div n = \mathit{signum}\left(\frac{m}{n}\right) \left\lfloor \left| \frac{m}{n} \right| \right\rfloor$. The operator `REM` gives the remainder from such a truncating division: $m \,\texttt{REM}\, n = m - n(m \div n)$. The operator `MOD` gives the remainder from a floor division: $m \,\texttt{MOD}\, n = m - n \left\lfloor \frac{m}{n} \right\rfloor$;

when $n > 0$ this is the usual modulus computation that evaluates integer m to an integer k such that $0 \leq k < n$ and n evenly divides $m - k$.

The special operators `DIVREM` and `DIVMOD` each return a pair of values, the quotient and the remainder; m `DIVREM` n returns $(m \div n, m \texttt{ REM } n)$ while m `DIVMOD` n returns $(\lfloor \frac{m}{n} \rfloor, m \texttt{ MOD } n)$.

Multiplication of a vector or matrix by a scalar is done with juxtaposition, as is multiplication of a vector by a matrix (on either side). Vector dot product is expressed by '$\cdot$' and vector cross product by '$\times$'. Division of a matrix or vector by a scalar may be expressed using '$/$'.

The syntactic interaction of juxtaposition, $\cdot$, $\times$, and $/$ is subtle. See Section 16.2 for a discussion of the relative precedence of these operations and how precedence may depend on the use of whitespace.

The handling of overflow depends on the type of the number produced. For integer results, overflow throws an $\mathrm{IntegerOverflowException}$. Rational computations do not overflow. For floating-point results, overflow produces $+\infty$ or $-\infty$ according to the rules of IEEE 754. For intervals, overflow produces an appropriate containing interval.

Underflow is relevant only to floating-point computations and is handled according to the rules of IEEE 754.

The handling of division by zero depends on the type of the number produced. For integer results, division by zero throws a $\mathrm{DivideByZeroException}$. For rational results, division by zero produces $1/0$. For floating-point results, division by zero produces a NaN value according to the rules of IEEE 754. For intervals, division by zero produces an appropriate containing interval (which under many circumstances will be the interval of all possible real values and infinities).

Wraparound multiplication on fixed-size integers is expressed by $\dot{\times}$. Saturating multiplication on fixed-size integers is expressed by $\boxdot$ or $\boxtimes$. These operations do not overflow.

Ordinary multiplication and division of floating-point numbers always use the IEEE 754 "round to nearest" rounding mode. This rounding mode may be emphasized by using the operators $\otimes$ (or $\odot$) and $\oslash$. Multiplication and division in "round toward zero" mode may be expressed with $\boxtimes$ (or $\boxdot$) and $\boxslash$. Multiplication and division in "round toward positive infinity" mode may be expressed with $\overset{\blacktriangle}{\boxtimes}$ (or $\overset{\blacktriangle}{\boxdot}$) and $\overset{\blacktriangle}{\boxslash}$. Multiplication and division in "round toward negative infinity" mode may be expressed with $\overset{\blacktriangledown}{\boxtimes}$ (or $\overset{\blacktriangledown}{\boxdot}$) and $\overset{\blacktriangledown}{\boxslash}$.

16.8.8 Addition and Subtraction Operators

Addition and subtraction are expressed with $+$ and $-$ on all numeric quantities, including intervals, as well as vectors and matrices.

The handling of overflow depends on the type of the number produced. For integer results, overflow throws an $\mathrm{IntegerOverflowException}$. Rational computations do not overflow. For floating-point results, overflow produces $+\infty$ or $-\infty$ according to the rules of IEEE 754. For intervals, overflow produces an appropriate containing interval.

Underflow is relevant only to floating-point computations and is handled according to the rules of IEEE 754.

Wraparound addition and subtraction on fixed-size integers are expressed by $\dot{+}$ and $\dot{-}$. Saturating addition and subtraction on fixed-size integers are expressed by $\boxplus$ and $\boxminus$. These operations do not overflow.

Ordinary addition and subtraction of floating-point numbers always use the IEEE 754 "round to nearest" rounding mode. This rounding mode may be emphasized by using the operators $\oplus$ and $\ominus$. Addition and subtraction in "round toward zero" mode may be expressed with $\boxplus$ and $\boxminus$. Addition and subtraction in "round toward positive infinity" mode may be expressed with $\overset{\blacktriangle}{\boxplus}$ and $\overset{\blacktriangle}{\boxminus}$. Addition and subtraction in "round toward negative infinity" mode may be expressed with $\overset{\blacktriangledown}{\boxplus}$ and $\overset{\blacktriangledown}{\boxminus}$.

The construction $x \pm y$ produces the interval $\langle\!|x - y, x + y|\!\rangle$.

16.8.9 Intersection, Union, and Set Difference Operators

Sets support the operations of intersection $\cap$, union $\cup$, disjoint union $\uplus$, set difference $\setminus$, and symmetric set difference $\ominus$. Disjoint union throws $\mathrm{DisjointUnionException}$ if the arguments are in fact not disjoint.

Intervals support the operations of intersection $\cap$, union $\cup$, and interior hull $\Cup$. The operation $\Cap$ returns a pair of intervals; if the intersection of the arguments is a single contiguous span of real numbers, then the first result is an interval representing that span and the second result is an empty interval, but if the intersection is two spans, then two (disjoint) intervals are returned. The operation $\Cup$ returns a pair of intervals; if the arguments overlap, then the first result is the union of the two intervals and the second result is an empty interval, but if the arguments are disjoint, they are simply returned as is.

16.8.10 Minimum and Maximum Operators

The operator `MAX` returns the larger of its two operands, and `MIN` returns the smaller of its two operands.

For floating-point numbers, if either argument is a NaN then NaN is returned. The floating-point operations `MAXNUM` and `MINNUM` behave similarly except that if one argument is NaN and the other is a number, the number is returned. For all four of these operators, when applied to floating-point values, -0 is considered to be smaller than $+0$.

16.8.11 GCD, LCM, and CHOOSE Operators

The infix operator `GCD` computes the greatest common divisor of its integer operands, and `LCM` computes the least common multiple. The operator `CHOOSE` computes binomial coefficients: $n \texttt{ CHOOSE } k = \binom{n}{k} = \frac{n!}{k!(n-k)!}$.

16.8.12 Equivalence and Inequivalence Operators

The $\equiv$ operator denotes strict equivalence testing. If the two expressions tested denote object references, the equivalence test evaluates to $true$ if and only if the object references are identical; otherwise, it evaluates to $false$. If the two expressions tested denote value objects, the test evaluates to $true$ if and only if the value objects have the identical type, environment, and fields. Otherwise, it evaluates to $false$. If the two expressions tested denote function objects, the test throws a $\mathrm{FunctionEquivalenceTest}$ exception.

The expression $e \neq e'$ is semantically equivalent to the expression $\neg(e = e')$.

16.8.13 Comparisons Operators

Unless otherwise noted, the operators described in this section produce boolean ($true/false$) results.

The operators $<$, $\leq$, $\geq$, and $>$ are used for numerical comparisons and are supported by integer, rational, and floating-point types. Comparison of rational values throws $\mathrm{RationalComparisonException}$ if either argument is the rational infinity $1/0$ or the rational indefinite $0/0$. Comparison of floating-point values throws $\mathrm{FloatingComparisonException}$ if either argument is a NaN.

The operators $<$, $\leq$, $\geq$, and $>$ may also be used to compare characters (according to the numerical value of their Unicode codepoint values) and strings (lexicographic order based on the character ordering). They also use lexicographic order when used to compare lists whose elements support these same comparison operators.

When $<$, $\leq$, $\geq$, and $>$ are used to compare numerical intervals, the result is a boolean interval. The functions $possibly$ and $certainly$ are useful for converting boolean intervals to boolean values for testing. Thus $possibly(x > y)$

is true if and only if there is some value in the interval x that is greater than some value in the interval y, while $certainly(x > y)$ is true if and only if x and y are nonempty and every value in x is greater than every value in y.

The operators $\subset$, $\subseteq$, $\supseteq$, and $\supset$ may be used to compare sets or intervals regarded as sets.

The operator $\in$ may be used to test whether a value is a member of a set, list, array, interval, or range.

16.8.14 Logical Operators

The following binary operators may be used on boolean values:

$\wedge$	AND
$\vee$	inclusive OR
$\underline{\vee}$ or $\oplus$ or $\neq$	exclusive OR
$\equiv$ or $\leftrightarrow$ or $=$	equivalence (if and only if)
$\rightarrow$	IMPLIES
$\overline{\wedge}$	NAND
$\overline{\vee}$	NOR

These same operators may also be applied to boolean intervals to produce boolean interval results.

The following operators may be used on integers to perform "bitwise" operations:

$\mathbb{\wedge}$	bitwise AND
$\mathbb{\vee}$	bitwise inclusive OR
$\underline{\mathbb{\vee}}$	bitwise exclusive OR

The prefix operator $\dashv$ computes the bitwise NOT of an integer.

16.8.15 Conditional Operators

If $p(x)$ and $q(x)$ are expressions that produce boolean results, the expression $p(x) \wedge : q(x)$ computes the logical AND of those two results by first evaluating $p(x)$. If the result of $p(x)$ is $true$, then $q(x)$ is also evaluated, and its result becomes the result of the entire expression; but if the result of $p(x)$ is $false$, then $q(x)$ is not evaluated, and the result of the entire expression is $false$ without further ado. (Similarly, evaluating the expression $p(x) \vee : q(x)$ does not evaluate $q(x)$ if the result of $p(x)$ is $true$.) Contrast this with the expression $p(x) \wedge q(x)$ (with no colon), which evaluates both $p(x)$ and $q(x)$, in no specified order and possibly in parallel.

Chapter 17

Conversions and Coercions

Fortress provides a mechanism, called *coercion*, to allow a value of one type to be automatically converted to a value of another type. Programmers can define coercions (described in Section 17.2) and coercions may change the set of applicable declarations for a call as described in Section 17.4. If multiple coercions can be applied to a particular functional call then the most appropriate coercion is chosen statically. Restrictions on coercion declarations (described in Section 17.6) guarantee that the static resolution of coercion (described in Section 17.5) is well-defined. Fortress provides implicit coercions for tuple types and arrow types (described in Section 17.7). Fortress also provides an additional feature of coercion that allows "widest-need" evaluation of numerical expressions (described in Section 17.8).

17.1 Principles of Coercion

In certain situations it is convenient to be able to use a value of one type T as if it were a value of another type U even though T is not a subtype of U. For example, it is convenient to be able to use the integer-valued expression 2 in a floating-point expression even though its type is not a subtype of any floating-point type. Fortress supports the automatic conversion of integer values to floating-point values (the technical term for this kind of automatic conversion is coercion). In this way one can write $(x+1)/2$ rather than $(x+1.0)/2.0$, for example. Such coercion applies generally to function and method (collectively called as functional) calls as well as to operators; one can write $\ln 2$ rather than $\ln 2.0$, or $\arctan(1,1)$ rather than $\arctan(1.0,1.0)$, for example.

One way to think about coercion is that if type T can be coerced to type U, then a value of type T can be used to "stand in" for a value of type U in a functional call. This is different from actually *being* of type U; it means only that, given any value of type T, an appropriate value of type U can be computed to substitute for it. Also, coercion occurs only when the declared type of the corresponding parameter in the functional declaration is exactly the type U being coerced to (in whose declaration the coercion is defined) not if it is a supertype of U. Note, however, that if type T can be coerced to type U then any subtype of T can also be coerced to type U.

Coercion from T to U may also occur in variable definitions and assignment expressions when the declared type of the left-hand side is U and the type of the expression on the right-hand side is a subtype of T.

Coercion is *not* automatically chained in Fortress, unlike some other programming languages: even if type T can be coerced to type U and type U can be coerced to type V, it is not necessarily the case that type T can be coerced to type V. Type T can be coerced to type V only if trait V itself contains an appropriate coercion declaration to handle that particular type coercion.

Example 1: For any floating-point parameter, a decimal integer literal argument may be used.

Example 2: For any floating-point parameter, a floating-point argument of a shorter format may be used.

Example 3: For any floating-point interval parameter, a non-interval floating-point argument (of the same or shorter floating-point format) may be used.

17.2 Coercion Declarations

Syntax:

Coercion	::=	`coercion` [*StaticParams*] (*Id IsType*) *CoercionClauses* = *Expr*
CoercionClauses	::=	[*Throws*] [*CoercionWhere*] [*Contract*]
CoercionWhere	::=	`where` { *CoercionWhereClauseList* }
CoercionWhereClauseList	::=	*CoercionWhereClause* (, *CoercionWhereClause*)*
CoercionWhereClause	::=	*WhereClause*
	|	*TypeRef* `widens or coerces` *TypeRef*

To declare that trait U allows a coercion from type T, the declaration of trait U must provide a coercion declaration whose parameter type is T. Coercion declarations are like functional declarations, except that `coercion` is actually a special reserved word, a coercion declaration may have special `where`-clause constraints (described below), and it is not permitted to specify a return type, because the return type must be the very trait in whose declaration the coercion declaration appears. The coercion body is required; there is no such thing as an abstract coercion declaration.

Coercions may have static parameters (described in Chapter 11) and a `where` clause (described in Section 11.6), just like functionals. The `where` clause may contain one of the following special constraints:

Type_1 `coerces` Type_2
Type_1 `widens` Type_2
Type_1 `widens or coerces` Type_2

The first is true if trait Type_1 has a coercion from Type_2. The second is true if trait Type_1 has a widening coercion (described in Section 17.8) from Type_2. The third may be used only in a coercion with the "`widening`" special reserved word; it is true if Type_1 has a coercion from Type_2, and furthermore the coercion declaration to which the `where` clause belongs is widening only if Type_1 has a widening coercion from Type_2. For example:

```
trait Vector⟦T extends Number⟧
  widening coercion ⟦U extends Number⟧(x:Vector⟦U⟧)
    where {T widens or coerces U} = ...
end
```

Coercions, unlike methods, are not inherited. If trait V extends trait U, and trait U has a coercion from type T, then V does *not* thereby have a coercion from type T. (It may, however, have its own coercion from type T, separately defined within the body of V.) For example, given the following declarations,

```
trait A
  coercion (b:B) = ...
end
object B end
trait C extends A end
f(c:C) = 5
```

a call to $f(B)$ is considered a static error because trait C does not inherit a coercion from type B.

17.3 Coercion Invocations

One may invoke a coercion explicitly with the syntax: $\mathrm{Trait}.\ \texttt{coercion}\ (expr)$. Overloading resolution (described in Section 15.5) applies in the usual way if the specified trait has more than one coercion that applies to the actual argument.

One way to think about coercion is that explicit calls to coercion declarations are automatically inserted into the arguments of functional calls and the right-hand sides of assignment expressions and variable definitions. If trait U has a coercion from type T, then whenever a functional, f, is declared with a parameter of type U, a call $f(t)$ where t has type T can be rewritten to $f(U.\ \texttt{coercion}\ (t))$ making the declaration of f applicable to the call. However, this rewriting does not occur if there exists a declaration that is applicable to the call before the rewriting (see Section 17.5 for further discussion).

If coercion is possible for more than one element of a tuple argument, a cross-product effect is obtained. For example, in this code:

```
object X
  coercion (kiki : Y) = ...
  coercion (tutu : Q) = ...
  hack(other : X) = 1
end
object Y end
object Q end
foo(dodo : X) = 2
bar(hyar : X, yon : X) = 3
bar(hyar : Q, yon : Q) = 4
```

the following method invocations are valid:

```
X.hack(Y)
X.hack(Q)
```

Because there is no declaration of $hack$ applicable to Y and Q, these invocations are rewritten to:

```
X.hack(X. coercion (Y))
X.hack(X. coercion (Q))
```

Similarly, the function calls in the left-hand side of the following are valid and rewritten to the right-hand side:

$foo(Y)$	*is rewritten to*	$foo(X.\ \texttt{coercion}\ (Y))$
$foo(Q)$	*is rewritten to*	$foo(X.\ \texttt{coercion}\ (Q))$
$bar(Y, X)$	*is rewritten to*	$bar(X.\ \texttt{coercion}\ (Y), X)$
$bar(Q, X)$	*is rewritten to*	$bar(X.\ \texttt{coercion}\ (Q), X)$
$bar(X, Y)$	*is rewritten to*	$bar(X, X.\ \texttt{coercion}\ (Y))$
$bar(X, Q)$	*is rewritten to*	$bar(X, X.\ \texttt{coercion}\ (Q))$
$bar(Y, Y)$	*is rewritten to*	$bar(X.\ \texttt{coercion}\ (Y), X.\ \texttt{coercion}\ (Y))$
$bar(Q, Q)$	*is rewritten to*	$bar(Q, Q)$
$bar(Q, Y)$	*is rewritten to*	$bar(X.\ \texttt{coercion}\ (Q), X.\ \texttt{coercion}\ (Y))$
$bar(Y, Q)$	*is rewritten to*	$bar(X.\ \texttt{coercion}\ (Y), X.\ \texttt{coercion}\ (Q))$

Note that the call $bar(Q, Q)$ remains unchanged because there exists a declaration for bar that is applicable without coercion.

To continue the example, let us illustrate a point about type parameters. If we add the following definitions:

```
trait Frobboz
   frob(item : X) = 5
   coercion (m : Matrix) = ... (* irrelevant! *)
end
object Bozo⟦T extends Frobboz⟧(t : T) ... end
```

then this says nothing about whether the type parameter T of Bozo has coercions, because coercions are not inherited. (In fact, we can make a stronger statement: types named by type parameters *do not have coercions*!) But it is guaranteed that the following method invocations are valid if they occur within the declaration of Bozo:

$t.frob(X)$
$t.frob(Y)$
$t.frob(Q)$

because T inherits the method declaration $frob(item : X)$ from Frobboz and X contains coercions from Y and Q.

17.4 Applicability with Coercion

As discussed in the previous section, coercion in Fortress alters the set of applicable declarations for a particular functional call. In this section we formally define the applicability of declarations with coercion. (This level of formality is necessary to discuss the interaction of overloading and coercion.) We build on the terminology and notation defined in Chapter 15.

We write $T \to U$ if U defines a coercion from T. We say that T *can be coerced to* U, and we write $T \rightsquigarrow U$, if U defines a coercion from T or any supertype of T; that is, $T \rightsquigarrow U \iff \exists T' : T \preceq T' \wedge T' \to U$.

Because of automatic coercion, a declaration may be applicable even if its parameter type is not a supertype of the argument type of the call. We define a new relation, *substitutability*, that takes coercion into account.

We say that a type T is *substitutable* for type U, and we write $T \sqsubseteq U$, if T is a subtype of U or T can be coerced to U; that is,

$$T \sqsubseteq U \iff T \preceq U \vee T \rightsquigarrow U.$$

Note that $\sqsubseteq$ need not be transitive. This is a result of the fact that coercion does not chain. However, if T is substitutable for U then so are T's subtypes; that is, $A \preceq T \wedge T \sqsubseteq U \implies A \sqsubseteq U$.

A declaration $f(P)$ is *applicable with coercion* to a call $f(C)$ if the call is in the scope of the declaration and $C \sqsubseteq P$. (See Chapter 7 for the definition of scope.)

Recall that Section 15.2 describes a rewriting of named functional calls into dotted method calls when no declarations are applicable to the named functional call. This rewriting accounts for the applicability with coercion of dotted method declarations to named functional calls.

Similarly, a dotted method declaration $P_0.f(P)$ is *applicable with coercion* to a dotted method call $C_0.f(C)$ if it $C_0 \preceq P_0$ and $C \sqsubseteq P$. Notice that the self parameter and receiver are compared using the subtype relation instead of the substitutable relation. This reflects the restriction that `self` parameters of dotted methods cannot be coerced. However, `self` parameters of functional methods can be coerced. This distinction is consistent with the intuition that functional methods are rewritten into top-level functions.

Note that the only difference between applicability and applicability with coercion is that substitutability is used instead of subtyping to compare the parameter types of the declarations.

Also note that the definition of applicability with coercion does not take keyword parameters or varargs parameters into account. Such declarations are conceptually rewritten into numerous declarations which cannot be called with elided

arguments nor with variable number of arguments (as described in Section 15.4). If one of the expanded declarations is applicable with coercion to a call, then the original declaration (with keyword or varargs parameters) is applicable with coercion to the call.

17.5 Coercion Resolution

For a given functional call, we first determine whether there exists a declaration that is applicable without coercion. If so, the most specific declaration is selected; if not, then coercions are explicitly added to the functional call. If more than one coercion is possible then the coercion that yields the most specific type is chosen. Section 17.6 and Chapter 33 describe restrictions on the declarations of coercions and overloaded functionals that guarantee there exists always a most specific coercion when two or more are possible.

We first define a notion of more specific types in the presence of coercion. Recall from Section 8.1 that Fortress defines an *exclusion* relation between types which is denoted by $\lozenge$.

For types T and U, we say that T *rejects* U, and write $T \not\rightarrow U$, if for all coercions to T, the type coerced from excludes U:

$$T \not\rightarrow U \iff \forall A : A \rightarrow T \implies A \lozenge U.$$

Note that $T \not\rightarrow U$ implies $U \not\rightarrow T$.

We say that T is *no less specific* than U, and write $T \trianglelefteq U$, if T is a subtype of U or T excludes, can be coerced to, and rejects U:

$$T \trianglelefteq U \iff T \preceq U \vee (T \lozenge U \wedge T \rightsquigarrow U \wedge T \not\rightarrow U).$$

The $\trianglelefteq$ relation is reflexive and antisymmetric but not necessarily transitive. It is possible, for example, for three types, T, U and V, each excluding the other two, to have coercions from T to U and from U to V. In this case, $T \trianglelefteq U$ and $U \trianglelefteq V$ but $T \not\trianglelefteq V$.

Notice that if two tuple types have a bijective correspondence between their element types then the $\trianglelefteq$ relationship between the tuple types is equivalent to applying the $\trianglelefteq$ relation elementwise.

We say that T is *more specific* than U, and write $T \triangleleft U$, if $T \trianglelefteq U \wedge T \neq U$.

We extend the definitions of no less specific and more specific to declarations. We say that a declaration $f(P)$ is *no less specific* than a declaration $f(Q)$ if $P \trianglelefteq Q$. Similarly, we say that a declaration $P_0.f(P)$ is *no less specific* than a declaration $Q_0.f(Q)$ if $(P_0, P) \trianglelefteq (Q_0, Q)$. A declaration $f(P)$ is *more specific* than a declaration $f(Q)$ if $P \triangleleft Q$. Similarly, a declaration $P_0.f(P)$ is *more specific* than a declaration $Q_0.f(Q)$ if $(P_0, P) \triangleleft (Q_0, Q)$.

Now we describe how to determine which coercion is applied to a given call in terms of rewriting functional calls. The rewriting is for pedagogical purposes; implementation techniques may vary.

Consider a static call $f(A)$ or $A_0.f(A)$. Let Σ be the set of parameter types of functional declarations of f that are applicable to the call. Let Σ' be the set of parameter types of functional declarations of f that are applicable with coercion to the call. If Σ is not empty then we use the overloading resolution described in Section 15.5 to determine which element of Σ is called. If Σ is empty but Σ' is not, then we rewrite the call as follows. Define:

$$\mathcal{C} = \{T \in \Sigma' \mid S \rightarrow T \wedge A \preceq S\}.$$

Let $T \in \mathcal{C}$ be the most specific element of $\mathcal{C}$. In other words, there does not exist $T' \in \mathcal{C}$ such that $T' \triangleleft T$. The restrictions given in Section 17.6 and Chapter 33 guarantee that such a T exists and that it is unique. The call $f(A)$ or $A_0.f(A)$ is rewritten to $f(T.\texttt{coercion}\,(\mathrm{A}))$ or $A_0.f(T.\texttt{coercion}\,(A))$ and the declaration with parameter type T is applied to the call.

As an example, consider the following definitions:

```
trait ℤ32 end
trait ℤ64
  coercion (z:ℤ32) = ...
end
trait ℤ128
  coercion (z:ℤ32) = ...
  coercion (z:ℤ64) = ...
end
f(z:ℤ64) = 1
f(z:ℤ128) = 2
```

Then the call $f(\mathbb{Z}32)$ resolves statically to the declaration $f(z:\mathbb{Z}64)$ because $\mathbb{Z}64 \triangleleft \mathbb{Z}128$, which follows from the fact that $\mathbb{Z}64$ coerces to $\mathbb{Z}128$. The statically chosen declaration will be called at run time.

Notice that coercion is resolved statically. Once a coercion is statically chosen for a call, this coercion is conceptually inserted into the call. This means that the statically chosen coercion is applied at run time.

For example, in the following program:

```
trait A
  coercion (c:C) = ...
end
trait B excludes A end
trait C end
object D extends {B, C} end
f(a:A) = 3
f(b:B) = 4
c:C = D
f(c)
```

the call $f(c)$ resolves to the declaration $f(A)$ despite the fact that the declaration $f(B)$ is applicable to the dynamic call $f(D)$.

17.6 Restrictions on Coercion Declarations

We place two restrictions on coercion declarations to ensure that it is always possible to resolve coercions.

It is a static error for a type to define a coercion from any of its subtypes. For example, the following program is statically rejected:

```
trait Number
  coercion (z:ℤ) = ...
end
trait ℤ extends Number end
```

It is also a static error for cycles to exist in the type hierarchy produced by extension and coercion declarations. Such cycles may be composed of solely subtype relationships or solely coercion or a mixture of the two. In all cases the cycles are statically rejected. An example showing a cycle that is a mixture of subtype and coercion relationships follows:

```
trait A end
```

```
trait B extends A end
trait C extends B
  coercion (a : A) = ...
end
```

17.7 Coercions for Tuple and Arrow Types

Unlike other types, tuple types (described in Section 8.4) and arrow types (described in Section 8.5) are not defined explicitly. Therefore coercions to these types are also not defined explicitly. Instead, the following rules describe when these coercions are implicitly defined.

There is a coercion from a tuple type X to a tuple type Y if all the following conditions hold:

1. X is not a subtype of Y;
2. for every plain type T in Y, there is a corresponding plain type in X whose type is substitutable for T;
3. if neither X nor Y has a varargs type, then they have the same number of plain types;
4. if X has a varargs type, then Y has a varargs type, the type S of the varargs type $S...$ in X is substitutable for the type T of the varargs type $T...$ in Y, and X and Y have the same number of plain types;
5. if X has no varargs type and Y has a varargs type $T...$, then every plain type in X that has no corresponding plain type in Y is substitutable for T; and
6. the correspondence between keyword-type pairs in X and Y is bijective, and the type of each such pair in X is substitutable for the type in the corresponding pair in Y.

Tuple type coercions are invoked by distributing the coercion elementwise. However, if an element type of the tuple type being coerced from is a subtype of the corresponding element type of the tuple type being coerced to, the coercion is ignored. For example, the following coercions:

$(A, B).$ `coercion` (x, y)
$(kwd = A).$ `coercion` $(kwd = x)$

are rewritten to:

$(A.$ `coercion` $(x), B.$ `coercion` $(y))$
$(kwd = A.$ `coercion` $(x))$

However, if the type of y were a subtype of B then the first coercion would instead be rewritten to:

$(A.$ `coercion` $(x), y)$

Coercions to varargs types such as the following:

$(A\ldots).$ `coercion` (x, y)

are invoked by applying the coercion to the type in the varargs type, A in this example, to each of the elements of the tuple. Additionally, there is a coercion from any type T to a tuple type solely with varargs type $(T\ldots)$.

There is a coercion from arrow type "$A \to B$ `throws` C" to arrow type "$D \to E$ `throws` F" if all of the following conditions hold:

1. "$A \to B$ `throws` C" is not a subtype of "$D \to E$ `throws` F";
2. D is substitutable for A;

3. B is substitutable for E; and
4. for all X in C, there exists Y in F such that X is substitutable for Y.

Arrow type coercions are invoked by wrapping the argument function of the coercion in a function expression that applies the appropriate coercions to the argument and result of the function. For example, assuming that f is a function from type A to type B the following coercion:

$(C \to D).\ \texttt{coercion}\ (f)$

is rewritten to:

$\texttt{fn}\ x \Rightarrow D.\ \texttt{coercion}\ (f(A.\ \texttt{coercion}\ (x)))$

17.8 Automatic Widening

Syntax:

Coercion ::= `widening coercion` [*StaticParams*] (*Id IsType*) *CoercionClauses* = *Expr*

Fortress supports what is sometimes called "widest-need" evaluation of numerical expressions. To see the problem, suppose that a and b are 32-bit floating-point numbers and c is a 64-bit floating-point number. It is easy, and tempting, to write an expression such as

$c = c + a \cdot b$

but if you stop to think about it, if interpreted naively it will compute the product $a \cdot b$ as a 32-bit floating-point number, and the impression that the sum may be accurate to the precision of a 64-bit floating-point number is only an illusion. Widest-need processing takes context into account and "widens" (or "upgrades") the operands a and b to 64-bit floating-point numbers before the product is computed, so that the product will be computed as a 64-bit result.

Before widening is considered, a Fortress compiler, in its normal course of operation, analyzes an expression bottom-up. For every functional invocation, it needs to statically identify a specific declaration to be invoked. Once this is done, then for every functional invocation, one of two conditions holds: (a) The type of the argument expression is a subtype of the type of the parameter in the identified declaration. (b) The type of the argument expression can be coerced to the type of the parameter in the identified declaration.

This process has to be done bottom-up because in order to select a specific declaration for a functional invocation, it is necessary to know the types of the argument expressions, and if an argument expression is itself a functional invocation, it is necessary to select the specific declaration for that invocation in order to find out the return type of the invocation.

Once an expression has been fully analyzed in this way, then widening is performed as a top-down process. For each functional invocation that appears as an argument to a functional invocation, or on the right-hand side of a variable definition or an assignment expression, ask whether the argument expression (which, remember, is a functional invocation) falls under case (a) or case (b) above. If case (b), coercion of the functional result is required; if the relevant coercion declaration is a "widening" coercion, then this functional invocation is a candidate for widening. Call the type of the functional invocation T, and call the type of the corresponding functional parameter, or of the left-hand side of the variable definition or assignment expression, U.

When the functional call was considered during the bottom-up analysis, in general many overloaded declarations may have been considered; of all those that were applicable with coercion to the static call, the most specific one was chosen. When considering widening, we consider the same set of declarations, but first discard all declarations whose return types are not subtypes of U. Then if any remain, and there is a unique most specific one among them, then that declaration is attached to the functional call instead. In this way a coercion step is eliminated (coercion is no longer needed to convert the result of the functional invocation from type T to type U), possibly at the expense of requiring

a new coercion in a subexpression—but this is exactly the desired effect. Once this is done, then the subexpressions of the functional call are processed recursively.

This process is general enough not only to provide for widening floating-point precision, but to handle two other cases of interest as well: widening point computations to interval computations, and widening computations involving aggregate data structures whose components are widenable. For example, in $d(v \times w)$ suppose that d is of type $\mathbb{R}64$ and v and w are 3-vectors with components of type $\mathbb{R}32$; this strategy is capable of promoting v and w to 3-vectors with components of type $\mathbb{R}64$ and then performing all computations with $\mathbb{R}64$ precision.

Let's go through the example of $c + a \cdot b$ in detail. The relevant declarations might look something like this:

```
trait ℝ32
  ...
end

trait ℝ64
  widening coercion (x : ℝ32) = ...
  ...
end

opr · (l: ℝ32, r: ℝ32): ℝ32 = ... (* 1 *)
opr · (l: ℝ64, r: ℝ64): ℝ64 = ... (* 2 *)
opr +(l: ℝ32, r: ℝ32): ℝ32 = ... (* 3 *)
opr +(l: ℝ64, r: ℝ64): ℝ64 = ... (* 4 *)
```

The bottom-up analysis observes that a and b are both of type $\mathbb{R}32$, and discovers that declarations 1 and 2 are both applicable to the product, but declaration 1 would be chosen because it requires no coercion and declaration 2 would require coercion of both arguments to type $\mathbb{R}64$. The analysis then observes that c has type $\mathbb{R}64$ and the product expression has type $\mathbb{R}32$, so declaration 3 is not applicable but declaration 4 is applicable (though requiring a coercion from $\mathbb{R}32$ to $\mathbb{R}64$ for its second argument).

Now the top-down widening analysis is performed. The product is a functional call that is an argument to another functional call, and its result requires coercion, and that coercion is a widening coercion. Therefore the compiler reconsiders the applicable declarations, which were declarations 1 and 2. The result type of declaration 1 is not a subtype of $\mathbb{R}64$, but the result type of declaration 2 is a subtype of $\mathbb{R}64$ (in fact, it *is* $\mathbb{R}64$). Declaration 2 is the most specific among the applicable declarations with that property (in fact, in this example it's the only one), so declaration 2 replaces declaration 1 for the product invocation. This in turn requires a and b to be coerced from type $\mathbb{R}32$ to $\mathbb{R}64$ before the product is performed.

Widening is a tricky business, best left to expert library designers. When used judiciously, it can greatly improve the precision of numerical computations without requiring a great deal of thought on the part of the application programmer and without cluttering up code with explicit conversions.

Future versions of this specification will include tables of coercions and widening coercions supported by the Fortress standard libraries.

Chapter 18

Dimensions and Units

Syntax:

DimType	::=	*DimRef*
	|	*TypeRef DimRef* | *TypeRef* · *DimRef*
	|	*TypeRef* / *DimRef* | *TypeRef* per *DimRef*
	|	*TypeRef UnitRef* | *TypeRef* · *UnitRef*
	|	*TypeRef* / *UnitRef* | *TypeRef* per *UnitRef*
	|	*TypeRef* `in` *DimRef*
DimRef	::=	Unity
	|	*DottedId*
	|	*DimRef DimRef* | *DimRef* · *DimRef*
	|	*DimRef* / *DimRef* | *DimRef* per *DimRef*
	|	*DimRef* ^ *NatRef* | 1 / *DimRef* | (*DimRef*)
	|	*DUPreOp DimRef* | *DimRef DUPostOp*
DUPreOp	::=	square | cubic | inverse
DUPostOp	::=	squared | cubed
UnitExpr	::=	*UnitRef*
	|	*Expr UnitRef* | *Expr* · *UnitRef*
	|	*Expr* / *UnitRef* | *Expr* per *UnitRef*
	|	*Expr* `in` *UnitRef*
UnitRef	::=	dimensionless
	|	*DottedId*
	|	*UnitRef UnitRef* | *UnitRef* · *UnitRef*
	|	*UnitRef* / *UnitRef* | *UnitRef* per *UnitRef*
	|	*UnitRef* ^*NatRef* | 1/ *UnitRef* | (*UnitRef*)
	|	*DUPreOp UnitRef* | *UnitRef DUPostOp*

There are special type-like constructs called *dimensions* that are separate from other types (described in Chapter 8). There are also special constructs that modify types and values called *units* that are instances of dimensions. These are used to describe physical quantities.

The Fortress standard libraries define dimensions and units for the standard SI system of measurement based on meters, kilograms, seconds, amperes, and so on (as described in Section 29.1). The Fortress standard libraries also provide supplemental units of measurement, such as feet and miles (as described in Section 29.2). For example, the Fortress standard libraries provide a dimension named Length whose default unit is named meter and abbreviated `m_`. By rendering convention, this abbreviation is rendered in roman type without the underscore: m. In contrast, the variable `m` is rendered as in standard mathematical notation: m. See Section 5.17 for a discussion of formatting conventions

for tokens.

For readability, plural forms of the unit names are defined as equivalent to the corresponding singular forms; thus one can write meters per second, for example. Standard SI prefixes may be used on both the name and the symbol, so that nanometer and nm are also units of the dimension named Length, related to meter and m by a conversion factor of 10^{-9}.

Every dimension may have a default unit that is used for representing values of quantities of that dimension if no unit is specified explicitly. The Fortress standard libraries define these default dimensions and units:

Length	meter	MagneticFlux	weber
Mass	kilogram	MagneticFluxDensity	tesla
Time	second	Inductance	henry
Frequency	hertz	Velocity	meters per second
Force	newton	Acceleration	meters per second squared
Pressure	pascal	Angle	radian
Energy	joule	SolidAngle	steradian
Power	watt	LuminousIntensity	candela
Temperature	kelvin	LuminousFlux	lumen
Area	square meter	Illuminance	lux
Volume	cubic meter	RadionuclideActivity	becquerel
ElectricCurrent	ampere	AbsorbedDose	gray
ElectricCharge	coulomb	DoseEquivalent	*sievert*
ElectricPotential	volt	AmountOfSubstance	mole
Capacitance	farad	CatalyticActivity	katal
Resistance	ohm	MassDensity	kilograms per cubic meter
Conductance	siemens		

In addition, the Fortress standard libraries define the dimension Information with units bit and byte (and the plurals bits and bytes), a byte being equal to 8 bits. To avoid confusion, SI prefixes are *not* provided for these units; instead, programmers must use appropriate powers of 2 or 10, for example 10^6bits or 2^{20}bits.

Here are some examples of the use of dimensions and units:

$x\colon \mathbb{R}64\,\text{Length} = 1.3\,\text{m}$
$t\colon \mathbb{R}64\,\text{Time} = 5\,\text{s}$
$v\colon \mathbb{R}64\,\text{Velocity} = x/t$
$w\colon \mathbb{R}64\,\text{Velocity}\ \texttt{in}\ \text{nm/s} = 17\,\text{nm/s}$
$x\colon \mathbb{R}64\,\text{Velocity}\ \texttt{in}\ \text{furlongs per fortnight} = v\ \texttt{in}\ \text{furlongs per fortnight}$

Dimensions and units can be multiplied, divided, and raised to rational powers to produce new dimensions and units. Both the numerator and the denominator of a rational power of a dimension or a unit must be a valid `nat` *parameter* instantiation (as described in Section 11.2). Multiplication can be expressed using juxtaposition or $\cdot$; division can be expressed using $/$ or per. The syntactic operators square and cubic may be applied to a dimension or unit in order to raise it to the second power, third power, respectively; the special postfix syntactic operators squared and cubed may be used for the same purpose. The syntactic operator inverse may be applied to a dimension or unit to divide it into 1. All of these syntactic operators are merely syntactic sugar, expanded before type checking.

$\text{grams per cubic centimeter}$
$\text{meter per second squared}$
inverse ohms

One can also write $1/X$ as a synonym for X^{-1} if X is either a dimension or a unit.

Most numerical values in Fortress are dimensionless quantity values. Multiplying or dividing a dimensionless value by a unit produces a dimensioned value; thus $5\,\text{s}$ is the dimensioned value equal to five seconds, which has numerical

value 5 and second for its unit. A dimensioned value may also be multiplied or divided by a unit, and the result is to combine the units; the expression $(17\,\mathrm{nm})/\mathrm{s}$ first multiplies 17 by nm to produce the dimensioned value seventeen nanometers, which is then divided by the unit s to produce seventeen nanometers per second.

The unit of a dimensioned value may be changed to another unit of the same dimension by using the `in` operator, which takes a quantity to its left and a unit to its right. The `in` operator changes the unit and multiplies or divides the numerical value by an appropriate conversion constant so as to preserve the overall dimensioned value. Thus $1.3\,\mathrm{m}$ `in` nm produces $1300000000\,\mathrm{nm}$.

Multiplying or dividing a dimensionless numerical type by a unit produces an equivalent dimensioned numerical type with that unit associated; thus $\mathbb{R}64\,\mathrm{meter}$ is a type that is just like $\mathbb{R}64$ but whose values are values of dimension Length measured in meters. A dimensioned numerical type may be further multiplied or divided by a unit. As a convenience, if a dimension has a default unit, a numerical type may also be multiplied or divided by a dimension, in which case the result is as if the default unit for that dimension had been used. The `in` operator may also be used to change the unit associated with a dimensioned type; in this situation the effect is merely to alter the unit associated with the type; no numerical operation is performed at run time.

Certain aggregate types, such as Vector, may also have associated units.

There are two reasons for using dimensions and units. One is that the `in` operator can supply conversion factors automatically. The other is that certain programming errors may be detected at compile time. When dimensioned values are added, subtracted, or compared, it is a static error if the units do not match. When dimensioned values are multiplied or divided, their units are multiplied or divided. When taking the square root of a dimensioned value, the unit of the result is the square root of the argument's unit. Other numerical functions, such as $\sin$ and $\log$, require dimensionless arguments.

A variable whose declared type includes a dimension without an accompanying unit is understood to have the default unit for that dimension. Thus, in most cases, the runtime unit of an expression can be statically inferred. However, there are exceptions. For example, consider the following declaration:

$$a : \mathrm{Object}[3] = [5\,\mathrm{mg}, 3\,\mathrm{m}, 4\,\mathrm{s}]$$

Now suppose we declare a function that takes an array of objects and returns one of its elements:

$$f(xs : \mathrm{Object}[3]) = xs_1$$

The value of the call $f(a)$ is $3\,\mathrm{m}$. However, the static type of $f(a)$ is simply Object. When the value $3\,\mathrm{m}$ is placed in an array of objects, the value is boxed, and the unit associated with the value must be included as part of the boxed value. However, unboxed values need not include unit information at runtime, as this information is statically evident in such cases.

There are also special static parameters for units and dimensions; see Section 11.4.

Chapter 19

Tests and Properties

Fortress supports automated program testing. Components and APIs can declare *tests*. Test declarations specify explicit finite collections over which the test is run. Components, APIs, traits, and objects can declare *properties* which describe boolean conditions that the enclosing construct is expected to obey. Tests and properties may modify the program state.

19.1 The Purpose of Tests and Properties

To help make programs more robust, Fortress programs are allowed to include special constructs called *tests* and *properties*. Tests consist of test data along with code that can be run on that data. Properties are documentation used to describe the behavior of the traits and functions of a program; they can be thought of as comments written in a formal language. For each property, there is a special function that can be called by a program's tests to ensure that the property holds for specific test data.

A fortress includes hooks to allow programmers to run a specific test on an executable component, and to run all tests on such a component. A particularly useful time to run the tests of an executable component is at component link time; errors in the behavior of constituent components can be caught before the linked program is run.

19.2 Test Declarations

Syntax:

TestDecl	::=	`test` *Id* [*GeneratorList*] = *Expr*
GeneratorList	::=	*Generator* (, *Generator*)*
Generator	::=	*Id* ← *Expr*
	|	(*Id* , *IdList*) ← *Expr*
	|	*Expr*
IdList	::=	*Id* (, *Id*)*

A test declaration begins with the special reserved word `test` followed by an identifier, a list of zero or more generators (described in Section 13.17) enclosed in square brackets, the token =, and a subexpression. When a test is *run* (See Section 22.7), the subexpression is evaluated in each extension of the enclosing environment corresponding to each point in the cross product of bindings determined by the generators in the generator list.

For example, the following test:

`test` $fxLessThanFy[x \leftarrow E, y \leftarrow F] = assert(f(x) < f(y))$

checks that, for each value x supplied by generator E, and for each value y supplied by generator F, $f(x)$ is less than $f(y)$.

19.3 Other Test Constructs

Syntax:

UniversalMod ::= `test`

The `test` modifier can also appear on a function definition, trait definition, object definition, or top-level variable definition. In these contexts, the modifier indicates that the program construct it modifies can be referred to by a test. Functions with modifier `test` must not be overloaded with functions that do not have modifier `test`, and traits with modifier `test` must not be extended by traits or objects that do not have modifier `test`. The collection of all constructs in a program that include modifier `test` are referred to collectively as the program's *tests*. Tests can refer to non-tests but it is a static error for a non-test to refer to any test.

For example, we can write the following helper function:

```
test ensureApplicationFails(g, x) = do
  applicationSucceeded := false
  try
    g(x)
    applicationSucceeded := true
  catch e
    Exception ⇒ ()
  end
  if applicationSucceeded then
    fail "Application succeeded!"
  end
end
```

The library function *fail* displays the error message provided and terminates execution of the enclosing test.

19.4 Running Tests

When a program's tests are *run*, the following actions are taken:

1. All top-level definitions, including constructs beginning with modifier `test`, are initialized. A test declaration with name t declares a function named t that takes a tuple of parameters corresponding to the variables bound in the generator list of the test declaration. For each variable v in the generator list of t, if the type of generator supplied for v is $\mathrm{Generator}[\![\alpha]\!]$ then the parameter in the function corresponding to v has type α. The return type of the function is $()$.

   Each such function bound in this manner is referred to as a *test function*. Test functions can be called from the rest of the program's tests.

2. Each test t in a program is run in each extension of t's enclosing environment with a point in the cross product of bindings determined by the generators in the test's generator list.

19.5 Test Suites

In order to allow programmers to run strict subsets of all tests defined in a program, Fortress allows tests to be assembled into *test suites*.

The convenience object $\mathrm{TestSuite}$ is defined in the Fortress standard libraries. An instance of this object contains a set of test functions that can all be called by invoking the method run:

```
test object TestSuite(testFunctions = {})
  add(f: () → ()) = testFunctions.insert(f)
  run() =
    for t ← testFunctions do
      t()
    end
end
```

Note that all tests in a $\mathrm{TestSuite}$ are run in parallel.

19.6 Property Declarations

Syntax:

PropertyDecl	::=	`property` [*Id* `=`] [$\forall$ *ValParam*] *Expr*
ValParam	::=	*ParamId*
	|	`(`[*ValParams*]`)`
ParamId	::=	*Id*
	|	`_`
ValParams	::=	*PlainParam*(`,` *PlainParam*)*
	|	[*PlainParam*(`,` *PlainParam*)* `,`] *Id* `:` *TypeRef* `...`
	|	[*PlainParam*(`,` *PlainParam*)* `,`] [*Id* `:` *TypeRef* `...` `,`] *PlainParam* `=` *Expr* (`,` *PlainParam* `=` *Expr*)*
PlainParam	::=	*ParamId* [*IsType*]
	|	*TypeRef*

Components and APIs may include *property* declarations, documenting boolean conditions that a program is expected to obey. Syntactically, a property declaration begins with the special reserved word `property` followed by an optional identifier followed by the token `=`, an optional value parameter, which may be a tuple, preceded by the token $\forall$, and a boolean subexpression. In any execution of the program, the boolean subexpression is expected to evaluate to true at any time for any binding of the property declaration's parameter to any value of its declared type. When a property declaration includes an identifier, the property identifier is bound to a function whose parameter and body are that of the property, and whose return type is $\mathrm{Boolean}$. A function bound in this manner is referred to as a *property function*.

Properties can also be declared in trait or object declarations. Such properties are expected to hold for all instances of the trait or object and for all bindings of the property's parameter. If a property in a trait or object includes a name, the name is bound to a method whose parameter and body are that of the property, and whose return type is $\mathrm{Boolean}$. A method bound in this manner is referred to as a *property method*. A property method of a trait T can be called, via dotted method notation, on an instance of T.

Property functions and methods can be referred to in a program's tests. If the result of a call to a property function or method is not $true$, a $\mathrm{TestFailure}$ exception is thrown. For example, we can write:

```
property fIsMonotonic = ∀(x: ℤ, y: ℤ) (x < y) → (f(x) < f(y))
```

$\texttt{test}\ s : \mathrm{Set}[\![\mathbb{Z}]\!] = \{-2000, 0, 1, 7, 42, 59, 100, 1000, 5697\}$
$\texttt{test}\ \mathit{fIsMonotonicOverS}[x \leftarrow s, y \leftarrow s] = \mathit{fIsMonotonic}(x, y)$
$\texttt{test}\ \mathit{fIsMonotonicHairy}[x \leftarrow s, y \leftarrow s] = \mathit{fIsMonotonic}(x, x^2 + y)$

The test $\mathit{fIsMonotonicOverS}$ tests that function f is monotonic over all values in set s. The test $\mathit{fIsMonotonicHairy}$ tests that f is monotonic with respect to the values in s compared to the set of values corresponding to all the ways in which we can choose an element of s, square it, and add it to another element of s.

Chapter 20

Type Inference

Type inference in Fortress is performed independently on each simple component (described in Chapter 22). For separation of concerns, in this chapter, we describe the Fortress type inference as a procedure performed over a whole Fortress program. We explain how this procedure can be adapted to perform type inference over a simple program component in Section 22.5. Note that type inference cannot be performed on each functional (function or method) declaration in isolation because it may be declared mutually recursively or may contain free variables.

20.1 What Is Inferred

Types of functional parameters, functional results, and variables may be elided in a program where they can be inferred automatically. Similarly, instantiations of static parameters of generic functional invocations may be elided where they can be inferred automatically. A Fortress compiler must allow types and static arguments inferable via the procedure described in Section 20.2 to be elided, no more and no less. This strict requirement is made for the sake of source-code portability; it is important that a program that compiles on one compiler will compile on all compilers. Of course, there is nothing preventing a development environment from aiding programmers by performing more sophisticated analyses and filling in additional information, but the text produced is not a valid Fortress program unless all elided types and static arguments can be inferred via the described procedure.

20.2 Type Inference Procedure

To perform type inference, we first infer any elided parameter type in each functional declaration that can be inferred from other declaration as follows:

1. If the declaration is a functional declaration and the type of the declared functional is declared via a separate declaration, any elided parameter type is inferred to have the type provided by the separate declaration.
2. Otherwise, if the declaration is a method declaration that overloads a method declaration provided by a supertrait, any elided parameter type is inferred to have the type provided (or inferred) by the overloaded method declaration.

All remaining parameter types are inferred along with instantiations of static parameters.

In the following, we adopt the convention of writing "primed" static variables, $T'_0, ..., T'_m, U'_0, ...$, to stand for fresh static variables. Our procedure will introduce primed static variables as placeholders that are to be replaced with

nonprimed types and static arguments before the termination of type inference. We abuse notation by using *types* to refer to both types and static arguments when it is clear from context.

First, we annotate every expression e that is not a functional application with a fresh static variable (written here as a superscript):

$e^{T'}$

and every functional application $f(a_0, ..., a_n)$ (where f is the name of a generic functional) with fresh static variables for the instantiation of each of the functional's static parameters as well as a superscript:

$f[\![T'_0, ..., T'_{m-1}]\!](a_0, ..., a_n)^{T'_m}$

and every functional parameter x without a declared type with fresh static variable as its declared type:

$x : T'$

We say that a functional application is an *outermost* functional application if and only if it is not a proper subexpression of another functional application.

For each outermost functional application:

$f[\![T'_0, ..., T'_{m-1}]\!](a_0, ..., a_n)^{T'_m}$

let $f[\![R_0, ..., R_{m-1}]\!](p_0 : S_0, ..., p_n : S_n) : R_m$ be a declaration of f. Some of the types and static parameters appearing at the declaration might be primed static variables themselves.

Note that there may be several declarations of f due to overloading. When there are multiple declarations for f, type inference is performed to each declaration. Only the declarations to which type inference succeed are considered for overload resolution.

We accumulate a table of subtyping constraints as follows:

First, we add the following constraint to our table:

$T'_m <: [R_0 \mapsto T'_0, ..., R_{m-1} \mapsto T'_{m-1}]R_m$

where we use the notation $[R_0 \mapsto T'_0, ..., R_{m-1} \mapsto T'_{m-1}]$ to denote the safe substitution of $R_0, ..., R_{m-1}$ with $T'_0, ..., T'_{m-1}$.

If the functional application occurs as the right-hand side of a declaration $x : T$ `=` e, we add the constraint:

$T'_m <: T$

An analogous constraint is added if the application occurs as the right-hand-side of an assignment to a variable with type T, or as the body of a functional with return type T, or as an expression ascripted with type T, etc. We refer to all of these cases collectively by saying that the functional application "occurs in the context of type T".

Additionally, for every argument $a_i^{U'_i}$ corresponding to value parameter p_i, we add the following constraint:

$U'_i <: [R_0 \mapsto T'_0, ..., R_{m-1} \mapsto T'_{m-1}]S_i$

Additionally, we add all "nested constraints" accumulated for a_i, where nested constraints are accumulated as follows:

1. If a_i is not a functional application or a `typecase` expression, the nested constraints are the union of the nested constraints of all subexpressions of a_i plus the constraint $R <: U'_i$, where R is the type of a_i in the enclosing context.
2. If a_i is a `typecase` expression, only nested constraints common to all branches are accumulated. Furthermore, static parameter instantiations of generic functional calls within each branch must be inferred independently, in a separate table extending the constraints of the enclosing context with only those constraints available in that branch.

3. If a_i is a functional application, the nested constraints are all constraints accumulated for a_i as if it were an outermost generic functional application occurring in context U_i'.

Once we have accumulated all constraints for each outermost functional application, the constraints are solved in the following phases:

1. For every cycle of constraints of naked primed static variables $T_1' <: ... <: T_n' <: T_1'$, one of $T_1', ..., T_n'$ is chosen. We call the chosen naked primed static variable T^*. All occurrences of $T_1', ..., T_n'$ in the constraints and throughout the program are replaced with T^*. Redundant constraints are eliminated. This process is repeated until a fixed point is reached.

2. For every naked primed static variable T', consider the set of all subtype constraints on T': $T' <: U_1, ...T' <: U_n$. The *unexpanded* inferred least upper bound for T' is the intersection type (described in Section 8.8) $U_1 \cap ... \cap U_n$.

3. If some primed static variable R' appears in one of the U_i in the unexpanded inferred least upper bound for T' and R' does not appear within a type constructor, R' is replaced with its own unexpanded inferred type. This process is repeated until a fixed point is reached.

   If T' is expanded to a type with no primed variables, this phase is done. If T' is expanded to a type containing no primed variables except T', the expansion E of T' is replaced with a ***fixed-point type*** $\mu T'.E$. Fixed-point types are needed to express types that would be "infinitely" large if expanded out into conventional ground types. We have the following property for fixed-point types: $\mu X.E = [X \mapsto \mu X.E]E$. Otherwise, T' is inferred to have type Object.

4. For every naked primed static variable T', consider the set of all supertype constraints on T': $V_1 <: T', ...V_n <: T'$. The unexpanded inferred greatest lower bound for T' is the union type (described in Section 8.8) $V_1 \cup ... \cup V_n$. This type is expanded analogously to the procedure used for the intersection of T'.

5. The inferred type for a static variable T' is the intersection of the expanded inferred least upper bound and the expanded inferred greatest lower bound for T'. For the purposes of type checking, the inferred type is put into clausal normal form as a canonical form, eliminating redundant clauses.

20.3 Finding "Closest Expressible Types" for Inferred Types

Once solutions to the primed variables are inferred, we must find the *closest expressible type* to each inferred type. Expressible types are all types that are neither intersection types nor union types and that syntactically contain only expressible types. Note that fixed-point types are expressible types as in the Java type inference.

1. Given an intersection type $S_1 \cap ... \cap S_n$, if there is a unique most general expressible type T such that T is a subtype of $S_1, ..., S_n$, then T is the closest expressible type to $S_1 \cap ... \cap S_n$. If there is not a unique most general expressible type, then the close expressible type is the closest expressible supertype of the multiple closest expressible types; this process is guaranteed to terminate because the type hierarchy is rooted at type Object.

2. Given a union type $S_1 \cup ... \cup S_n$, if there is a unique most specific expressible type T such that T is a supertype of $S_1, ..., S_n$, then T is the closest expressible type to $S_1 \cup ... \cup S_n$. If there is not a unique most general expressible type, then the close expressible type is the closest expressible supertype of the multiple closest expressible types.

3. Given a generic type instantiation $C[\![S_1, ..., S_n]\!]$, where some of the $S_1, ..., S_n$ are not expressible, the closest expressible type is $C[\![E(S_1), ..., E(S_n)]\!]$ where $E(S_i)$ is the closest expressible type to S_i.

Each primed type in a program is replaced with the closest expressible type to its solution. Any remaining primed types that have not yet been inferred are then inferred to have type Object.

Chapter 21

Memory Model

Fortress programs are highly multithreaded by design; the language makes it easy to expose parallelism. However, many Fortress objects are mutable; without judicious use of synchronization constructs—reductions and `atomic` expressions—*data races* will occur and programs will behave in an unpredictable way. The memory model has two important functions:

1. Define a programming discipline for the use of data and synchronization, and describe the behavior of programs that obey this discipline. This is the purpose of Section 21.2.

2. Define the behavior of programs that do not obey the programming discipline. This constrains the optimizations that can be performed on Fortress programs. The remaining sections of this chapter specify the detailed memory model that Fortress programs must obey.

21.1 Principles

The Fortress memory model has been written with several important guiding principles in mind. Violations of these principles should be taken as a flaw in the memory model specification rather than an opportunity to be exploited by the programmer or implementor. The most important principle is this: violations of the Fortress memory model must still respect the underlying data abstractions of the Fortress programming language. All data structures must be properly initialized before they can be read by another thread, and a program must not read values that were never written. When a program fails, it must fail gracefully by throwing an exception.

The second goal is nearly as important, and much more difficult: present a memory model which can be understood thoroughly by programmers and implementors. It should never be difficult to judge whether a particular program behavior is permitted by the model. Where possible, it should be possible to check that a program obeys the programming discipline.

The final goal of the Fortress memory model is to permit aggressive optimization of Fortress programs. A multiprocessor memory model can rule out optimizations that might be performed by a compiler for a uniprocessor. The Fortress memory model attempts to rule out as few behaviors as possible, but more importantly attempts to make it easy to judge whether a particular optimization is permitted or not. The semantics of Fortress already allows permissive operation reordering in many programs, simply by virtue of the implicitly parallel semantics of tuple evaluation and looping.

21.2 Programming Discipline

If Fortress programmers obey the following discipline, they can expect sequentially consistent behavior from their Fortress programs:

- Updates to shared locations should always be performed using an `atomic` expression. A location is considered to be shared if and only if that location can be accessed by more than one thread at a time. For example, statically partitioning an array among many threads need not make the array elements shared; only elements actually accessed by more than one thread are considered to be shared.
- Within a thread or group of implicit threads objects should not be accessed through aliased references; this can yield unexpected results. Section 21.2.3 defines the notion of *apparently disjoint* references. An object should not be written through one reference when it is accessed through another apparently disjoint reference.

The following stylistic guidelines reduce the possibility of pathological behavior when a program strays from the above discipline:

- Where feasible, reduction should be used in favor of updating a single shared object.
- Immutable fields and variables should be used wherever practical. We discuss this further in Section 21.2.1.
- A getter or a setter should behave as though it is performing a simple field access, even if it internally accesses many locations. The simplest (but not necessarily most efficient) way to obtain the appropriate behavior is to make hand-coded accessors `atomic`. Section 21.2.2 expands on this.

21.2.1 Immutability

Recall from Section 4.3 that we can distinguish mutable and immutable memory locations. Any thread that reads an immutable field will obtain the initial value written when the object was constructed. In this regard it is worth re-emphasizing the distinction between an object reference and the object it refers to. A location that does not contain a value object contains an object reference. If the field is immutable, that means the reference is not subject to change; however, the object referred to may still be modified in accordance with the memory model.

By contrast, recall that all the fields of a value object are immutable; however, a mutable location may have a value type. Such a location can be written, completely replacing the value object it contains. Similarly, reading the value contained in such a location conceptually causes the entire value object to be read; if this isn't followed by an immediate field reference, the read must be performed atomically. This may potentially be expensive (see Section 21.3).

21.2.2 Providing the Appearance of a Single Object

In practice, most accesses to fields occur through a mediating getter or setter method. It should not be possible for the programmer to tell whether a given getter or setter directly accesses a field or if it performs additional computations. Similarly, any subscripting operation must be indistinguishable from accessing a single field. Thus, accessor methods and subscripting methods should provide the appearance of atomicity, as described in Section 21.3. The Fortress standard libraries are written to preserve this abstraction. For example, the Array type in Fortress makes use of one or more underlying $\mathrm{HeapSequence}$s, and array subscripting preserves the atomicity guarantees of this underlying object.

21.2.3 Modifying Aliased Objects

In common with Fortran, and unlike most other popular programming languages, Fortress gives special treatment to accesses to a location through different aliases. For the purposes of variable aliasing, it is important to define the notion

of *apparently disjoint* (or simply disjoint) object and field references. If two references are not disjoint, we say they are *certainly the same*, or just *the same*. By contrast, we say object references are *identical* if they refer to the same object, and *distinct* otherwise. Accesses to fields reached via apparently disjoint object references may be reordered (except an initializing write is never reordered with respect to other accesses to the identical location).

Distinct references are always disjoint. Two identical references are apparently disjoint if they are obtained from any of the following locations:

- distinct parameters of a single function call
- distinct fields
- a parameter and a field
- identically named fields read from apparently disjoint object references
- distinct reads of a single location for which there may be an interposing write

When comparing variables defined in different scopes, these rules will eventually lead to reads of fields or to reads of parameters in some common containing scope.

We extend this to field references as follows: two field references are apparently disjoint if they refer to distinct fields, or they refer to identically named fields read from apparently disjoint object references.

Consider the following example:

$f(x : \mathbb{Z}64[2], y : \mathbb{Z}64[2]) : \mathbb{Z}64 =$ `do`
  $x_0 := 17$
  $y_0 := 32$
`end`

Here x and y in f are apparently disjoint; the writes may be reordered, so the call $f(a, a)$ may assign either 17 or 32 to a_0.

A similar phenomenon occurs in the following example:

$g(x : \mathbb{Z}64[2], y : \mathbb{Z}64[2]) : \mathbb{Z}64 =$ `do`
  $x_0 := 17$
  y_0
`end`

Again x and y are apparently distinct in g, so the write to x_0 and the read of y_0 may be reordered. The call $g(a, a)$ will assign 17 to a_0 but may return either the former value of a_0 or 17.

It is safe to *read* an object through apparently disjoint references:

$h(x : \mathbb{Z}64[2], y : \mathbb{Z}64[2]) : \mathbb{Z}64 =$ `do`
  $u : \mathbb{Z}64 = x_0$
  $v : \mathbb{Z}64 = y_0$
  $u + v$
`end`

A call to $h(a, a)$ will read a_0 twice without ambiguity. Note, however, that the reads may still be reordered, and if a_0 is written in parallel by another thread this reordering can be observed.

If necessary, `atomic` expressions can be used to order disjoint field references:

$f'(x : \mathbb{Z}64[2], y : \mathbb{Z}64[2]) : () =$ `do`
  `atomic` $x_0 := 17$
  `atomic` $y_0 := 32$
`end`

Here the call $f(a, a)$ ends up setting a_0 to 32. Note that simply using a single `atomic` expression containing one or both writes is not sufficient; the two writes must be in distinct `atomic` expressions to be required to occur in order.

When references occur in distinct calling contexts, they are disambiguated at the point of call:

```
j(x : ℤ64[2], y : ℤ64) : () = x_0 := y
k(x : ℤ64[2]) : () = do
  j(x, 17)
  j(x, 32)
end
l(x : ℤ64[2], y : ℤ64[2]) : () = do
  j(x, 17)
  j(y, 32)
end
```

Here if we call $k(a)$ the order of the writes performed by the two calls to j is unambiguous, and a_0 is 32 in the end. By contrast, $l(a, a)$ calls j with two apparently disjoint references, and the writes in these two calls may thus be reordered.

21.3 Read and Write Atomicity

Any read or write to a location is *indivisible*. In practical terms, this means that each read operation will see exactly the data written by a single write operation. Note in particular that indivisibility holds for a mutable location containing a large value object. It is convenient to imagine that every access to a mutable location is surrounded by an `atomic` expression. However, there are a number of ordering guarantees provided by `atomic` accesses that are not respected by non-`atomic` accesses.

21.4 Ordering Dependencies among Operations

The Fortress memory model is specified in terms of two orderings: dynamic program order and memory order. *Dynamic program order* is a partial order between the expressions evaluated in a particular execution of a program. The actual order of memory operations in a given program execution is *memory order*, a total order on all memory operations. Dynamic program order is used to constrain memory order. However, memory operations need not be ordered according to dynamic program order; many memory operations, even reads and writes to a single field or array element, can be reordered. Programmers who adhere to the model in Section 21.2 can expect sequentially consistent behavior: there will be a global ordering for all memory operations that respects dynamic program order.

Here is a summary of the salient aspects of memory order:

- There is a single memory order which is respected in all threads.
- Every read obtains the value of the immediately preceding write to the identical location in memory order.
- Memory order on `atomic` expressions respects dynamic program order.
- Memory order respects dynamic program order for operations that certainly access the same location.
- Initializing writes are ordered before any other memory access to the same location.

21.4.1 Dynamic Program Order

Much of the definition of *dynamic program order* is given in the descriptions of individual expressions in Chapter 13. It is important to understand that dynamic program order represents a conceptual, naive view of the order of operations in an execution; this naive view is used to define the more permissive memory order permitted by the memory model. Dynamic program order is a partial order, rather than a total order; in most cases operations in different threads will not be ordered with respect to one another. There is an important exception: there is an ordering dependency among threads when a thread starts or must be complete.

An expression is ordered in dynamic program order after any expression it dynamically contains, with one exception: a `spawn` expression is dynamically ordered before any subexpression of its body. The body of the `spawn` is dynamically ordered before any point at which the spawned thread object is observed to have completed.

Only expressions whose evaluation completes normally occur in dynamic program order, unless the expression is "directly responsible" for generating abrupt termination. Examples of the latter case are `throw` and `exit` expressions and division by zero. In particular, when the evaluation of a subexpression of an expression completes abruptly, causing the expression itself to complete abruptly, the containing expression does not occur in dynamic program order. A `label` block is ordered after an `exit` that targets it. The expressions in a `catch` clause whose `try` block throws a matching exception are ordered after the `throw` and before any expression in the `finally` clause. If the `catch` completes normally, the `try` block as a whole is ordered after the expressions in the `finally` clause. For this reason, when we refer to the place of non-`spawn` expression in dynamic program order, we mean the expression or any expression it dynamically contains.

For any construct giving rise to implicit threads—tuple evaluation, function or method call, or the body of an expression with generators such as `for`—there is no ordering in dynamic program order between the expression executed in each thread in the group. These subexpressions are ordered with respect to expressions which precede or succeed the group.

When a function or method is called, the body of the function or method occurs dynamically after the arguments and function or receiver; the call expression is ordered after the body of the called function or method.

For conditional expressions such as `if`, `case`, and `typecase`, the expression being tested is ordered dynamically before any chosen branch. This branch is in turn ordered dynamically before the conditional expression itself.

Iterations of the body of a `while` loop are ordered by dynamic program order. Each evaluation of the guarding predicate is ordered after any previous iteration and before any succeeding iteration. The `while` loop as a whole is ordered after the final evaluation of the guarding predicate, which yields $false$.

An iteration of the body of a `for` loop, and each evaluation of the body expression in a comprehension or big operator, is ordered after the generator expressions.

21.4.2 Memory Order

Memory order gives a total order on all memory accesses in a program execution. A read obtains the value of the most recent prior write to the identical location in memory order. In this section we describe the constraints on memory order, guided by dynamic program order. We can think of these constraints as specifying a partial order which must be respected by memory order. The simplest constraint is that accesses certainly to the same location must respect dynamic program order. Apparently disjoint accesses need not respect dynamic program order, but an initializing write must be ordered before all other accesses to the identical location in program order.

Accesses in distinct (non-nested) `atomic` expressions respect dynamic program order. Given an `atomic` expression, we divide accesses into four classes:

1. Components, dynamically contained within the `atomic` expression.

2. Ancestors, dynamically ordered before the `atomic` expression.
3. Descendants, dynamically ordered before the `atomic` expression.
4. Peers, dynamically unordered with respect to operations dynamically contained within the `atomic` expression.

We say an `atomic` expression is *effective* if it contains an access to a location, there is a peer access to the identical location, and at least one of these accesses is a write. For an effective `atomic` expression, every peer access must either be a *predecessor* or a *successor*. A predecessor must occur before every component and every descendant in memory order. A successor must occur after every component and every ancestor in memory order. Every ancestor must occur before every descendant in memory order.

The above conditions guarantee that there is a single, global ordering for the effective `atomic` expressions in a Fortress program. This means that for any pair of `atomic` expressions A and B one of the following conditions holds:

- A is dynamically contained inside B.
- B is dynamically contained inside A.
- Every expression dynamically contained in A precedes every expression dynamically contained in B in memory order. This will always hold when A is dynamically ordered before B.
- Every expression dynamically contained in B precedes every expression dynamically contained in A in memory order. This will always hold when B is dynamically ordered before A.

The above rules are also sufficient to guarantee that `atomic` expressions nested inside an enclosing `atomic` behave with respect to one another just as if they had occurred at the top level in an un-nested context.

Any access preceding a `spawn` in dynamic program order will precede accesses in the spawned expression in memory order. Any access occurring after a spawned thread has been observed to complete in dynamic program order will occur after accesses in the spawned expression in memory order.

A reduction variable in a `for` loop does not have a single associated location; instead, there is a distinct location for each loop iteration, initialized by writing the identity of the reduction. These locations are distinct from the location associated with the reduction variable in the surrounding scope. In memory order there is a read of each of these locations each of which succeeds the last access to that variable in the loop iteration, along with a read of the location in the enclosing scope which succeeds all accesses to that location preceding the loop in dynamic program order. These reads are followed by a write of the location in the enclosing scope which in turn precedes all accesses to that location that succeed the loop in dynamic program order.

Finally, reads and writes in Fortress programs must respect dynamic program order for operations that are *semantically related*. If the read A precedes the write B in dynamic program order, and the value of B can be determined in some fashion without recourse to A, then these operations are not semantically related. A simple example is if A is a reference to variable x and B is the assignment $y := x \cdot 0$. Here it can be determined that $y := 0$ without recourse to x and these variables are not semantically related. By contrast, the write $y := x$ is always semantically related to the read of x. Note that two operations can only be semantically related if a transitive data or control dependency exists between them.

Chapter 22

Components and APIs

Fortress programs are developed, compiled, and deployed as *encapsulated upgradable components* that exist not only as programming language features, but also as self-contained run-time entities that are managed throughout the life of the software. The imported and exported references of a component are described with explicit *APIs*. With components and APIs, Fortress provides the stability benefits of static linking with the sharing and upgrading benefits of dynamic linking. [1] In addition to an informal description of the component system in this chapter, we also formally specify key functionality of the system, and illustrate how we can reason about the correctness of the system in Appendix C.

22.1 Overview

Components are the fundamental structure of Fortress programs. They export and import APIs, which serve as "interfaces" of the components. Components do not refer directly to other components. Rather, all external references are to APIs imported by the component. These references are resolved by linking components together: the references of a component to an imported API are resolved to a component that exports that API. Linking components produces new components, whose *constituents* are the components that were linked together.

Components are similar to modules in other programming languages, such as those of ML and Scheme [18, 14, 13]. But, unlike modules in those languages, components are designed for use during both development and deployment of software. In addition to compilation and linking, components can be produced by upgrading one component using another component that exports some of the APIs exported by the first component.

A key aspect of Fortress components is that they are encapsulated, so that upgrading one component does not affect any other component, even those produced by linking with the component that was upgraded. Abstractly, each component has its own copy of its constituents. However, implementations are expected to share common constituents when possible.

Users do not manipulate components directly. Instead, every component is installed in a persistent database on the system. We think of this database, which we call a *fortress*, as the agent that actually performs operations such as compilation, linking, upgrading, and execution of components: a virtual machine, a compiler, and a library registry all rolled into one. A fortress also maintains a list of APIs that are installed on it. A fortress also provides a shell by which the user can issue commands to it. Components and APIs are immutable objects. A fortress maps names to components installed on the system. The fortress operations are modeled as methods of the fortress that change the mapping.

[1] The system described in this chapter is based on that described in [2].

The ways in which fortresses are actually realized on particular platforms are beyond the scope of this specification. An implementor might choose to instantiate a fortress as a process, or as a persistent object database stored in a file system, with fortress operations being implemented as scripts that manipulate this database.

We call the source code for a single software component a *project*. Typically, when a project written in other programming languages is compiled, each file in the project is separately compiled. To ship an application, these files are linked together to form an application or library. Fortress uses a different model: a project is compiled directly into a single component, which is installed in the compiling fortress.

From the point of view of the compiler, all the source code for a project is contained in a single file. This approach simplifies the design, and gives a well-defined order for initialization of static elements of the component. However, this approach is unworkable for components of substantial size. Therefore, the compiler can be instructed to concatenate several source files together before compiling, while maintaining the original source location information.

After these components are compiled from source files, they can then be linked together to form larger components.

22.2 Components

Syntax:

Component	::=	`component` *DottedId Import** *Export** *Decl** `end`
DottedId	::=	*Id* (. *Id*)*
Import	::=	`import` *ImportFrom* `from` *DottedId*
	|	`import` *AliasedDottedIds*
ImportFrom	::=	`*` [`except` *Names*]
	|	*AliasedNames*
Names	::=	*Name*
	|	{ *NameList* }
Name	::=	*Id*
	|	`opr` *Op*
NameList	::=	*Name* (, *Name*)*
AliasedNames	::=	*AliasedName*
	|	{ *AliasedNameList* }
AliasedName	::=	*Id* [`as` *DottedId*]
	|	`opr` *Op* [`as` *Op*]
	|	`opr` *LeftEncloser RightEncloser* [`as` *LeftEncloser RightEncloser*]
AliasedNameList	::=	*AliasedName* (, *AliasedName*)*
AliasedDottedIds	::=	*AliasedDottedId*
	|	{ *AliasedDottedIdList* }
AliasedDottedId	::=	*DottedId* [`as` *DottedId*]
AliasedDottedIdList	::=	*AliasedDottedId* (, *AliasedDottedId*)*
Export	::=	`export` *DottedIds*
DottedIds	::=	*DottedId*
	|	{ *DottedIdList* }
DottedIdList	::=	*DottedId* (, *DottedId*)*

In this specification, we will refer to components created by compiling a file as *simple components*, while components created by linking components together will be known as *compound components*.

The source code of a simple component definition begins with the special reserved word `component` followed by a *possibly qualified name* (an identifier or a sequence of identifiers separated by periods with no intervening whitespace), followed by a sequence of *import* statements, and a sequence of *export* statements, and finally a sequence of declarations.

An import statement either imports an API and allows the specified names (separated by commas) declared in the API to be referred to with their unqualified names:

`import` $\{name^+\}$ `from` $apiName$

or imports an API as another name:

`import` $apiName$ `as` $anotherAPIName$

For convenience, an import statement can import an API and allow all elements declared in that API to be referred to with unqualified names:

`import * from` $apiName$

or can import an API and allow all elements except the specified names (separated by commas) declared in that API to be referred to with unqualified names:

`import * except` $\{name^+\}$ `from` $apiName$

If multiple elements with conflicting names are imported from separate APIs, all references to those elements within the component definition must be fully qualified. An export statement specifies the APIs that the component exports.

Every component implicitly imports the Fortress core APIs; every fortress has at least one component implementing all of these APIs. A *preferred* component exporting these APIs (configurable by the user) is implicitly linked to every component installed in the fortress.

An API (described in Section 22.3) serves as an interface of a component. For every API A exported by a component C, C must provide a definition for every program construct declared in A. These definitions must match the declarations in A exactly; the modifiers on constructs, the types of variables, the headers of functions and methods, and the headers of traits must be identical. There is one exception: A trait declaration with an empty `comprises` clause in A can be implemented by an object declaration in C. However, it is permissible for a trait or object definition to include additional methods and fields that are not declared in A. Also, a component is allowed to include top-level definitions that do not correspond to declarations in any of its exported APIs. The additional definitions that are not declared in A are not visible from outside the component.

When a component is compiled, the APIs it refers to must be present in the fortress. The import statements in a component are not a way to abbreviate unqualified names of objects or functions. In our system, an import statement merely allows references to the imported API to appear in the component definition. References to elements of an imported API must be fully qualified unless they are imported by an import statement with a `from` clause. When a component imports a functional f (either a top-level function or a functional method) by an import statement with a `from` clause, the imported f may be overloaded with a functional f declared by the component. When a component imports a top-level declaration f from an API A, all the relevant types to type check the uses of f are implicitly imported from A. However, these implicitly imported types for type checking are not expressible by programmers; programmers must import the types explicitly by import statements to use them.

A key design choice we make is to require that components never refer to other components directly; all external references are to APIs. This requirement allows programmers to extend and test existing components more easily, swapping new implementations of libraries in and out of programs at will.

One important restriction on components is that no API may be both imported and exported by the same component. This restriction is required throughout to ground the semantics of operations on components, as discussed in Section 22.7.

Every component has a unique name, used for the purposes of component linking. This name includes a user-provided identifier. In the case of a simple component, the identifier is determined by a component name given at the top of the source file from which it is compiled. A build script may keep a tally on version numbers and append them to the first line of a component, incrementing its tally on each compilation. The name of a compound component is specified as an argument to the `link` operation (described in Section 22.7) that defines it.

Component equivalence is determined nominally to allow mutually recursive linking of components. By programmer convention, identifiers associated with components that are not included in the Fortress standard libraries begin with the reverse of the URL of the development team. A fortress does not allow the installation of distinct components with the same name. Component names are used during `link` and `upgrade` operations to ensure that the restrictions on upgrades to a component are respected, as explained in Section 22.7.

Every component also includes a vendor name, the name of the fortress it is compiled on, and a timestamp, denoting the time of compilation. The time of compilation is measured by the compiling fortress, and the name of the fortress is provided by the fortress automatically. Every timestamp issued by a fortress must be unique. The vendor name typically remains the same throughout a significant portion of the life of a user account, and is best provided as a user environment variable.

In our examples, we use published descriptions of packages in the Java 6.0 API [26] as examples of APIs expressible in our component system. We use, as names for these APIs, the names of the corresponding Java packages, with `java` replaced with Fortress. For example, the following is the beginning of a source file for a fictional application IronCrypto:

```
component Com.Sun.IronCrypto
import Fortress.IO
import Fortress.Security
export Fortress.Crypto
...
end
```

22.3 APIs

Syntax:

Api	::=	`api` *DottedId Import** *AbsDecl** `end`
AbsDecl	::=	*AbsTraitDecl*
	|	*AbsObjectDecl*
	|	*AbsFnDecl*
	|	*AbsVarDecl*
	|	*AbsDimUnitDecl*
	|	*AbsTypeAlias*
	|	*TestDecl*
	|	*PropertyDecl*
AbsTraitDecl	::=	*TraitHeader* (*AbsMdDecl* | *AbsCoercion* | *ApiFldDecl* | *PropertyDecl*)* `end`
AbsObjectDecl	::=	*ObjectHeader* (*AbsMdDecl* | *AbsCoercion* | *ApiFldDecl* | *PropertyDecl*)* `end`
AbsCoercion	::=	[`widening`] `coercion` [*StaticParams*] (*Id IsType*) *CoercionClauses*
ApiFldDecl	::=	*ApiFldMod** *Id IsType*
ApiFldMod	::=	`hidden` | `settable` | *UniversalMod*
AbsVarDecl	::=	*VarWTypes*
	|	*VarWoTypes* : *TypeRef* ...
	|	*VarWoTypes* : *SimpleTupleType*
AbsDimUnitDecl	::=	`dim` *Id* [`default` *Unit*]
	|	(`unit` | `SI_unit`) *Id*$^+$ [: *DimRef*]
	|	`dim` *Id* (`unit` | `SI_unit`) *Id*$^+$
AbsTypeAlias	::=	`type` *Id* [*StaticParams*]

APIs are compiled from special API definitions. These are source files which declare the entities declared by the API, the names of all APIs referred to by those declarations, and prose documentation. In short, the source code of an API

should specify all the information that is traditionally provided for the published APIs of libraries in other languages.

The syntax of an API definition is identical to the syntax of a component definition, except that:

1. An API definition begins with the special reserved word `api` rather than `component`. As with components, the identifiers associated with APIs that are not included in the Fortress standard libraries are prefixed with the reverse of the URL of the development team.
2. An API does not include `export` declarations. (However, it does include `import` declarations, which name the other APIs used in the API definition.)
3. Only declarations (but not definitions!) are included in an API definition except test and property declarations. A method or field declaration may include the modifier `abstract`. (Whether a declaration includes the modifier `abstract` has a significant effect on its meaning, as discussed below).

For the sake of simplicity, every identifier reference in an API definition must refer either to a declaration in a used API (i.e., an API named in an import declaration, or the Fortress core APIs, which are implicitly imported), or to a declaration in the API itself. In this way, APIs differ from signatures in most module systems: they are not parametric in their external dependencies.

Every API has a unique name that consists of a user-provided identifier. As with components, API equivalence is determined nominally. Every API also includes a vendor name, the name of the fortress it is compiled on, and a timestamp.

Component and API names exist in separate namespaces. For convenience, a compiler can also produce an API directly from a project with the same name as the component it is derived from. Such an API includes *matching* declarations of the component. All declarations in the component appear in the API.

A component must include, for every API A it exports, matching definitions for all the declarations in A. A matching definition of a declaration d is a definition d' with the same name as d that includes definitions for all declarations other than the methods or fields declared `abstract` in d. The header and type of d' must be the same as the header and type of d. d' may include additional definitions not declared in d.

For example, consider the APIs Fortress.IO, Fortress.Security, and Fortress.Crypto, with declarations similar to those in their respective Java packages. These APIs are interdependent. For example, both PublicKey in Fortress.Security and SecretKey in Fortress.Crypto extend the trait Fortress.IO.Serializable and the trait CipherSpi in Fortress.Crypto has methods that return values of type AlgorithmParameters in Fortress.Security. So the API Fortress.Crypto must import Fortress.IO and Fortress.Security as follows:

```
api Fortress.Crypto

import Fortress.IO
import Fortress.Security
...
end
```

22.4 Tests in Components and APIs

A component may include definitions of tests, as described in Chapter 19. These definitions are allowed to refer to both test and non-test code defined in the same component or declared in APIs imported by the component.

An API may also include definitions of tests. These definitions may refer to all declarations in the API as well as in any APIs it imports. Tests defined in APIs should be thought of as "executable documentation" that partially specifies the required behavior of the declared entities.

See Section 22.7 for an explanation of how tests defined in components and APIs are executed.

22.5 Type Inference for Components

Type inference for Fortress has been described as a procedure performed over a whole Fortress program in Chapter 20. In this section, we explain how this procedure can be adapted to perform type inference over a simple program component. For a compound component, concatenate all declarations of all constituents in the order specified by the constructing `link` operation. Constituent compound component declarations are recursively concatenated.

Type inference over a simple component C is performed by first expanding C into a self-contained Fortress program, as follows:

1. All program constructs corresponding to declarations in APIs exported by C are expanded so that they include all types and static parameters included in the exported APIs.
2. All types provided by all declarations in the APIs imported by C are prepended to C. Note that these declarations must include types for all variables, functions, fields, and methods; otherwise the APIs that declare them are not well-formed.
3. In order for the resulting expanded program to be well-formed, we assume that all declarations in these APIs are expanded into special definitions that include *empty bodies*. The empty body of such a definition is a conceptual body which cannot be expressed directly in Fortress programs. Because declarations in APIs do not have any elided type, type inference ignores empty bodies.

Once C is expanded, type inference is performed over all program constructs that still include elided types. Empty bodies are ignored.

22.6 Initialization Order for Components

To ensure that all objects and all variables are initialized before their use, execution of program components proceeds according to the procedure defined in this section. This procedure assumes that the program's type hierarchy is already checked to be acyclic.

If a component is a compound component, all constituent components are initialized nondeterministically, but before first use. If a simple component has imports, take the transitive closure of all imported APIs. Collect all declarations in this transitive closure, in any order, and prepend them to the component definition. Finally, for a simple component without imports, initialize all top-level variables and singleton object fields in what we call *demand-driven textual order*: Initialization is done in textual order except when the initialization of one object involves evaluating a reference to another object that is not yet initialized. In such cases, initialization of the object referred to occurs before initialization of the referring object is completed. Note that cyclic references can diverge. Initialization of parametric objects is entirely demand-driven.

22.7 Basic Fortress Operations

We now describe the operations that can be performed on a fortress by developers and end-users for developing, installing, testing, and maintaining components. We can think of these operations as commands to an interactive shell provided by the fortress.

In this section, we discuss operations on a fortress in their most basic form, postponing the discussion of more advanced options, including additional optional parameters, to Section 22.8. Although these more advanced options are critical to performing some real-world tasks with components, it is easier to describe their behavior after the basic forms of operations have been discussed.

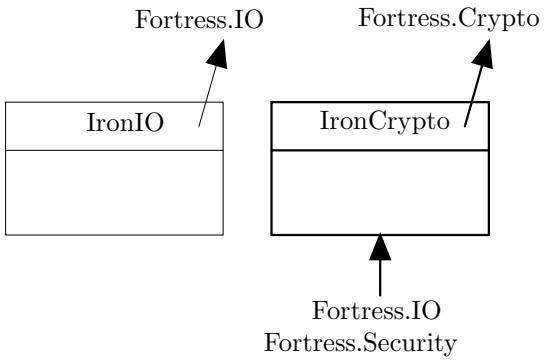

Figure 22.1: Simple components in box notation: A component is represented by a box, with the name of the component at the top of the box. The arrow protruding from the upper right corner of a box is labeled with the APIs exported by the component. The arrow pointing into the bottom of a box is labeled with APIs imported by the component. If no APIs are imported, we elide the arrow.

Compile This operation takes the source code for a simple component (or API) definition and produces a new component object (or API object) that is installed on the fortress. Its type is as follows:

```
compile(file:String):()
```

For example, suppose `IronCrypto.fss` contains the source code for the aforementioned IronCrypto application, which imports Fortress.IO and Fortress.Security, and exports Fortress.Crypto. Suppose we also have source code, `IronIO.fss`, for another application, IronIO, which imports nothing and exports Fortress.IO. We generate these components by compiling the source files:

```
compile("IronIO.fss")
compile("IronCrypto.fss")
```

The results are depicted diagrammatically in Figure 22.1.

Link A collection of one or more components exporting different APIs may be combined to form a new, compound, component by calling the `link` operation, passing the names of the components to link along with the name of the resulting compound component. Syntactically, a `link` operation is written as follows:[2]

```
link(result:String, constituents:String...):()
```

The components being linked are called *constituents* of the resulting component, which exports all the APIs exported by any of its constituents, and imports the APIs imported by at least one of its constituents but not exported by any of them.

For example, we can link the IronIO and IronCrypto libraries compiled above:

```
link(IronLink, IronIO, IronCrypto)
```

The resulting component, illustrated in Figure 22.2, imports Fortress.Security and exports Fortress.IO and Fortress.Crypto.

`link` does not distinguish between simple and compound components, so we can get arbitrarily nested components. For example, we can construct an application CoolCryptoApp by compiling another source code, `IronSecurity.fss`, for the library IronSecurity that imports Fortress.IO and exports Fortress.Security, and then linking the result with IronLink.

[2]We present only the basic form of `link` here. `link` has additional optional arguments that we discuss in the Section 22.8.

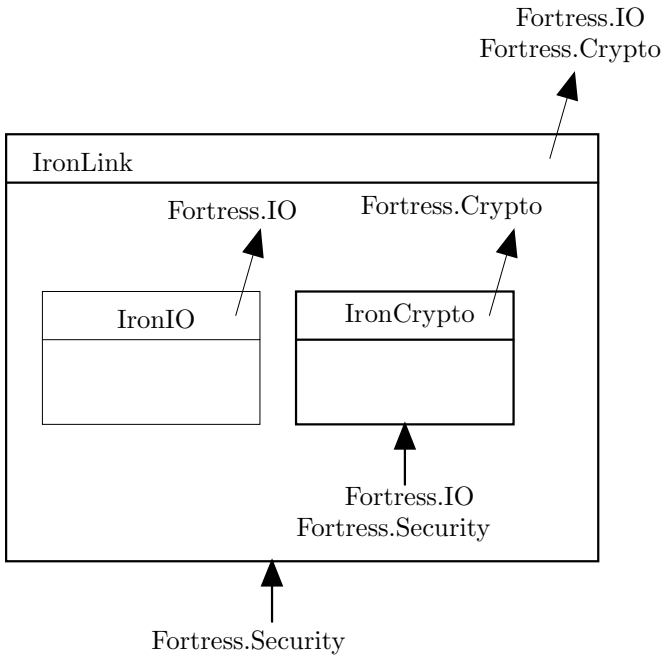

Figure 22.2: A compound component: A component inside another component is a constituent of the component that immediately encloses it.

```
compile(IronSecurity.fss)
link(CoolCryptoApp, IronSecurity, IronLink)
```

The resulting components are illustrated in Figure 22.3.

Two components cannot be linked if they export the same API.[3] This restriction is made for the sake of simplicity; it allows programmers to link a set of components without having to specify explicitly which constituent exporting an API A provides the implementation exported by the linked component, and which constituent connects to the constituents that import A: only one component exports A, so there is only one choice. Although we lose expressiveness with this design, the user interface to link is vastly simplified, and it is rare that including multiple components that export a given API in a set of linked components is even desirable. We discuss how even such rare cases can be supported in Section 22.8.

For a compound component, in addition to the exported and imported APIs, we want to know what its constituents are. It is an invariant of the system that for any compound component C, any API imported by any of its constituents is either imported by C or exported by one of its constituents. This property is crucial for executing components, as we discuss below. A simple component (i.e., one produced directly by compilation) has no constituents.

Execute Components provide implementations of the APIs they export. A component is *executable* if it imports no APIs and it exports the special API $\mathrm{Executable}$, defined as follows:

```
api Executable
run(args : String ...) : ()
end
```

An executable component may be *executed* by calling the `execute` operation, resulting in a call to the component's implementation of the run function in a new process. Arguments to the run function are passed to the shell:

[3]There is one exception to this rule: the special API $\mathrm{Upgradable}$, which is used during upgrades discussed below.

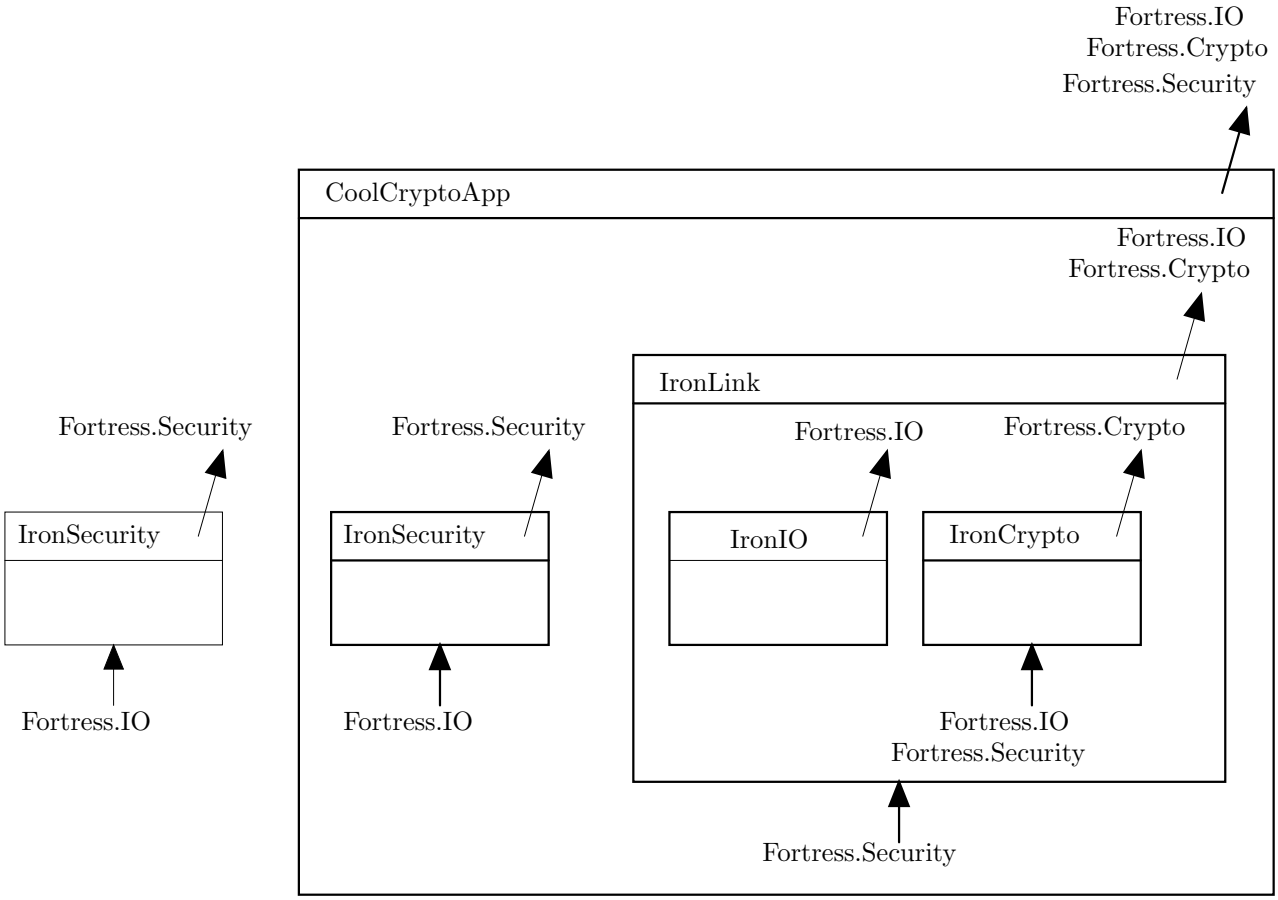

Figure 22.3: Repeated linking

```
execute(componentName:String, args:String...):()
```

We say that a component is being executed when `execute` has been called on that component and has not yet returned, or if it is the constituent component of a component being executed. During an execution, references may be made to APIs exported by a component being executed, which may in turn make references to APIs that it imports.

For references to an API A exported by the component, if the component is simple, then it contains the code necessary to evaluate any reference to an API it exports, possibly making references to APIs that it imports to do so. If the component is compound, then it contains a unique constituent that exports A; the reference is resolved to that constituent component.

For external references within a constituent component, recall that all such references in a component must be to APIs that the component imports. A component being executed either does not import any API (and thus there are no external references to resolve), or else is a constituent of another component that is being executed. In the latter case, the constituent defers the reference to its enclosing component.

For example, suppose $\mathrm{CoolCryptoApp}$ above is the constituent of some executable component, and when that component is executed, it generates a reference to $\mathrm{SecretKey}$ in $\mathrm{Fortress.Crypto}$, which it resolves to $\mathrm{CoolCryptoApp}$. $\mathrm{CoolCryptoApp}$ resolves this reference to $\mathrm{IronLink}$, which resolves it to $\mathrm{IronCrypto}$, which is a simple component. Suppose that in evaluating this reference, $\mathrm{IronCrypto}$ generates a reference to $\mathrm{PublicKey}$ in $\mathrm{Fortress.Security}$. Because $\mathrm{IronCrypto}$ imports $\mathrm{Fortress.Security}$, it resolves this reference to its enclosing component, $\mathrm{IronLink}$, which in turn resolves it to $\mathrm{CoolCryptoApp}$, which resolves it to $\mathrm{IronSecurity}$, which is a simple component.

Not all projects are compiled to components that export $\mathrm{Executable}$. For example, a library component does not usually export $\mathrm{Executable}$.

Upgrade Compound components may be upgraded with new constituent components by calling an `upgrade` operation, passing the name of the component to upgrade (the *target*), the name of a component to upgrade with (the *replacement*), and a name for the resulting component (which we call the *result*). The type of the `upgrade` operation is as follows:

```
upgrade(target:String, replacement:String, result = target):()
```

If no result name is provided, the result is bound to the name of the target, and the target is uninstalled (see below).

For example, we can upgrade $\mathrm{CoolCryptoApp}$ with a component $\mathrm{CoolSecurity}$, which exports $\mathrm{Fortress.Security}$ and imports nothing to $\mathrm{CoolCryptoApp.2.0}$.

```
upgrade(CoolCryptoApp, CoolSecurity, CoolCryptoApp.2.0)
```

The resulting component is illustrated in Figure 22.4. Notice that the constituent, $\mathrm{IronSecurity}$, exporting $\mathrm{Fortress.Security}$ has been replaced.

A component can be upgraded only if it exports the special API $\mathrm{Upgradable}$, defined as follows:

```
api Upgradable
import Component from Components
isValidUpgrade(that : Component) : Boolean
upgrade(that : Component) : Component requires isValidUpgrade(that)
end
```

The $\mathrm{Upgradable}$ API imports a special API $\mathrm{Components}$ that provides handles on $\mathrm{Component}$ and Api objects. The $\mathrm{Components}$ API is described in Chapter 39.

An `upgrade` operation on a component invokes the $isValidUpgrade$ function, as declared in the API $\mathrm{Upgradable}$. This function must take a component and return $true$ if and only if it is legal to upgrade with respect to that component.

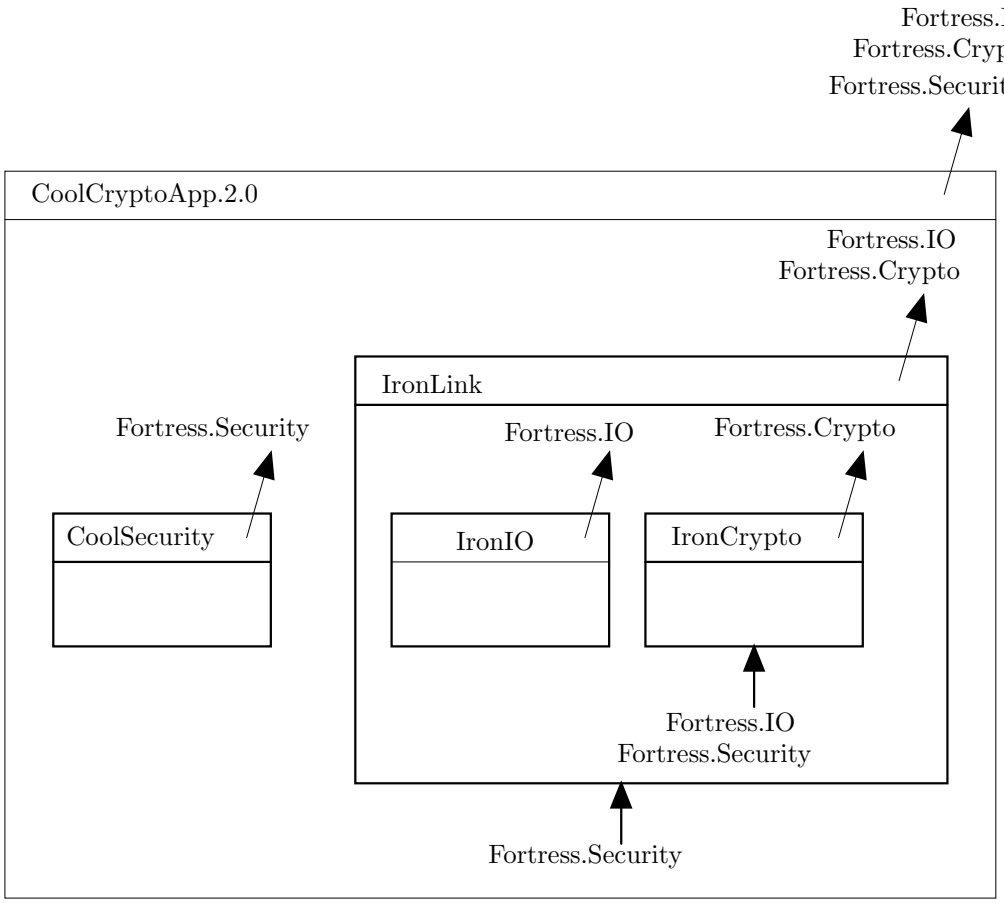

Figure 22.4: An upgraded component

Developers can define their own versions of this component to restrict how their components can be upgraded. For example, they can prevent upgrades with older versions of a component, or with a matching component from an untrusted vendor.

The Upgradable API presents a problem for our model. Its implementation by the various constituent components in a compound component must be accessed during an `upgrade` operation. However, because the exported APIs of the constituent components must be disjoint, they cannot all export Upgradable after linking.

We solve this problem by introducing an additional step during linking. In a `link` operation, a special component, called a *restriction component*, is constructed automatically, based on the provided constituents. This component exports the Upgradable API; its implementation is a function of all the constituents provided to the `link` operation. The provided constituents are then used to construct a new set of constituents that are identical to the provided constituents except that they do not export Upgradable. These new constituents are then combined, along with the restriction component, to form the constituents of a new compound component.

In addition to the constraints imposed by a component's $isValidUpgrade$ function, there are several other conditions that must be met in order for an upgrade to be valid. These conditions are necessary to ensure that the resulting component is well-formed and imports and exports the same APIs as the target: [4]

1. Every API imported by the replacement must be either imported or exported by the target.
2. The APIs exported by the replacement must be a subset of those exported by the target.

[4] These conditions are sufficient provided there are no hidden or constrained APIs, which are discussed in Section 22.8.

3. If the replacement does not export all the APIs that a constituent exports then either the replacement and constituent do not export any APIs in common or the constituent can be upgraded with the replacement.

The rationale for the first two conditions is straightforward: If an API is imported by the replacement but not imported or exported by the target, then references to that API cannot be resolved in the result (unless we also import that API in the result). If an API is exported by the replacement but not the target, then the result will export an API not exported by the target.

The third condition says that the constituents of the target can be partitioned into three sets: those that are subsumed by the replacement, those that are unaffected by the upgrade, and all the rest, which can be upgraded with the replacement. This condition enables recursive propagation of upgrades. That is, an upgrade not only replaces constituents at the top level of the component, but is also propagated into any constituents with which it exports some APIs in common. Thus, in the example above, we could have upgraded CoolCryptoApp with a component that exports Fortress.IO. However, we could not have upgraded CoolCryptoApp with a component that exports both Fortress.Security and Fortress.IO because IronLink exports Fortress.IO but not Fortress.Security. In Section 22.8, we show how hiding and constraining APIs can help us get around many of the limitations that this condition imposes.

Recall that in our system, unlike with dynamic linking, components are encapsulated so that an upgrade to one component does not affect any other component on the system. We can imagine that all operations on components copy the components that they operate on rather than share them. Because components are immutable, these two interpretations are semantically indistinguishable. Convenience operations that support mass upgrades are provided on fortresses (e.g., an `upgradeAll` operation that takes a component and upgrades all components in the fortress that can be upgraded with its argument).

Extract and Install A component installed on a fortress may be *extracted* by calling an `extract` operation on the fortress, passing the name of the component as an argument, along with an argument `prereqs`, denoting the names of all APIs that must be installed on any fortress before this component can be installed.

```
extract(componentName:String, prereqs:Set[\String\] = {}):()
```

Furthermore, the destination fortress must have a component that exports these APIs and is a valid upgrade of the extracted component. Intuitively, a `prereqs` argument allows a component to be serialized without having to include all of its libraries; new libraries can be provided when the component is installed at a destination fortress.

The `prereqs` argument is optional; if omitted, the extracted component can be installed on any fortress. Any component can be extracted; however only compound components can be extracted with a `prereqs` argument: because extracted components must be upgradable with respect to a component exporting their `prereqs`, no `prereqs` argument makes sense for a simple component.

The APIs included in a `prereqs` argument must be the APIs exported by some subset of the extracted component's constituents (or a subset of the constituents of one of its constituents, and so on, due to recursive updating).

The extracted component is serialized to a file, including all the APIs it refers to (and, transitively, all APIs they refer to) and all constituent components, except those that export the `prereqs`. This operation does not remove the extracted component from the fortress; there is a separate `uninstall` operation for that.

When the component is extracted, if no `prereqs` were passed to the `extract` operation, then the contents of the file can be deserialized by any fortress into the extracted component, which can be installed on the fortress. However, if `prereqs` were passed to `extract`, then the file must be deserialized into a component that exports only the Installable API:

```
api Installable
import Component from Components
reconstitute(candidate : Component) : Component
end
```

The deserialized component is immediately linked with preferred implementations of all of its imported APIs. (Preferred implementations of APIs are maintained in a table by a fortress, which maps each API to a list of components that implements it, in order of preference). Because the deserialized and linked component exports the $\mathrm{Installable}$ API, it has a $reconstitute$ function that takes a $candidate$ component, which exports the `prereqs` APIs, and checks whether the given component satisfies the $isValidUpgrade$ condition of the extracted component. If so, it returns the extracted component upgraded with the given component. The $reconstitute$ function is called by the fortress with a new component, formed by linking the preferred components for each API in the extracted components' `prereqs` argument.

Note that an extracted component with `prereqs` APIs is *not* the same as an extracted component that imports the same APIs but has no `prereqs` APIs. The latter can always be installed on a fortress, and then can be subsequently linked with any component that exports the imported APIs. In contrast, the fortress has no access to an extracted component with `prereqs` APIs unless it has a component that exports these APIs and satisfies the $isValidUpgrade$ function of the extracted component. This difference provides a means for controlling access to the extracted component, for security, legal, or other reasons.

Syntactically, an `install` operation takes the name of a file constraining an extracted component. The `install` operation is overloaded with another operation that takes the name of a component to match `prereqs`. If this optional argument is provided, and the deserialized component exports the $\mathrm{Installable}$ API, then the $reconstitute$ function is called with the component denoted by the optional argument of `install`, rather than the fortress' preferred implementation of the `prereqs` APIs. Install operations are written as follows:

```
install(file:String):()
install(file:String, prereqs:Set[\String\]):()
```

By default, a fortress adds a newly installed component to the head of the "preferred" list for every API it exports. However, this default may be overridden by the end-user; an end-user may modify the table or even map some APIs differently during a particular installation. If one or more of the APIs required by an extracted component is not mapped to an API on the destination fortress, an exception is thrown.

There is a corresponding operation for APIs, `installAPI`, that takes a serialization of a set of APIs and installs them into a fortress.

```
installAPI(file:String):()
```

This set of APIs must be closed under imports. If an API that is installed in this way is already installed on the fortress, the definitions must match exactly, or an exception is thrown.

Uninstall An `uninstall` operation takes the name of a component as an argument and removes the top-level binding of that component from a fortress. Note that the uninstalled component may have been linked to other components, or used as a replacement in an upgrade, and the result may still be installed; an `uninstall` operation will not affect these other components.

```
uninstall(file:String):()
```

There is a corresponding operation for APIs, `uninstallAPI`, that removes an API from a fortress.

```
uninstallAPI(file:String):()
```

Typically, this operation is used only to remove APIs that have been corrupted in some fashion.

Testing A component can be tested by calling the method $runTests$ on it:

$runTests(inclusive = true) : ()$

This method runs all test functions defined in the component. All test functions are run in parallel; each test function is run for each combination of test cases (bound in its generator list as described in Section 19.2) in parallel. In the case of a compound component, the set of defined test functions consists of all test functions defined by all constituent components and by all exported APIs. The set of test functions run can be limited by first hiding the tested component in a more restrictive API. The set of test functions can also be expanded by linking with a component defining additional test functions.

The $runTests$ method includes a keyword parameter $inclusive$ that defaults to $true$. If this parameter is set to $false$, only test functions defined in the APIs exported by the component are run.

22.8 Advanced Features of Fortress Operations

The system we have described thus far provides much of the desired functionality of a component system. However it has a few significant weaknesses:

1. It exposes to everyone all the APIs used in the development of a project.
2. By allowing access to these APIs, it inhibits significant cross-component optimization.
3. It prevents components that use two different implementations of the same API from being linked, even if they never actually pass references to that API between each other.
4. It restricts the upgradability of compound components, as described earlier.

We can mitigate all these shortcomings by providing two simple operations, `hide` and `constrain`. Informally, `hide` makes APIs no longer visible from outside the component so that they cannot be upgraded, and `constrain` merely prevents them from being exported. An API that is constrained but not hidden can still be upgraded. There are other subtle consequences of this distinction, which we discuss as they arise.

Some of the properties about the APIs exported by a component discussed in Section 22.7 are actually properties of APIs that are visible or provided by a component. For example, APIs visible in a component cannot be imported by that component, even if they are not exported. Other properties are really properties only of the exported APIs. Most importantly, components that do not export any common APIs can be linked, as can components that share only visible APIs.

Constrain A `constrain` operation takes a component name of an installed component, a new component name, and a set of APIs, and produces a new component that does not export any of the APIs specified. Syntactically, we write:

```
constrain(source:String, destination = source, apis:Set[\String\]):()
```

If no `destination` name is provided, the name of the `source` is used.

The set of APIs provided must be a subset of the APIs exported by the component. Also, recall that every API used by an API exported by a component must be imported or exported by that component. Thus, if we constrain an API that is used by any other API exported by the component, then we must also constrain that other API.

If the component is a simple component, we first link it by itself, and then apply `constrain` to the result.

Hide A `hide` operation is like a `constrain` operation, except that the given set of APIs is subtracted from the visible and provided APIs, along with the exported APIs, in the resulting component.

```
hide(source:String, destination = source, apis:Set[\String\]):()
```

The requirement of APIs being imported or exported whenever an API using them is exported also applies to visible APIs. Thus, if we hide an API used by another exported API, we must hide that other API as well.

Link With constrained APIs, there is a new restriction on link: Any API visible in one constituent and imported by another must be exported by some constituent. This restriction is necessary because an API visible in a component cannot be imported by that component. Thus, if one of the component's constituents imports that API, then the API must be provided by some other constituent. Other than that, the `link` operation is largely unchanged: the visible APIs are just all the APIs visible in any constituent, and the provided APIs are just those exported by any constituent. There is a subtle additional restriction on how linked components can be upgraded, which we discuss below.

Rather than requiring users and developers to call `constrain` and `hide` directly, we provide optional parameters to the `link` operation to do these operations immediately. The `link` operation has the following type:

```
link(result:String, constituents:String..., exports = {}, hide = {}):()
```

If the `exports` clause is present, only those APIs listed in the set following `exports` are exported; the others are constrained. If the `hide` clause is present, those APIs listed in the set following `hide` are hidden. An exception is thrown if the `exports` clause contains any API not exported by any constituent, or if the `hide` clause contains any API not visible in any constituent.

Hiding enables us to handle the rare case in which programmers want to link multiple components that implement the same API without upgrading them to use the same implementation. Before linking, the programmer simply hides (or constrains) the API in every component that exports it except the one that should provide the implementation for the new compound component.

For example, suppose we wish to link the following two components:

- A component NetApp that imports Fortress.IO and exports the Fortress.Net API.
- A component EditApp that imports Fortress.IO and exports the Fortress.Swing.Textrf API.

We want to link these two components to use in building an application for editing messages and sending them over a network. But we want to use different implementations of Fortress.IO (e.g., IOApp1 and IOApp2 for the two components). We simply perform the following operations:

```
link(temp1, NetApp, IOApp1, exports = {Fortress.Net}, hide = {Fortress.IO})
link(temp2, EditApp,IOApp2, exports = {Fortress.Swing.Textrf}, hide = {Fortress.IO})
link(NetEdit, temp1, temp2)
```

In this case, the NetEdit component does not export, or even make visible, Fortress.IO at all.

Upgrade For the `upgrade` operation, there is no change at all in the semantics. However, because hiding and constraining APIs allow us to change the APIs exported by a component, it is possible to do some upgrades that are not possible without these operations.

For example, suppose we have a component IOSecurity that exports Fortress.IO and Fortress.Security, and we want to upgrade CoolCryptoApp with IOSecurity. As discussed above, we cannot use IOSecurity directly because IronLink exports Fortress.IO but not Fortress.Security. We can get around this restriction by doing two upgrades, one with Fortress.Security hidden and the other with Fortress.IO hidden.

```
hide(IOSecurity, NewIO, Fortress.Security)
hide(IOSecurity, NewSecurity, Fortress.IO)
upgrade(CoolCryptoApp, NewSecurity, temp1)
upgrade(CoolCryptoApp.3.0, temp1, NewIO)
```

The resulting component is shown in Figure 22.5.

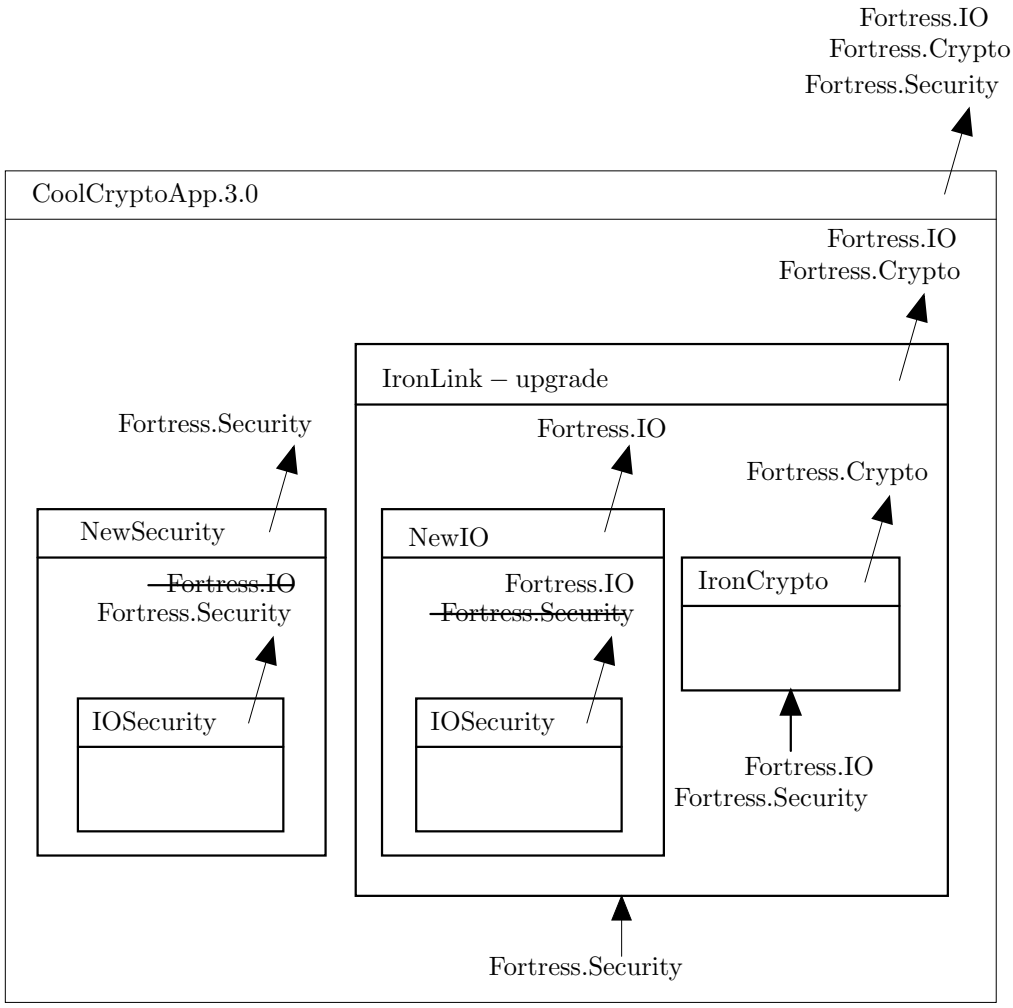

Figure 22.5: Upgrading with hidden APIs: Crossed out APIs are hidden.

The interplay between imported, exported, visible and provided APIs introduces subtleties that not present in our discussion above. In particular, the last of the three conditions imposed for well-formedness of upgrades is modified to state that for any constituent that is not subsumed by a replacement component, either it can be upgraded with the replacement, or its *visible* APIs are disjoint from the APIs exported by the replacement (i.e., it is unaffected by the upgrade). To maintain the invariant that no two constituents export the same API, we need another condition, which was implied by the previous condition when no APIs were constrained or hidden: if the replacement subsumes any constituents of the target, then its exported APIs must exactly match the exported APIs of some subset of the constituents of the target. In practice, this restriction is rarely a problem; in most cases, a user wishes to upgrade a target with a new version of a single constituent component, where the APIs exported by the old and new versions are either an exact match, or there are new APIs introduced by the new component that have no implementation in the target.

Part III

Fortress APIs and Documentation for Application Programmers

Chapter 23

Objects

23.1 The Trait Fortress.Core.Object

The trait Object is a single root of the type hierarchy; every object in Fortress has trait Object and therefore every object implements the methods of this trait.

```
trait Object extends { EquivalenceRelation⟦Object, ≡⟧, IdentityOperator⟦Object⟧ }
  opr ≡(self, other: Object): Boolean
  opr IDENTITY(self): Object
  hash(maxval: ℕ64): ℕ64
  hash(maxval: ℕ32): ℕ32
  getter hashCode(): ℕ64
  toString(): String
  property ∀(x, y, n: ℕ64) x ≡ y → x.hash(n) ≡ y.hash(n)
  property ∀(x, y, n: ℕ32) x ≡ y → x.hash(n) ≡ y.hash(n)
  property ∀(x) x.hashCode ≡ x.hash(2^64 − 1)
  property ∀(x, y) x ≡ y → x.toString() = y.toString()
end
```

23.1.1 opr ≡(self, other: Object): Boolean

The infix operator $\equiv$ (object equivalence) is used to decide whether two objects are "the same object" in the strictest sense possible; this is described in detail in Section 10.4.

For $\not\equiv$ see Section 26.1.1.

23.1.2 opr IDENTITY(self): Object

The operator `IDENTITY` simply returns its argument. (This may not be terribly useful for applications programming, but it has technical uses for specifying contracts and algebraic properties in libraries as described in Section 37.3.)

23.1.3 $hash(maxval\colon \mathbb{N}64)\colon \mathbb{N}64$
23.1.4 $hash(maxval\colon \mathbb{N}32)\colon \mathbb{N}32$

The $hash$ method returns a ***hash value*** for the object as an unsigned integer that is less than or equal to the $maxval$ argument. This hash value is not necessarily consistent from one Fortress application to another, nor from one execution of a Fortress application to another execution of the same application, but the hash value produced for a given value of the $maxval$ argument remains fixed during the execution of a single Fortress application. There is no defined relationship between hash values produced for the same object but with different $maxval$ values. Fortress programmers and implementors should be aware that the performance of hash tables is likely to be improved if, for any given collection of objects and given value for the $maxval$ argument, the $hash$ method assigns hash values to those objects with relatively uniform distribution.

23.1.5 `getter` $hashCode()\colon \mathbb{N}64$

Every object has associated with it a 64-bit unsigned integer value called its ***hash code***; this is the value returned by the $hash$ method when given the argument $2^{64} - 1$. Hash codes are not necessarily consistent from one Fortress application to another, nor from one execution of a Fortress application to another execution of the same application, but remain fixed during the execution of a single Fortress application. It is permitted for two objects to have the same hashCode, but Fortress programmers and implementors should be aware that assigning distinct hash codes to distinct objects may improve the performance of hash tables.

The trait Object defines its $hash$ methods in terms of $hashCode$; therefore it suffices for a subtrait to override $hashCode$ to get the benefit of the $hash$ methods as well.

23.1.6 $toString()\colon \mathrm{String}$

The general contract of $toString$ is that it returns a string that "textually represents" this object. The idea is to provide a concise but informative representation that will be useful to a person reading it.

Chapter 24

Booleans and Boolean Intervals

24.1 The Trait Fortress.Core.Boolean

```
trait Boolean
      extends {BooleanAlgebra⟦Boolean, ∧, ∨, ¬, ⊻, falsevalue, truevalue⟧,
               BooleanAlgebra⟦Boolean, ∧, ∨, ¬, ⊕, falsevalue, truevalue⟧,
               BooleanAlgebra⟦Boolean, juxtaposition, ∨, ¬, ⊻, falsevalue, truevalue⟧,
               BooleanAlgebra⟦Boolean, juxtaposition, ∨, ¬, ⊕, falsevalue, truevalue⟧,
               IdentityEquality⟦Boolean⟧,
               EquivalenceRelation⟦Boolean, ≡⟧,
               EquivalenceRelation⟦Boolean, ↔⟧,
               TotalOrder⟦Boolean, →⟧,
               Symmetric⟦Boolean, ∧⟧, Symmetric⟦Boolean, ∨⟧,
               Symmetric⟦Boolean, ⊻⟧, Symmetric⟦Boolean, ⊕⟧,
               Symmetric⟦Boolean, ⊼⟧, Commutative⟦Boolean, ⊼⟧,
               Symmetric⟦Boolean, ⊽⟧, Commutative⟦Boolean, ⊽⟧ }
      comprises {}
    coercion ⟦bool b⟧(x: BooleanLiteral⟦b⟧)
    opr juxtaposition (self, other: Boolean): Boolean
    opr ∧(self, other: Boolean): Boolean
    opr ∧(self, other: () → Boolean): Boolean
    opr ∨(self, other: Boolean): Boolean
    opr ∨(self, other: () → Boolean): Boolean
    opr ¬(self): Boolean
    opr ⊻(self, other: Boolean): Boolean
    opr ⊕(self, other: Boolean): Boolean
    opr ≡(self, other: Boolean): Boolean
    opr =(self, other: Boolean): Boolean
    opr ↔(self, other: Boolean): Boolean
    opr →(self, other: Boolean): Boolean
    opr →(self, other: () → Boolean): Boolean
    opr ⊼(self, other: Boolean): Boolean
    opr ⊽(self, other: Boolean): Boolean
    getter truevalue(): Boolean
    getter falsevalue(): Boolean
```

```
    opr ≡(self, other: Boolean): Boolean
    getter hashCode(): ℕ64
    toString(): String
end
test testData[ ] = { false, true }
```

24.1.1 coercion ⟦bool b⟧(x: BooleanLiteral⟦b⟧)

A boolean literal can always serve as a Boolean value.

24.1.2 opr juxtaposition (self, other: Boolean): Boolean

Juxtaposition of boolean expressions is equivalent to using the logical AND operator ∧.

24.1.3 opr ∧(self, other: Boolean): Boolean
24.1.4 opr ∧(self, other: () → Boolean): Boolean

The logical AND operator ∧ (AND) returns *true* if both arguments are *true*; otherwise it returns *false*.

The conditional logical AND operator ∧: (AND:) examines its first argument; if it is *false*, the result is *false*, and the second argument (a thunk) is not evaluated. But if the first argument is *true*, the second argument is evaluated and its result becomes the result of the conditional logical AND operator expression.

24.1.5 opr ∨(self, other: Boolean): Boolean
24.1.6 opr ∨(self, other: () → Boolean): Boolean

The logical OR operator ∨ (OR) returns *false* if both arguments are *false*; otherwise it returns *true*.

The conditional logical OR operator ∨: (OR:) examines its first argument; if it is *true*, the result is *true*, and the second argument (a thunk) is not evaluated. But if the first argument is *false*, the second argument is evaluated and its result becomes the result of the conditional logical OR operator expression.

24.1.7 opr ¬(self): Boolean

The logical NOT operator ¬ (NOT) returns *true* if its argument is *false*; it returns *false* if its argument is *true*.

24.1.8 opr ⩒(self, other: Boolean): Boolean
24.1.9 opr ⊕(self, other: Boolean): Boolean

The logical exclusive OR operator ⩒ (XOR) returns *true* if the arguments are different, one being *true* and the other *false*; it returns *false* if both arguments are *true* or both arguments are *false*.

The operator ⊕ (OPLUS) does the same thing as ⩒.

24.1.10 `opr` $\equiv$(`self`, *other*: Boolean): Boolean
24.1.11 `opr` $=$(`self`, *other*: Boolean): Boolean
24.1.12 `opr` $\leftrightarrow$(`self`, *other*: Boolean): Boolean

The logical equivalence, or exclusive NOR, operator $\equiv$ (`EQV`) returns *true* if both arguments are *true* or both arguments are *false*; it returns *false* if the arguments are different, one being *true* and the other *false*. (Thus its behavior on boolean values happens to be exactly the same as that of the strict equivalence operator $\equiv$.)

The equality operator $=$ and the if-and-only-if operator $\leftrightarrow$ (`IFF`) do the same thing as $\equiv$.

For $\not\equiv$ see Section 26.1.2. For $\neq$ see Section 26.1.4.

24.1.13 `opr` $\rightarrow$(`self`, *other*: Boolean): Boolean
24.1.14 `opr` $\rightarrow$(`self`, *other*: () $\rightarrow$ Boolean): Boolean

The logical implication operator $\rightarrow$ (`IMPLIES`) returns *false* if the first argument is *true* but the second argument is *false*; otherwise it returns *true*.

The conditional logical implication operator $\rightarrow$: (`IMPLIES:`) examines its first argument; if it is *false*, the result is *true*, and the second argument (a thunk) is not evaluated. But if the first argument is *true*, the second argument is evaluated and its result becomes the result of the conditional logical implication operator expression.

24.1.15 `opr` $\overline{\wedge}$(`self`, *other*: Boolean): Boolean

The logical NAND (NOT AND) operator $\overline{\wedge}$ (`NAND`) returns *false* if both arguments are *true*; otherwise it returns *true*.

24.1.16 `opr` $\overline{\vee}$(`self`, *other*: Boolean): Boolean

The logical NOR (NOT OR) operator $\overline{\vee}$ (`NOR`) returns *false* if both arguments are *false*; otherwise it returns *true*.

24.1.17 `getter` *true*(): Boolean
24.1.18 `getter` *false*(): Boolean

The getter *true* returns the value *true*, and the getter *false* returns the value *false*. These are defined primarily for the benefit of the BooleanAlgebra traits that Boolean extends.

24.1.19 `opr` $\not\equiv$(`self`, *other*: Boolean): Boolean

Two boolean values are strictly equivalent if and only if they are the same boolean value (that is, both *true* or both *false*).

24.1.20 `getter` $hashCode(): \mathbb{N}64$

24.1.21 $toString(): \mathrm{String}$

The $toString$ method returns either $true$ or $false$ as appropriate.

24.2 The Trait Fortress.Standard.BooleanInterval

A boolean interval is a set of boolean values. There are two distinct boolean values, $true$ and $false$, and therefore there are four distinct boolean intervals, which for convenience are given names:

$$\begin{aligned} \mathrm{True} &= \{true\} \\ \mathrm{False} &= \{false\} \\ \mathrm{Uncertain} &= \{true, false\} \\ \mathrm{Impossible} &= \{\,\} \end{aligned}$$

Logical operations on intervals obey the interval containment rule: the result interval must contain every boolean result that can be produced by applying the operator to a boolean value taken from each argument interval. For example, if P and Q are boolean intervals, then by definition $P \wedge Q = \{\, x \wedge y \mid x \leftarrow P, y \leftarrow Q \,\}$.

A principal application of boolean intervals is to express the results of numerical comparison of numerical intervals. In this way numerical comparisons can also obey the interval containment rule.

Set operations such as $\cup$ and $\cap$ may also be used on boolean intervals.

`trait` BooleanInterval
  `extends` { BooleanAlgebra⟦BooleanInterval, ∩, ∪, `SET_COMPLEMENT`, `SYMDIFF`, *empty*, *universe*⟧,
    Set⟦Boolean⟧,
    BinaryIntervalContainment⟦BooleanInterval, Boolean, ∧⟧,
    BinaryIntervalContainment⟦BooleanInterval, Boolean, ∨⟧,
    BinaryIntervalContainment⟦BooleanInterval, Boolean, $\underline{\vee}$⟧,
    BinaryIntervalContainment⟦BooleanInterval, Boolean, ≡⟧,
    BinaryIntervalContainment⟦BooleanInterval, Boolean, =⟧,
    BinaryIntervalContainment⟦BooleanInterval, Boolean, ↔⟧,
    BinaryIntervalContainment⟦BooleanInterval, Boolean, $\overline{\wedge}$⟧,
    BinaryIntervalContainment⟦BooleanInterval, Boolean, $\overline{\nabla}$⟧,
    BinaryIntervalContainment⟦BooleanInterval, Boolean, →⟧,
    UnaryIntervalContainment⟦BooleanInterval, Boolean, ¬⟧,
    Generator⟦Boolean⟧ }
  `comprises` {}
 `coercion` (*x*: Boolean)
 `opr` ∧(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` ∨(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` ¬(`self`): BooleanInterval
 `opr` $\underline{\vee}$(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` ⊕(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` ≡(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` =(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` ↔(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` →(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` $\overline{\wedge}$(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` $\overline{\nabla}$(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` ∈(*other*: Boolean, `self`): Boolean
 `opr` ∩(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` ∪(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` `SET_COMPLEMENT`(`self`): BooleanInterval
 `opr` `SYMDIFF`(`self`, *other*: BooleanInterval): BooleanInterval
 `opr` \(`self`, *other*: BooleanInterval): BooleanInterval
 possibly(`self`): Boolean
 necessarily(`self`): Boolean
 certainly(`self`): Boolean
 `getter` *empty*(): BooleanInterval
 `getter` *universe*(): BooleanInterval
 `opr` ≡(`self`, *other*: BooleanInterval): Boolean
 `getter` *hashCode*(): ℤ64
 toString(): String
 `property` *true* ∈ True ∧ *false* ∉ True
 `property` *true* ∉ False ∧ *false* ∈ False
 `property` *true* ∈ Uncertain ∧ *false* ∈ Uncertain
 `property` *true* ∉ Impossible ∧ *false* ∉ Impossible
 `property` ∀(*a*) *necessarily*(*a*) ≡ ¬*possibly*(¬*a*)
 `property` ∀(*a*) *possibly*(*a*) ↔ *true* ∈ *a*
 `property` ∀(*a*) *certainly*(*a*) ↔ (*true* ∈ *a* ∧ *false* ∉ *a*)
 `property` ∀(*a*, *b*) (*a* $\overline{\wedge}$ *b*) ↔ ¬(*a* ∧ *b*)
 `property` ∀(*a*, *b*) (*a* $\overline{\nabla}$ *b*) ↔ ¬(*a* ∨ *b*)
`end`
True: BooleanInterval

False: BooleanInterval
Uncertain: BooleanInterval
Impossible: BooleanInterval
`test` *testData*[ ] = { True, False, Uncertain, Impossible }

24.2.1 `coercion` (x: Boolean)

A boolean value can always serve as a BooleanInterval value. The value *true* is coerced to True; the value *false* is coerced to False.

24.2.2 `opr` ∧(`self`, *other*: BooleanInterval): BooleanInterval

The logical AND operator ∧ (`AND`) returns Impossible if either argument is Impossible; otherwise it returns False if either argument is False; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns True. It obeys the interval containment rule. The ∧ operator may be described by this table:

∧	Uncertain	True	False	Impossible
Uncertain	Uncertain	Uncertain	False	Impossible
True	Uncertain	True	False	Impossible
False	False	False	False	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.3 `opr` ∨(`self`, *other*: BooleanInterval): BooleanInterval

The logical OR operator ∨ (`OR`) returns Impossible if either argument is Impossible; otherwise it returns True if either argument is True; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns False. It obeys the interval containment rule. The ∧ operator may be described by this table:

∧	Uncertain	True	False	Impossible
Uncertain	Uncertain	True	Uncertain	Impossible
True	True	True	True	Impossible
False	Uncertain	True	False	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.4 `opr` ¬(`self`): BooleanInterval

The logical NOT operator ¬ (`NOT`) returns Impossible if its argument is Impossible, Uncertain if its argument is Uncertain, False if its argument is True, and True if its argument is False. It obeys the interval containment rule.

24.2.5 `opr` $\underline{\vee}$(`self`, *other*: BooleanInterval): BooleanInterval
24.2.6 `opr` $\oplus$(`self`, *other*: BooleanInterval): BooleanInterval

The logical exclusive OR operator $\underline{\vee}$ (`XOR`) returns Impossible if either argument is Impossible; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns False if the arguments are strictly equivalent; otherwise it returns True. It obeys the interval containment rule.

The operator $\oplus$ (`OPLUS`) does the same thing as $\underline{\vee}$. The $\underline{\vee}$ or $\oplus$ operator may be described by this table:

$\underline{\vee}$ or $\oplus$	Uncertain	True	False	Impossible
Uncertain	Uncertain	Uncertain	Uncertain	Impossible
True	Uncertain	False	True	Impossible
False	Uncertain	True	False	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.7 `opr` $\equiv$(`self`, *other*: BooleanInterval): BooleanInterval
24.2.8 `opr` $=$(`self`, *other*: BooleanInterval): BooleanInterval
24.2.9 `opr` $\leftrightarrow$(`self`, *other*: BooleanInterval): BooleanInterval

The logical equivalence, or exclusive NOR, operator $\equiv$ (`EQV`) returns Impossible if either argument is Impossible; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns True if the arguments are strictly equivalent; otherwise it returns False. It obeys the interval containment rule. (Thus its behavior on boolean interval values is *not* the same as that of the strict equivalence operator $\equiv$.)

The equality operator $=$ and the if-and-only-if operator $\leftrightarrow$ (`IFF`) do the same thing as $\equiv$. The $\equiv$ or $=$ or $\leftrightarrow$ operator may be described by this table:

$\equiv$ or $=$ or $\leftrightarrow$	Uncertain	True	False	Impossible
Uncertain	Uncertain	Uncertain	Uncertain	Impossible
True	Uncertain	True	False	Impossible
False	Uncertain	False	True	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.10 `opr` $\rightarrow$(`self`, *other*: BooleanInterval): BooleanInterval

The logical implication operator $\rightarrow$ (`IMPLIES`) returns Impossible if either argument is Impossible; otherwise it returns True if the first argument is False or the second argument is True; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns False. It obeys the interval containment rule. The $\rightarrow$ operator may be described by this table:

$\rightarrow$	Uncertain	True	False	Impossible
Uncertain	Uncertain	True	Uncertain	Impossible
True	Uncertain	True	False	Impossible
False	True	True	True	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.11 `opr` $\overline{\wedge}$(`self`, *other*: BooleanInterval): BooleanInterval

The logical NAND (NOT AND) operator $\overline{\wedge}$ (`NAND`) returns Impossible if either argument is Impossible; otherwise it returns True if either argument is False; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns False. It obeys the interval containment rule. The $\overline{\wedge}$ operator may be described by this table:

$\overline{\wedge}$	Uncertain	True	False	Impossible
Uncertain	Uncertain	Uncertain	True	Impossible
True	Uncertain	False	True	Impossible
False	True	True	True	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.12 `opr` $\overline{\vee}$(`self`, *other*: BooleanInterval): BooleanInterval

The logical NOR (NOT OR) operator $\overline{\vee}$ (`NOR`) returns Impossible if either argument is Impossible; otherwise it returns False if either argument is True; otherwise it returns Uncertain if either argument is Uncertain; otherwise it returns True. It obeys the interval containment rule. The $\overline{\vee}$ operator may be described by this table:

$\overline{\vee}$	Uncertain	True	False	Impossible
Uncertain	Uncertain	False	Uncertain	Impossible
True	False	False	False	Impossible
False	Uncertain	False	True	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.13 `opr` $\in$(*other*: Boolean, `self`): Boolean

The operator $\in$ (`IN`) returns *true* if its first argument, a boolean value, is contained in its second argument, a boolean interval regarded as a set; otherwise it returns *false*. The $\in$ operator may be described by this table:

$\in$	Uncertain	True	False	Impossible
true	*true*	*true*	*false*	*false*
false	*true*	*false*	*true*	*false*

24.2.14 `opr` $\cap$(`self`, *other*: BooleanInterval): BooleanInterval

The intersection operator $\cap$ (`INTERSECTION` or `CAP`) returns Impossible if either argument is Impossible; otherwise, if either argument is Uncertain, it returns the other argument; otherwise, if the arguments are the same value (strictly equivalent), it returns that value; otherwise it returns Impossible. The $\cap$ operator may be described by this table:

$\cap$	Uncertain	True	False	Impossible
Uncertain	Uncertain	True	False	Impossible
True	True	True	Impossible	Impossible
False	False	Impossible	False	Impossible
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.15 opr ∪(self, *other*: BooleanInterval): BooleanInterval

The union operator ∪ (`UNION` or `CUP`) returns Uncertain if either argument is Uncertain; otherwise, if either argument is Impossible, it returns the other argument; otherwise, if the arguments are the same value (strictly equivalent), it returns that value; otherwise it returns Uncertain. The ∪ operator may be described by this table:

∪	Uncertain	True	False	Impossible
Uncertain	Uncertain	Uncertain	Uncertain	Uncertain
True	Uncertain	True	Uncertain	True
False	Uncertain	Uncertain	False	False
Impossible	Uncertain	True	False	Impossible

24.2.16 opr SET_COMPLEMENT(self): BooleanInterval

The set complement operator `SET_COMPLEMENT` returns Uncertain if its argument is Impossible, Impossible if its argument is Uncertain, False if its argument is True, and True if its argument is False.

24.2.17 opr SYMDIFF(self, *other*: BooleanInterval): BooleanInterval

The symmetric difference operator `SYMDIFF` produces a result that contains a given boolean value if and only if exactly one of the arguments contains that boolean value. The `SYMDIFF` operator may be described by this table:

SYMDIFF	Uncertain	True	False	Impossible
Uncertain	Impossible	False	True	Uncertain
True	False	Impossible	Uncertain	True
False	True	Uncertain	Impossible	False
Impossible	Uncertain	True	False	Impossible

24.2.18 opr \(self, *other*: BooleanInterval): BooleanInterval

The set difference operator \ (`SETMINUS`) produces a result that contains a given boolean value if and only if the first argument contains that boolean value but the second argument does not. The \ operator may be described by this table:

\	Uncertain	True	False	Impossible
Uncertain	Impossible	False	True	Uncertain
True	Impossible	Impossible	True	True
False	Impossible	False	Impossible	False
Impossible	Impossible	Impossible	Impossible	Impossible

24.2.19 *possibly*(`self`): Boolean
24.2.20 *necessarily*(`self`): Boolean
24.2.21 *certainly*(`self`): Boolean

The predicate *possibly* returns *true* if and only if *true* is a member of this boolean interval.

The predicate *necessarily* returns *true* if and only if *false* is not a member of this boolean interval (thus "necessarily" is a concise way of saying "not possibly not").

The predicate *certainly* returns *true* if and only if this boolean interval is True, that is, it contains *true* but not *false* (thus "certainly" is a concise way of saying "both possibly and necessarily").

The fourteen nontrivial functions from a value x of type BooleanInterval to type Boolean may thus be expressed as follows:

$$necessarily(x) \wedge necessarily(\neg x)$$
$$certainly(\neg x)$$
$$necessarily(\neg x)$$
$$certainly(x)$$
$$necessarily(x)$$
$$possibly(x) \equiv necessarily(x)$$
$$necessarily(x) \vee necessarily(\neg x)$$
$$possibly(x) \wedge possibly(\neg x)$$
$$possibly(x) \veebar necessarily(x)$$
$$possibly(\neg x)$$
$$\neg certainly(x)$$
$$possibly(x)$$
$$\neg certainly(\neg x)$$
$$possibly(x) \vee possibly(\neg x))$$

There are other ways to express some of them; for example, $necessarily(x) \wedge necessarily(\neg x)$ is the same as $x \equiv \text{Impossible}$, and $possibly(x) \wedge possibly(\neg x)$ is the same as $x \equiv \text{Uncertain}$.

24.2.22 `getter` *empty*(): BooleanInterval
24.2.23 `getter` *universe*(): BooleanInterval

The getter *empty* returns the value ***Impossible***, and the getter *universe* returns the value ***Uncertain***. These are defined primarily for the benefit of the BooleanAlgebra trait that BooleanInterval extends.

24.2.24 `opr` ≡(`self`, *other*: BooleanInterval): Boolean

Two boolean intervals are strictly equivalent if and only if they are the same boolean interval.

24.2.25 `getter` *hashCode*(): ℤ64

24.2.26 *toString*(): String

The *toString* method returns either "`True`" or "`False`" or "`Uncertain`" or "`Impossible`" as appropriate.

24.3 Top-level BooleanInterval Values

24.3.1 True: BooleanInterval
24.3.2 False: BooleanInterval
24.3.3 Uncertain: BooleanInterval
24.3.4 Impossible: BooleanInterval

The immutable variables True, False, Uncertain, and Impossible have as their values the four boolean intervals. They are top-level variables declared in the Fortress standard libraries.

Chapter 25

Numbers

25.1 Rational Numbers

The trait $\mathbb{Q}$ (`QQ`) encompasses all finite rational numbers, the result of dividing any integer by any nonzero integer. The trait $\mathbb{Q}^*$ (`QQ_star`) is $\mathbb{Q}$ with two extra elements, $+\infty$ and $-\infty$. The trait $\mathbb{Q}^\#$ (`QQ_splat`) is $\mathbb{Q}^*$ with one additional element, the indefinite rational (written $0/0$), which is used as the result of dividing zero by zero or of adding $-\infty$ to $+\infty$.

Often it is desirable to indicate that a variable ranges over only a subset of the rationals, such as only positive values or only nonnegative values or only nonzero values. Unfortunately, traditional notations such as $\mathbb{Q}^+$ are not used consistently in the literature; one author may use $\mathbb{Q}^+$ to mean the set of strictly positive rationals and another may use it to mean the set of nonnegative rationals. Fortress therefore uses a notation that is novel but unambiguous:

$\mathbb{Q}$ (`QQ`) is the set of rationals (it is a subtype of $\mathbb{R}$ and $\mathbb{Q}^*$).
$\mathbb{Q}_<$ (`QQ_LT`) is the set of strictly negative rationals (it is a subtype of $\mathbb{R}_<$, $\mathbb{Q}^*_<$, $\mathbb{Q}$, $\mathbb{Q}_\leq$, and $\mathbb{Q}_\neq$).
$\mathbb{Q}_\leq$ (`QQ_LE`) is the set of nonpositive rationals, that is, $\mathbb{Q}_< \cup \{0\}$ (it is a subtype of $\mathbb{R}_\leq$, $\mathbb{Q}^*_\leq$, and $\mathbb{Q}$).
$\mathbb{Q}_\geq$ (`QQ_GE`) is the set of nonnegative rationals, that is, $\mathbb{Q}_> \cup \{0\}$ (it is a subtype of $\mathbb{R}_\geq$, $\mathbb{Q}^*_\geq$, and $\mathbb{Q}$).
$\mathbb{Q}_>$ (`QQ_GT`) is the set of strictly positive rationals (it is a subtype of $\mathbb{R}_>$, $\mathbb{Q}^*_>$, $\mathbb{Q}$, $\mathbb{Q}_\geq$, and $\mathbb{Q}_\neq$).
$\mathbb{Q}_\neq$ (`QQ_NE`) is the set of strictly nonzero rationals (that is, $\mathbb{Q}_< \cup \mathbb{Q}_>$) (it is a subtype of $\mathbb{R}_\neq$, $\mathbb{Q}^*_\neq$, and $\mathbb{Q}$).
$\mathbb{Q}^*$ (`QQ_star`) is $\mathbb{Q}$ with extra elements $+\infty$ and $-\infty$ (it is a subtype of $\mathbb{R}^*$ and $\mathbb{Q}^\#$).
$\mathbb{Q}^*_<$ (`QQ_star_LT`) is $\mathbb{Q}_<$ with extra element $-\infty$ (it is a subtype of $\mathbb{R}^*_<$, $\mathbb{Q}^\#_<$, $\mathbb{Q}^*$, $\mathbb{Q}^*_\leq$, and $\mathbb{Q}^*_\neq$).
$\mathbb{Q}^*_\leq$ (`QQ_star_LE`) is $\mathbb{Q}_\leq$ with extra element $-\infty$ (it is a subtype of $\mathbb{R}^*_\leq$, $\mathbb{Q}^\#_\leq$, and $\mathbb{Q}^*$).
$\mathbb{Q}^*_\geq$ (`QQ_star_GE`) is $\mathbb{Q}_\geq$ with extra element $+\infty$ (it is a subtype of $\mathbb{R}^*_\geq$, $\mathbb{Q}^\#_\geq$, and $\mathbb{Q}^*$).
$\mathbb{Q}^*_>$ (`QQ_star_GT`) is $\mathbb{Q}_>$ with extra element $+\infty$ (it is a subtype of $\mathbb{R}^*_>$, $\mathbb{Q}^\#_>$, $\mathbb{Q}^*$, $\mathbb{Q}^*_\geq$, and $\mathbb{Q}^*_\neq$).
$\mathbb{Q}^*_\neq$ (`QQ_star_NE`) is $\mathbb{Q}_\neq$ with extra elements $+\infty$ and $-\infty$ (it is a subtype of $\mathbb{R}^*_\neq$, $\mathbb{Q}^\#_\neq$, and $\mathbb{Q}^*$).
$\mathbb{Q}^\#$ (`QQ_splat`) is $\mathbb{Q}^*$ with extra element $0/0$ (it is a subtype of $\mathbb{R}^\#$).
$\mathbb{Q}^\#_<$ (`QQ_splat_LT`) is $\mathbb{Q}^*_<$ with extra element $0/0$ (it is a subtype of $\mathbb{R}^\#_<$, $\mathbb{Q}^\#$, $\mathbb{Q}^\#_\leq$, and $\mathbb{Q}^\#_\neq$).
$\mathbb{Q}^\#_\leq$ (`QQ_splat_LE`) is $\mathbb{Q}^*_\leq$ with extra element $0/0$ (it is a subtype of $\mathbb{R}^\#_\leq$ and $\mathbb{Q}^\#$).
$\mathbb{Q}^\#_\geq$ (`QQ_splat_GE`) is $\mathbb{Q}^*_\geq$ with extra element $0/0$ (it is a subtype of $\mathbb{R}^\#_\geq$ and $\mathbb{Q}^\#$).
$\mathbb{Q}^\#_>$ (`QQ_splat_GT`) is $\mathbb{Q}^*_>$ with extra element $0/0$ (it is a subtype of $\mathbb{R}^\#_>$, $\mathbb{Q}^\#$, $\mathbb{Q}^\#_\geq$, and $\mathbb{Q}^\#_\neq$).
$\mathbb{Q}^\#_\neq$ (`QQ_splat_NE`) is $\mathbb{Q}^*_\neq$ with extra element $0/0$ (it is a subtype of $\mathbb{R}^\#_\neq$ and $\mathbb{Q}^\#$).

The Fortress type system tracks these types closely through various arithmetic operations; for example, adding two values of type $\mathbb{Q}_>$ produces a result of type $\mathbb{Q}_>$, and adding a value of type $\mathbb{Q}^*_>$ and a value of type $\mathbb{Q}_\geq$ produces a value of type $\mathbb{Q}^*_\geq$.

Here we present only the trait $\mathbb{Q}$ and its methods. The other rational types have exactly the same methods and differ only in the details of the types of method arguments and results and exactly what traits are extended by each rational type. For example, $\mathbb{Q}$ is a field and is totally ordered, $\mathbb{Q}^*$ is totally ordered but is not a field, and $\mathbb{Q}^\#$ is neither totally ordered nor a field. For the exact details of how all this is implemented, see Section 38.1.

```
trait ℚ
     extends { ℝ, ℚ*,
               Field⟦ℚ, ℚ≠, +, −, ·, /⟧,
               Field⟦ℚ, ℚ≠, +, −, ×, /⟧,
               Field⟦ℚ, ℚ≠, +, −, juxtaposition, /⟧,
               TotalOrderOperators⟦ℚ, <, ≤, ≥, >, CMP⟧ }
   coercion (_: Identity⟦+⟧) = 0
   coercion (_: Identity⟦·⟧) = 1
   coercion (_: Identity⟦×⟧) = 1
   coercion (_: Identity⟦juxtaposition⟧) = 1
   coercion (_: Zero⟦·⟧) = 0
   coercion (_: Zero⟦×⟧) = 0
   coercion (_: Zero⟦juxtaposition⟧) = 0
   opr juxtaposition (self, other: ℚ): ℚ
   opr +(self): ℚ
   opr +(self, other: ℚ): ℚ
   opr −(self): ℚ
   opr −(self, other: ℚ): ℚ
   opr ·(self, other: ℚ): ℚ
   opr ×(self, other: ℚ): ℚ
   opr /(self): ℚ*
   opr /(self, other: ℚ): ℚ#
   opr _(self, power: ℤ): ℚ#
   opr <(self, other: ℚ): Boolean
   opr ≤(self, other: ℚ): Boolean
   opr =(self, other: ℚ): Boolean
   opr ≥(self, other: ℚ): Boolean
   opr >(self, other: ℚ): Boolean
   opr CMP(self, other: ℚ*): TotalComparison
   opr CMP(self, other: ℚ#): Comparison
   opr MAX(self, other: ℚ): ℚ
   opr MIN(self, other: ℚ): ℚ
   opr MAXNUM(self, other: ℚ): ℚ
   opr MINNUM(self, other: ℚ): ℚ
   opr |self| : ℚ≥
   signum(self): ℤ
   numerator(self): ℤ
   denominator(self): ℤ
   floor(self): ℤ
   opr ⌊self⌋: ℤ
   ceiling(self): ℤ
   opr ⌈self⌉: ℤ
   round(self): ℤ
   truncate(self): ℤ
   opr ⟦self⟧: ℕ
   opr ⟦self⟧: ℕ
   opr ⟦⟦self⟧⟧: ℕ
```

```
    opr ⟦self⟧: ℕ
    realpart(self): ℚ
    imagpart(self): ℚ
    check(self): ℚ throws CastException
    check*(self): ℚ* throws CastException
    check<(self): ℚ< throws CastException
    check≤(self): ℚ≤ throws CastException
    check≥(self): ℚ≥ throws CastException
    check>(self): ℚ> throws CastException
    check≠(self): ℚ≠ throws CastException
    check*<(self): ℚ*< throws CastException
    check*≤(self): ℚ*≤ throws CastException
    check*≥(self): ℚ*≥ throws CastException
    check*>(self): ℚ*> throws CastException
    check*≠(self): ℚ*≠ throws CastException
    check#<(self): ℚ#< throws CastException
    check#≤(self): ℚ#≤ throws CastException
    check#≥(self): ℚ#≥ throws CastException
    check#>(self): ℚ#> throws CastException
    check#≠(self): ℚ#≠ throws CastException
end
```

25.1.1 opr juxtaposition (self, *other*: ℚ): ℚ

Juxtaposition of rational expressions is equivalent to using the multiplication operator $\cdot$.

25.1.2 opr +(self): ℚ

The unary addition operator $+$ simply returns its argument.

25.1.3 opr +(self, *other*: ℚ): ℚ

The binary addition operator $+$ returns the sum of its arguments.

For types $\mathbb{Q}^*$ and $\mathbb{Q}^\#$, the sum of an infinity and either a finite rational or another infinity of the same sign is equal to the given infinity, but the sum of infinities of differing sign is $0/0$, and the sum of $0/0$ and any rational value is $0/0$.

25.1.4 opr −(self): ℚ

The unary negation operator $-$ returns the negative of its argument.

For types $\mathbb{Q}^*$ and $\mathbb{Q}^\#$, the negative of $+\infty$ is $-\infty$, the negative of $-\infty$ is $+\infty$, and the negative of $0/0$ is $0/0$.

25.1.5 `opr` $-(\texttt{self}, other\colon \mathbb{Q})\colon \mathbb{Q}$

The binary subtraction operator $-$ returns the difference of its arguments, which is equal to the sum of (a) the first argument and (b) the negation of the second argument.

25.1.6 `opr` $\cdot(\texttt{self}, other\colon \mathbb{Q})\colon \mathbb{Q}$
25.1.7 `opr` $\times(\texttt{self}, other\colon \mathbb{Q})\colon \mathbb{Q}$

The multiplication operator $\cdot$ returns the product of its arguments. The multiplication operator $\times$ does exactly the same thing.

For types $\mathbb{Q}^*$ and $\mathbb{Q}^\#$, the product of $0/0$ and any rational value is $0/0$, and the product of zero and an infinity (regardless of sign) is $0/0$; the product of an infinity and any rational value other than zero and $0/0$ is an infinity whose sign is positive if and only if the two arguments have the same sign.

25.1.8 `opr` $/(\texttt{self})\colon \mathbb{Q}^*$

The unary reciprocal operator $/$ returns the reciprocal of its argument. The reciprocal of zero is $+\infty$ (and therefore the result type of $/$ when given an arguments of type $\mathbb{Q}$ is necessarily $\mathbb{Q}^*$).

For types $\mathbb{Q}^*$ and $\mathbb{Q}^\#$, the reciprocal of either $+\infty$ or $-\infty$ is zero, and the reciprocal of $0/0$ is $0/0$.

25.1.9 `opr` $/(\texttt{self}, other\colon \mathbb{Q})\colon \mathbb{Q}^*$

The binary division operator $/$ returns the quotient of its arguments, which is equal to the product of (a) the first argument and (b) the reciprocal of the second argument.

25.1.10 `opr` $\_(\texttt{self}, power\colon \mathbb{Z})\colon \mathbb{Q}^\#$

Exponentiation of a rational number to an integer power produces a rational result. If the $power$ is 0, then the result is always 1, even if the rational number base is 0 (this definition is somewhat arbitrary but is computationally useful).

`property` $\forall(x, y\colon \mathbb{Z})\ x^y = 1/(x^{-y})$
`property` $\forall(x, y\colon \mathbb{Z})\ x^y = x^{(\lfloor y/2 \rfloor)} x^{(\lceil y/2 \rceil)}$

25.1.11 `opr` $<(\texttt{self}, \mathit{other}\colon \mathbb{Q})\colon \mathrm{Boolean}$
25.1.12 `opr` $\leq(\texttt{self}, \mathit{other}\colon \mathbb{Q})\colon \mathrm{Boolean}$
25.1.13 `opr` $=(\texttt{self}, \mathit{other}\colon \mathbb{Q})\colon \mathrm{Boolean}$
25.1.14 `opr` $\geq(\texttt{self}, \mathit{other}\colon \mathbb{Q})\colon \mathrm{Boolean}$
25.1.15 `opr` $>(\texttt{self}, \mathit{other}\colon \mathbb{Q})\colon \mathrm{Boolean}$

The comparison operators $<$, $\leq$, $=$, $\geq$, and $>$ allow any rational value to be compared numerically to any other rational value.

For types $\mathbb{Q}^*$ and $\mathbb{Q}^\#$, the rational values are totally ordered except for $0/0$, which is unordered with respect to all other rational values; moreover, for compatibility with floating-point arithmetic, $0/0$ is unordered with respect to itself, and therefore these five comparison operators always return *false* if either argument is $0/0$. The value $-\infty$ is less than any finite rational value, and $+\infty$ is greater than any finite rational value.

For $\neq$ see Section 26.1.4.

25.1.16 `opr CMP`$(\texttt{self}, \mathit{other}\colon \mathbb{Q})\colon \mathrm{TotalComparison}$
25.1.17 `opr CMP`$(\texttt{self}, \mathit{other}\colon \mathbb{Q}^\#)\colon \mathrm{Comparison}$

The `CMP` operator compares the arguments and returns one of the four values $\mathrm{LessThan}$, $\mathrm{EqualTo}$, $\mathrm{GreaterThan}$, and $\mathrm{Unordered}$. If the argument types are such that the result cannot be $\mathrm{Unordered}$, then the result has type $\mathrm{TotalComparison}$ rather than simply $\mathrm{Comparison}$.

25.1.18 `opr MAX`$(\texttt{self}, \mathit{other}\colon \mathbb{Q})\colon \mathbb{Q}$
25.1.19 `opr MIN`$(\texttt{self}, \mathit{other}\colon \mathbb{Q})\colon \mathbb{Q}$
25.1.20 `opr MAXNUM`$(\texttt{self}, \mathit{other}\colon \mathbb{Q})\colon \mathbb{Q}$
25.1.21 `opr MINNUM`$(\texttt{self}, \mathit{other}\colon \mathbb{Q})\colon \mathbb{Q}$

The operators `MAX` and `MAXNUM` return whichever argument is larger in the total order defined by $<$, $\leq$, $=$, $\geq$, $>$, and `CMP`, and the operators `MIN` and `MINNUM` return whichever argument is smaller. (For all four, if the arguments are equal, then the result equals that same value.)

For type $\mathbb{Q}^\#$, `MAXNUM` and `MINNUM` differ from `MAX` and `MIN` in their treatment of $0/0$: if one argument is $0/0$ and the other is not, then `MAX` or `MIN` returns $0/0$ but `MAXNUM` or `MINNUM` returns the argument that is not $0/0$.

25.1.22 `opr` $|\texttt{self}| : \mathbb{Q}_{\geq}$

The absolute value operator $|\ldots|$ returns the negative of this rational number if the argument is less than zero, and otherwise returns the argument.

For type $\mathbb{Q}^\#$, the absolute value of $0/0$ is $0/0$.

25.1.23 $signum(\texttt{self}) : \mathbb{Z}$

The method $signum$ returns -1 if this rational number is less than zero, 0 if this rational number is zero, and 1 if this rational number is greater than zero.

For type $\mathbb{Q}^{\#}$, the signum of $0/0$ is $0/0$.

25.1.24 $numerator(\texttt{self}) : \mathbb{Z}$
25.1.25 $denominator(\texttt{self}) : \mathbb{Z}$

The method $numerator$ returns the numerator of this rational number, and the method $denominator$ returns the denominator of this rational number, when this rational number is represented in lowest terms (such that the greatest common divisor of numerator and denominator is 1).

For types $\mathbb{Q}^*$ and $\mathbb{Q}^{\#}$, the numerator of $+\infty$ is 1, the numerator of $-\infty$ is -1, and the numerator of $0/0$ is 0; the denominator of $+\infty$, $-\infty$, or $0/0$ is 0.

25.1.26 *floor*(`self`): $\mathbb{Z}$
25.1.27 `opr` $\lfloor$`self`$\rfloor$: $\mathbb{Z}$
25.1.28 *ceiling*(`self`): $\mathbb{Z}$
25.1.29 `opr` $\lceil$`self`$\rceil$: $\mathbb{Z}$
25.1.30 *round*(`self`): $\mathbb{Z}$
25.1.31 *truncate*(`self`): $\mathbb{Z}$

The method *floor*, likewise the enclosing operator $\lfloor \ldots \rfloor$, returns the largest integer that is not greater than this rational number.

The method *ceiling*, likewise the enclosing operator $\lceil \ldots \rceil$, returns the smallest integer that is not less than this rational number.

The method *round* returns the integer that is closest to this rational number, but if this rational number is exactly halfway between two consecutive integers, then *round* returns whichever of the two integers is even.

The method *truncate* returns the ceiling of this rational number if it is negative, and otherwise returns the floor of this rational number. (This has the effect of taking the floor of the magnitude, also called "rounding toward zero.")

For types $\mathbb{Q}^*$ and $\mathbb{Q}^\#$, all of these methods simply return the argument if it is $+\infty$, $-\infty$, or $0/0$.

`opr` $\llbracket$`self`$\rrbracket$: $\mathbb{N}$
`opr` $\lceil\!\lceil$`self`$\rceil\!\rceil$: $\mathbb{N}$
`opr` $\lfloor\!\lfloor\!\lfloor$`self`$\rfloor\!\rfloor\!\rfloor$: $\mathbb{N}$
`opr` $\lceil\!\lceil\!\lceil$`self`$\rceil\!\rceil\!\rceil$: $\mathbb{N}$

The hyperfloor operation $\llbracket x \rrbracket$ computes $2^{(\lfloor log_2 x \rfloor)}$ and returns the result as a natural number. If the argument is equal to 0, the result is 0. If the argument is negative, an $\mathrm{InvalidArgumentException}$ is thrown.

The hyperceiling operation $\lceil\!\lceil x \rceil\!\rceil$ computes $2^{(\lceil log_2 x \rceil)}$ and returns the result as a natural number. If the argument is equal to 0, the result is 0. If the argument is negative, an $\mathrm{InvalidArgumentException}$ is thrown.

The hyperhyperfloor operation $\lfloor\!\lfloor\!\lfloor x \rfloor\!\rfloor\!\rfloor$ computes $2^{(\llbracket log_2 x \rrbracket)}$ and returns the result as a natural number. If the argument is equal to 0 or 1, the result is the same as the argument. If the argument is negative, an $\mathrm{InvalidArgumentException}$ is thrown.

The hyperhyperceiling operation $\lceil\!\lceil\!\lceil x \rceil\!\rceil\!\rceil$ computes $2^{(\lceil\!\lceil log_2 x \rceil\!\rceil)}$ and returns the result as a natural number. If the argument is equal to 0 or 1, the result is the same as the argument. If the argument is negative, an $\mathrm{InvalidArgumentException}$ is thrown.

25.1.32 *realpart*(`self`): $\mathbb{Q}$

The method *realpart* for a rational number simply returns its argument.

25.1.33 *imagpart*(`self`): $\mathbb{Q}$

The method *imagpart* for a rational number simply returns zero.

25.1.34 $check(\texttt{self}) \colon \mathbb{Q}$ `throws` CastException
25.1.35 $check^{*}(\texttt{self}) \colon \mathbb{Q}^{*}$ `throws` CastException
25.1.36 $check_{<}(\texttt{self}) \colon \mathbb{Q}_{<}$ `throws` CastException
25.1.37 $check_{\leq}(\texttt{self}) \colon \mathbb{Q}_{\leq}$ `throws` CastException
25.1.38 $check_{\geq}(\texttt{self}) \colon \mathbb{Q}_{\geq}$ `throws` CastException
25.1.39 $check_{>}(\texttt{self}) \colon \mathbb{Q}_{>}$ `throws` CastException
25.1.40 $check_{\neq}(\texttt{self}) \colon \mathbb{Q}_{\neq}$ `throws` CastException
25.1.41 $check^{*}_{<}(\texttt{self}) \colon \mathbb{Q}^{*}_{<}$ `throws` CastException
25.1.42 $check^{*}_{\leq}(\texttt{self}) \colon \mathbb{Q}^{*}_{\leq}$ `throws` CastException
25.1.43 $check^{*}_{\geq}(\texttt{self}) \colon \mathbb{Q}^{*}_{\geq}$ `throws` CastException
25.1.44 $check^{*}_{>}(\texttt{self}) \colon \mathbb{Q}^{*}_{>}$ `throws` CastException
25.1.45 $check^{*}_{\neq}(\texttt{self}) \colon \mathbb{Q}^{*}_{\neq}$ `throws` CastException
25.1.46 $check^{\#}_{<}(\texttt{self}) \colon \mathbb{Q}^{\#}_{<}$ `throws` CastException
25.1.47 $check^{\#}_{\leq}(\texttt{self}) \colon \mathbb{Q}^{\#}_{\leq}$ `throws` CastException
25.1.48 $check^{\#}_{\geq}(\texttt{self}) \colon \mathbb{Q}^{\#}_{\geq}$ `throws` CastException
25.1.49 $check^{\#}_{>}(\texttt{self}) \colon \mathbb{Q}^{\#}_{>}$ `throws` CastException
25.1.50 $check^{\#}_{\neq}(\texttt{self}) \colon \mathbb{Q}^{\#}_{\neq}$ `throws` CastException

Each of these methods checks this rational number to see whether it belongs to the result type of the method. If, the number is returned; if not, a CastException is thrown.

Chapter 26

Negated Relational Operators

26.1 Negated Relational Operators

26.1.1 `opr` $\not\equiv(x\colon \mathrm{Object}, y\colon \mathrm{Object})\colon \mathrm{Boolean}$
26.1.2 `opr` $\not\equiv[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \equiv]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.3 `opr` $\not\equiv[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \equiv]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.4 `opr` $\neq[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, =]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.5 `opr` $\neq[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, =]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.6 `opr` $\not\simeq[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \simeq]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.7 `opr` $\not\simeq[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \simeq]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.8 `opr` $\not\approx[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \approx]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.9 `opr` $\not\approx[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \approx]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$

The infix operator $\not\equiv$ applies $\neg$ to the result of $\equiv$ on the same operands.

The infix operator $\not\equiv$ applies $\neg$ to the result of $\equiv$ on the same operands.

The infix operator $\neq$ applies $\neg$ to the result of $=$ on the same operands.

The infix operator $\not\simeq$ applies $\neg$ to the result of $\simeq$ on the same operands.

The infix operator $\not\approx$ applies $\neg$ to the result of $\approx$ on the same operands.

26.1.10 `opr` $\not<[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, <]\!](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.11 `opr` $\not<[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, <]\!](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.12 `opr` $\not\leq[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \leq]\!](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.13 `opr` $\not\leq[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \leq]\!](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.14 `opr` $\not\geq[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \geq]\!](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.15 `opr` $\not\geq[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \geq]\!](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.16 `opr` $\not>[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, >]\!](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.17 `opr` $\not>[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, >]\!](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$

The infix operator $\not<$ applies $\neg$ to the result of $<$ on the same operands.

The infix operator $\not\leq$ applies $\neg$ to the result of $\leq$ on the same operands.

The infix operator $\not\geq$ applies $\neg$ to the result of $\geq$ on the same operands.

The infix operator $\not>$ applies $\neg$ to the result of $>$ on the same operands.

26.1.18 `opr` $\not\subset[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \subset]\!](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.19 `opr` $\not\subset[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \subset]\!](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.20 `opr` $\not\subseteq[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \subseteq]\!](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.21 `opr` $\not\subseteq[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \subseteq]\!](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.22 `opr` $\not\supseteq[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \supseteq]\!](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.23 `opr` $\not\supseteq[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \supseteq]\!](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.24 `opr` $\not\supset[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \supset]\!](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.25 `opr` $\not\supset[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \supset]\!](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$

The infix operator $\not\subset$ applies $\neg$ to the result of $\subset$ on the same operands.

The infix operator $\not\subseteq$ applies $\neg$ to the result of $\subseteq$ on the same operands.

The infix operator $\not\supseteq$ applies $\neg$ to the result of $\supseteq$ on the same operands.

The infix operator $\not\supset$ applies $\neg$ to the result of $\supset$ on the same operands.

26.1.26 `opr` $\not\prec[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \prec]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.27 `opr` $\not\prec[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \prec]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.28 `opr` $\not\preceq[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \preccurlyeq]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.29 `opr` $\not\preceq[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \preccurlyeq]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.30 `opr` $\not\succeq[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \succcurlyeq]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.31 `opr` $\not\succeq[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \succcurlyeq]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.32 `opr` $\not\succ[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \succ]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.33 `opr` $\not\succ[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \succ]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$

The infix operator $\not\prec$ applies $\neg$ to the result of $\prec$ on the same operands.

The infix operator $\not\preceq$ applies $\neg$ to the result of $\preceq$ on the same operands.

The infix operator $\not\succeq$ applies $\neg$ to the result of $\succeq$ on the same operands.

The infix operator $\not\succ$ applies $\neg$ to the result of $\succ$ on the same operands.

26.1.34 `opr` $\not\sqsubset[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \sqsubset]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.35 `opr` $\not\sqsubset[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \sqsubset]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.36 `opr` $\not\sqsubseteq[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \sqsubseteq]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.37 `opr` $\not\sqsubseteq[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \sqsubseteq]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.38 `opr` $\not\sqsupseteq[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \sqsupseteq]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.39 `opr` $\not\sqsupseteq[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \sqsupseteq]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.40 `opr` $\not\sqsupset[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \sqsupset]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.41 `opr` $\not\sqsupset[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \sqsupset]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$

The infix operator $\not\sqsubset$ applies $\neg$ to the result of $\sqsubset$ on the same operands.

The infix operator $\not\sqsubseteq$ applies $\neg$ to the result of $\sqsubseteq$ on the same operands.

The infix operator $\not\sqsupseteq$ applies $\neg$ to the result of $\sqsupseteq$ on the same operands.

The infix operator $\not\sqsupset$ applies $\neg$ to the result of $\sqsupset$ on the same operands.

26.1.42 `opr` $\notin[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \in]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.43 `opr` $\notin[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \in]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$
26.1.44 `opr` $\not\ni[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \ni]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.45 `opr` $\not\ni[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \ni]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$

The infix operator $\notin$ applies $\neg$ to the result of $\in$ on the same operands.

The infix operator $\not\ni$ applies $\neg$ to the result of $\ni$ on the same operands.

26.1.46 `opr` $\nparallel[\![T$ `extends` $\mathrm{BinaryPredicate}[\![T, \|]\!]](x\colon T, y\colon T)\colon \mathrm{Boolean}$
26.1.47 `opr` $\nparallel[\![T$ `extends` $\mathrm{BinaryIntervalPredicate}[\![T, \|]\!]](x\colon T, y\colon T)\colon \mathrm{BooleanInterval}$

The infix operator $\nparallel$ applies $\neg$ to the result of $\|$ on the same operands.

Chapter 27

Exceptions

27.1 The Trait Fortress.Standard.Exception

The trait Exception is a single root of the exception hierarchy; every exception in Fortress has trait Exception. An exception is either a CheckedException or an UncheckException. Every exception has optional fields: a message and a chained exception. These fields are default to Nothing where an optional value v is either Nothing or Just(v) as declared in Section 31.2.

```
trait Exception comprises { CheckedException, UncheckedException }
  settable message: Maybe⟦String⟧
  settable chain: Maybe⟦Exception⟧
end
```

27.1.1 `settable` *message*: Maybe⟦String⟧

When an exception is thrown, its *message* may be set.

27.1.2 `settable` *chain*: Maybe⟦Exception⟧

When an exception is thrown, its *chain* may be set to the exception thrown immediately before this exception.

27.2 The Trait Fortress.Standard.CheckedException

```
trait CheckedException
  extends { Exception }
  excludes { UncheckedException }
end
```

27.3 The Trait Fortress.Standard.UncheckedException

```
trait UncheckedException
  extends { Exception }
  excludes { CheckedException }
end
```

Chapter 28

Threads

28.1 The Trait Fortress.Standard.Thread

Every thread in Fortress has trait Thread.

```
trait Thread
   val⟦T⟧(): T
   wait(): ()
   ready(): Boolean
   stop(): () throws Stopped
end
```

28.1.1 $val[\![T]\!](): T$

The val method returns the value computed by the expression of the thread. If the thread has not yet completed execution, the invocation of val blocks until it has done so.

28.1.2 $wait(): ()$

The $wait$ method waits for a thread to complete, but does not return a value.

28.1.3 $ready(): \mathrm{Boolean}$

The $ready$ method returns $true$ if a thread has completed, and returns $false$ otherwise.

28.1.4 $stop(): ()$ `throws` $\mathrm{Stopped}$

The $stop$ method attempts to terminate a thread.

Chapter 29

Dimensions and Units

29.1 Fortress.SIUnits

(∗ Reference: http://physics.nist.gov/cuu/Units/index.html ∗)

(∗ SI base units ∗)

`dim` Length `SI_unit` meter meters m
`dim` Mass `default` kilogram; `SI_unit` gram grams g: Mass
`dim` Time `SI_unit` second seconds s
`dim` ElectricCurrent `SI_unit` ampere amperes A
`dim` Temperature `SI_unit` kelvin kelvins K
`dim` AmountOfSubstance `SI_unit` mole moles mol
`dim` LuminousIntensity `SI_unit` candela candelas cd

(∗ SI derived units with special names and symbols ∗)

`dim` Angle = Unity `SI_unit` radian radians rad
`dim` SolidAngle = Unity `SI_unit` steradian steradians sr
`dim` Frequency = 1/Time `SI_unit` hertz Hz
`dim` Force = Mass Acceleration `SI_unit` newton newtons N
`dim` Pressure = Force/Area `SI_unit` pascal pascals Pa
`dim` Energy = Length Force `SI_unit` joule joules J
`dim` Power = Energy/Time `SI_unit` watt watts W
`dim` ElectricCharge = ElectricCurrent Time `SI_unit` coulomb coulombs C
`dim` ElectricPotential = Power/Current `SI_unit` volt volts V
`dim` Capacitance = ElectricCharge/Voltage `SI_unit` farad farads F
`dim` Resistance = ElectricPotential/Current `SI_unit` ohm ohms Ω
`dim` Conductance = 1/Resistance `SI_unit` siemens S
`dim` MagneticFlux = Voltage Time `SI_unit` weber webers Wb
`dim` MagneticFluxDensity = MagneticFlux/Area `SI_unit` tesla teslas T
`dim` Inductance = MagneticFlux/Current `SI_unit` henry henries H
`dim` LuminousFlux = LuminousIntensity SolidAngle `SI_unit` lumen lumens lm
`dim` Illuminance = LuminousFlux/Area `SI_unit` lux lx
`dim` RadionuclideActivity = 1/Time `SI_unit` becquerel becquerels Bq
`dim` AbsorbedDose = Energy/Mass `SI_unit` gray grays Gy
`dim` CatalyticActivity = AmountOfSubstance/Time `SI_unit` katal katals kat

(∗ other derived dimensions ∗)

```
dim Area = Length^2
dim Volume = Length^3
dim Velocity = Length/Time
dim Speed = Velocity
dim Acceleration = Velocity/Time
dim Momentum = Mass Velocity
dim AngularVelocity = Angle/Second
dim AngularAcceleration = Angle/Second^2
dim WaveNumber = 1/Length
dim MassDensity = Mass/Volume
dim CurrentDensity = Current/Area
dim MagneticFieldStrength = Current/Length
dim Luminance = LuminousIntensity/Area
dim Work = Energy
dim Action = Energy Time
dim MomentOfForce = Force Length
dim Torque = MomentOfForce
dim MomentOfInertia = Mass Length^2
dim Voltage = ElectricPotential
dim Conductivity = Conductance/Length
dim Resistivity = 1/Conductivity
dim Impedance = Resistance
dim Permittivity = Capacitance/Length
dim Permeability = Inductance/Length
dim Irradiance = Power/Area
dim RadiantIntensity = Power/SolidAngle
dim Radiance = Power/Area SolidAngle
dim AbsorbedDoseRate = AbsorbedDose/Time
dim CatalyticConcentration = CatalyticActivity/Volume
dim HeatCapacity = Energy/Temperature
dim Entropy = Energy/Temperature
dim DynamicViscosity = Pressure Time
dim SpecificHeatCapacity = Energy/Mass Temperature
dim SpecificEntropy = Energy/Mass Temperature
dim SpecificEnergy = Energy/Mass
dim ThermalConductivity = Energy/Length Temperature
dim EnergyDensity = Energy/Volume
dim ElectricFieldStrength = ElectricPotential/Length
dim ElectricChargeDensity = ElectricCharge/Volume
dim ElectricFlux = ElectricCharge
dim ElectricFluxDensity = ElectricCharge/Area
dim MolarEnergy = Energy/AmountOfSubstance
dim MolarHeatCapacity = Energy/AmountOfSubstance Temperature
dim MolarEntropy = Energy/AmountOfSubstance Temperature
dim RadiationExposure = ElectricCharge/Mass

(* Units outside the SI that are accepted for use with the SI *)

unit minute minutes min: Time
unit hour hours h: Time
unit day days d: Time
unit degreeOfAngle degrees: Angle
unit minuteOfAngle minutesOfAngle: Angle
```

```
unit secondOfAngle secondsOfAngle: Angle
SI_unit metricTon metricTons tonne tonnes t: Mass
SI_unit liter liters L: Volume
```

29.2 Fortress.EnglishUnits

```
import { Length, Area, Volume, Time, Mass, millimeters, liters, grams }
       from Fortress.SIUnits

unit inch inches: Length
unit foot feet: Length
unit yard yards: Length
unit mile miles: Length
unit rod rods: Length
unit furlong furlongs: Length

unit surveyFoot surveyFeet: Length
unit surveyMile surveyMiles: Length

unit nauticalMile nauticalMiles: Length
unit knot knots: Speed

unit week weeks: Time
unit fortnight fortnights: Time
unit microfortnight microfortnights

unit gallon gallons: Volume
unit fluidQuart fluidQuarts: Volume
unit fluidPint fluidPints: Volume
unit fluidCup fluidCups: Volume
Unit fluidOunce fluidOunces: Volume
unit fluidDram fluidDrams: Volume
unit minim minims: Volume

unit traditionalTablespoon traditionalTablespoons: Volume
unit traditionalTeaspoon traditionalTeaspoons: Volume
unit federalTablespoon federalTablespoons: Volume
unit federalTeaspoon federalTeaspoons: Volume

unit dryPint dryPints: Volume
unit dryQuart dryQuarts: Volume
unit peck pecks: Volume
unit bushel bushels: Volume

unit acre: Area

unit imperialGallon: Volume
unit imperialQuart: Volume
unit imperialPint: Volume
unit imperialGill: Volume
unit imperialFluidOunce: Volume
unit imperialFluidDrachm: Volume
unit imperialFluidDRam : Volume
unit imperialFluidScruple: Volume
unit imperialMinim: Volume

unit pound pounds lb lbs: Mass
```

`unit` ounce ounces oz: Mass
`unit` grain grains: Mass
`unit` troyPound troyPounds: Mass
`unit` troyOunce troyOunces: Mass

29.3 Fortress.InformationUnits

`dim` Information `unit` bit bits
`unit` byte bytes

Chapter 30

Tests

30.1 The Object Fortress.Standard.TestSuite

An instance of the object $\mathrm{TestSuite}$ contains a set of test functions that can all be called by invoking the method run:

```
test object TestSuite(testFunctions = {})
  add(f: () → ()): ()
  run(): ()
end
```

30.1.1 $add(f\colon () \to ())\colon ()$

The add method adds a given test function to the $testFunctions$ field of this object.

30.1.2 $run()\colon ()$

The run method calls each test function in the $testFunctions$ field of this object. Note that all tests in a $\mathrm{TestSuite}$ are run in parallel.

30.2 Test Functions

30.2.1 `test` $fail(message\colon \mathrm{String})\colon ()$

The helper function $fail$ displays the error message provided and terminates execution of the enclosing test.

Chapter 31

Convenience Functions and Types

31.1 Convenience Functions

31.1.1 $cast[\![T]\!](x : \mathrm{Object}) : T$

The function $cast$ converts the type of its argument to a given type. If the static type of the argument is not a subtype of the given type, a $\mathrm{CastException}$ is thrown.

31.1.2 $instanceOf[\![T]\!](x : \mathrm{Object}) : \mathrm{Boolean}$

The function $instanceOf$ tests whether its argument has a given type and returns a boolean value.

31.1.3 $ignore(x : \mathrm{Object}) : ()$

The function $ignore$ discards the value of its argument and returns $()$.

31.1.4 $tuple(x : \mathrm{Object}) : \mathrm{Object}$

The function $tuple$ returns its argument as a tuple expression.

31.1.5 $identity(x : \mathrm{Object}) : \mathrm{Object}$

The function $identity$ returns its argument.

31.2 Convenience Types

An optional value v is either $\mathrm{Nothing}$ or $\mathrm{Just}(v)$ declared as follows:

```
(∗ Optional Values ∗)
trait Maybe⟦T⟧ comprises { Nothing, Just⟦T⟧ }
   isNothing: Boolean
end
object Nothing extends Maybe⟦T⟧ excludes Just⟦T⟧ where {T extends Object}
end
object Just⟦T⟧(just: T) extends Maybe⟦T⟧
end
```

Part IV

Fortress for Library Writers

Chapter 32

Parallelism and Locality

Fortress is designed to make parallel programming as simple and as painless as possible. This chapter describes the internals of Fortress parallelism designed for use by authors of library code (such as *distributions*, generators, and arrays). We adopt a multi-tiered approach to parallelism:

- At the highest level, we provide libraries which allocate locality-aware distributed arrays (Section 32.2) and implicitly parallel constructs such as tuples and loops. Synchronization is accomplished through the use of atomic sections (Section 13.23). More complex synchronization makes use of abortable atomicity, described in Section 32.3.
- There is an extensive library of distributions, which permits the programmer to specify locality and data distribution explicitly (Section 32.5).
- Immediately below that, the `at` expression requests that a computation take place in a particular region of the machine (Section 32.7). We also provide a mechanism to terminate a spawned thread early (Section 32.6).
- Finally, there are mechanisms for constructing new generators via recursive subdivision into tree structures with individual elements at the leaves. Section 32.8 explains how iterative constructs such as `for` loops and comprehensions are desugared into calls to methods of trait $\mathrm{Generator}$, and how new instances of this trait may be defined.

We begin by describing the abstraction of *regions*, which Fortress uses to describe the machine on which a program is run.

32.1 Regions

Every thread and every object in Fortress, and every element of a Fortress array, has an associated *region*. The region in which an object o resides can be obtained by calling $o.region$. Regions abstractly describe the structure of the machine on which a Fortress program is running. They are organized hierarchically to form a tree, the *region hierarchy*, reflecting in an abstract way the degree of locality which those regions share. The different levels of this tree reflect underlying machine structure, such as execution engines within a CPU, memory shared by a group of processors, or resources distributed across the entire machine. Objects which reside in regions near the leaves of the tree are local entities; those which reside at higher levels of the region tree are logically spread out. The method call $r.isLocalTo(s)$ returns $true$ if r is contained within the region tree rooted at s.

It is important to understand that regions and the structures (such as distributions, Section 32.5) built on top of them exist purely for performance purposes. The placement of a thread or an object does not have any semantic effect on

the meaning of a program; it is simply an aid to enable the implementation to make informed decisions about data placement.

It may not be possible for an object or a thread to reside in any possible region. The *execution level* of the region hierarchy is where threads of execution reside, and is generally the bottommost level in the region tree. A thread is generally associated with some region at the execution level, where that spawned thread will preferentially be run. The programmer can affect the choice of region by using an `at` expression (Section 32.7) when the thread is created. A spawned thread may be assigned a region higher in the region hierarchy than the execution level, either because a higher region was requested or because scheduling decisions permit the thread to run in several possible execution regions. The region to which a thread is assigned may also change over time due to scheduling decisions. The $region$ method for the object associated with a spawned thread returns the region of the associated thread.

The *memory level* of the region hierarchy is where individual reference objects reside; on a machine with nodes composed of multiple processor cores sharing a single memory, this will not generally be the leaves of the region hierarchy. Imagine a constructor for a reference object is called by a thread residing in region r, yielding an object o. Except in very rare circumstances (for example when a local node is out of memory) either $r.isLocalTo(o.region)$ or $(o.region).isLocalTo(r)$ ought to hold: data is allocated locally to the thread which runs the constructor. For a value object v being manipulated by a thread residing in region r either $(v.region).isLocalTo(r)$ or $r.isLocalTo(v.region)$ (value objects always appear to be local).

Note that $region$ is a getter method and can be overridden like any other method. The chief example of this is arrays, which are generally composed from many reference objects; the $region$ method is overridden to return the location of the array as a whole—the region which contains all of its constituent reference objects.

32.2 Distributed Arrays

Arrays, vectors, and matrices in Fortress are assumed to be spread out across the machine. As in Fortran, Fortress arrays are complex data structures; simple linear storage is encapsulated by the $\mathrm{HeapSequence}$ type, which is used in the implementation of arrays (see Section 32.7). The default distribution of an array is determined by the Fortress libraries; in general it depends on the size of the array, and on the size and locality characteristics of the machine running the program. For advanced users, the distribution library (introduced in Section 32.5) provides a way of combining and pivoting distributions, or of redistributing two arrays so that their distributions match. Programmers should create arrays by using an array comprehension (Section 13.29) or an aggregate expression (Section 13.28). The operational internals of array comprehensions are described in Section 32.8.

Because the elements of a fortress array may reside in multiple regions of the machine, there is an additional method $a.region(i)$ which returns the region in which the array element a_i resides. An element of an array is always local to the region in which the array as a whole is contained, so $(a.region(i)).isLocalTo(a.region)$ must always return $true$. When an array contains reference objects, the programmer must be careful to distinguish the region in which the array element a_i resides, $a.region(i)$, from the region in which the object referred to by the array element resides, $a_i.region$. The former describes the region of the array itself; the latter describes the region of the data referred to by the array. These may differ.

32.3 Abortable Atomicity

Fortress provides a user-level $abort()$ function which abandons execution of an `atomic` expression and rolls back its changes, requiring the `atomic` expression to execute again from the beginning. This permits an atomic section to perform consistency checks as it runs. However, the functionality provided by $abort()$ can be abused; it is possible to induce deadlock or livelock by creating an atomic section which always fails. Here is a simple example of a program using $abort()$ which is incorrect because Fortress does not guarantee that the two implicit threads (created

by evaluating the two elements of the tuple) will always run in parallel; it is possible for the first element of the tuple to continually abort without ever running the second element of the tuple:

```
r : ℤ64 := 0
(a, b) = (atomic if r = 1 then 17 else abort() end,
          do r := 1; r end)(* INCORRECT! *)
```

Fortress also includes a `tryatomic` expression, which attempts to run its body expression atomically. If it succeeds, the result is returned; if it aborts due to a call to *abort*, the AtomicAborted exception is thrown; if it aborts due to conflict (as described in Section 13.23), the AtomicConflict exception is thrown. These exceptions both implement the AtomicFailed trait, which is an instance of CheckedException. Conceptually `atomic` can be defined in terms of `tryatomic` as follows:

```
label AtomicBlock
  while true do
    try
      result = tryatomic body
      exit AtomicBlock with result
    catch e
      AtomicFailed ⇒ ()(* continue execution *)
    end
  end
  throw UnreachableCode(* inserted for type correctness *)
end AtomicBlock
```

Unlike the above definition, an implementation may choose to suspend a thread running an `atomic` expression which invokes *abort*, re-starting it at a later time when it may be possible to make further progress. The above definition restarts the body of the `atomic` expression immediately without suspending.

32.4 Shared and Local Data

Every object in a Fortress program is considered to be either *shared* or *local* (collectively referred to as the *sharedness* of the object). A local object must be transitively reachable (through zero or more object references) from the variables of at most one running thread. A local object may be accessed more cheaply than a shared object, particularly in the case of atomic reads and writes. Sharedness is ordinarily managed implicitly by the Fortress implementation. Control over sharedness is intended to be a performance optimization; however, methods such as *isShared* and *localize* can affect program semantics, and must be used with care.

The sharedness of an object should be contrasted with its region. The region of an object describes where that object is located on the machine. The sharedness of an object describes whether the object is visible to one thread or to many. A local object need not actually reside in a region near the thread to which it is visible (though ordinarily it will).

The following rules govern sharedness:

- Reference objects are initially local when they are constructed.
- The sharedness of an object may change as the program executes.
- If an object is currently transitively reachable from more than one running thread, it must be shared.
- When a reference to a local object is stored into a field of a shared object, the local object must be *published*: Its sharedness is changed to shared, and all of the data to which it refers is also published.
- The value of a local variable referenced by a thread must be published before that thread may be run in parallel with the thread which created it. Values assigned to the variable while the threads run in parallel must also be published.

- A field with value type is assigned by copying, and thus has the sharedness of the containing object or closure.

Publishing can be expensive, particularly if the structure being broadcast is large and heavily nested; this can cause an apparently short `atomic` expression (a single write, say) to run arbitrarily long. To avoid this, the library programmer can request that an object be published by calling the semantically transparent function *shared*:

$x := \mathit{shared}\ \mathrm{Cons}(x, xs)$
$\mathit{shared}(y)$

A local copy of an object can be obtained by calling *copy*, a method on trait Object:

$\mathit{localVar} := \mathit{sharedVar}.\mathit{copy}()$

Two additional methods are provided which permit different choices of program behavior based on the sharedness of objects:

- The getter $o.\mathit{isShared}$ returns *true* when o is shared, and *false* when it is local. This permits the program to take different actions based on sharedness.
- Method $o.\mathit{localize}()$ is equivalent to the following expression:

  `if` $o.\mathit{isShared}$ `then` $o.\mathit{copy}()$ `else` o `end`

These methods must be used with extreme caution. For example, *localize* should be used only when there is a unique reference to the object being localized. The *localize* method can have unexpected behavior if there is a reference to o from another local object p. Updates to o will be visible through p; subsequent publication of p will publish o. By contrast, if o was already shared, and referred to by another shared object, the newly-localized copy will be entirely distinct; changes to the copy will not be visible through p, and publishing p will not affect the locality of the copy.

32.5 Distributions

Most of the heavy lifting in mapping threads and arrays to regions is performed by *distributions*. An instance of the trait Distribution describes the parallel structure of ranges and other numeric generators (such as the generators for the index space of an array), and provides for the allocation and distribution of arrays on the machine:

```
trait Distribution extends Object
  distribute⟦T extends ArrayIndex⟧(Range⟦T⟧) : Range⟦T⟧
  distribute⟦E, B extends ArrayIndex⟧(a : Array⟦E, B⟧) : Array⟦E, B⟧ =
    distributeFromTo⟦E, B⟧(a, a.distribution, self)
end
```

Abstractly, a Distribution acts as a transducer for generators and arrays. The *distribute* method applied to a multidimensional Range organizes its indices into the leaves of a tree whose inner nodes correspond to potential levels of parallelism and locality in the underlying computation, producing a fresh Range whose behavior as a Generator may differ from that of the passed-in Range. The *distribute* method applied to an array creates a copy of that array distributed according to the given distribution. This is specified in terms of a call to the overloaded function *distributeFromTo*. This permits the definition of specialized versions of this function for particular pairs of distributions.

The intention of distributions is to separate the task of data distribution and program correctness. That is, it should be possible to write and debug a perfectly acceptable parallel program using only the default data distribution provided by the system. Imposing a distribution on particular computations, or designing and implementing distributions from scratch, is a task best left for performance tuning, and one which should not affect the correctness of a working program.

There is a DefaultDistribution which is defined by the underlying system. This distribution is designed to be reasonably adaptable to different system scales and architectures, at the cost of some runtime efficiency. Arrays and generators that are not explicitly allocated through a distribution are given the DefaultDistribution.

We said in Section 13.15 that there is a generator, $indices$, associated with every array. This generator is distributed in the same way as the array itself. When we re-distribute an array, we also re-distribute the generator; thus $d.distribute(a.indices)$ is equivalent to $(d.distribute(a)).indices$.

There are a number of built-in distributions:

DefaultDistribution	Name for distribution chosen by system.
Sequential	Sequential distribution. Arrays are allocated in one contiguous piece of memory.
Local	Equivalent to Sequential.
Par	Blocked into chunks of size 1.
Blocked	Blocked into roughly equal chunks.
Blocked(n)	Blocked into n roughly equal chunks.
Subdivided	Chopped into 2^k-sized chunks, recursively.
Interleaved($d_1, d_2, \ldots d_n$)	The first n dimensions are distributed according to $d_1 \ldots d_n$, with subdivision alternating among dimensions.
Joined($d_1, d_2, \ldots d_n$)	The first n dimensions are distributed according to $d_1 \ldots d_n$, subdividing completely in each dimension before proceeding to the next.

From these, a number of composed distributions are provided:

Morton	Bit-interleaved Morton order [20], recursive subdivision in all dimensions.
Blocked($x_1, x_2, \ldots x_n$)	Blocked in n dimensions into chunks of size x_i in dimension i; remaining dimensions (if any) are local.

To allocate an array which is local to a single thread (and most likely allocated in contiguous storage), the Local distribution can be used:

$a = \text{Local}.distribute[\,1\ 0\ 0; 0\ 1\ 0; 0\ 0\ 1\,]$

Other distributions can be requested in a similar way.

Distributions can be constructed and given names:

$spatialDist = \text{Blocked}(n, n, 1)(*$ Pencils along the z axis $*)$

The system will lay out arrays with the same distribution in the same way in memory (as much as this is feasible), and will run loops with the same distribution in the same way (as much as this is feasible). By contrast, if we replace every occurrence of $spatialDist$ by Blocked$(n, n, 1)$, this code will likely divide up arrays and ranges into the same-sized pieces as above, but these pieces need not be collocated.

32.6 Early Termination of Threads

As noted in Section 4.4, an implicit thread can be terminated if its group is going to throw an exception. Similarly, a spawned thread t may be terminated by calling $t.stop()$. A successful attempt to terminate a thread causes the thread to complete asynchronously. There is no guarantee that termination attempts will be prompt, or that they will occur at all; the implementation will make its best effort. If a thread completes normally or exceptionally before an attempt to terminate it succeeds, the result is retained and the termination attempt is simply dropped.

A termination attempt acts as if a special hidden *stop exception* is thrown in that thread. This exception cannot be thrown by `throw` or caught by `catch`; however, `finally` clauses are run as with any other exception. If the stopped thread was in the middle of an `atomic` expression, the effects of that expression are rolled back just as with an ordinary `throw`. A special wrapper around every spawned thread is provided by the Fortress implementation; it catches the

stop exception and transforms it into a deferred $\mathrm{Stopped}$ exception. This is visible to the programmer and should be caught by invoking the val method on the thread object. Implicit threads are terminated only if another thread in the group completes abruptly, and the threads that are terminated are ignored for the purposes of the completion of the group.

Typical code for stopping a thread looks something like the following example:

```
x : ℤ64 := 0
t = spawn do
          try
            atomic if x = 0 then abort() else () end
          finally
            x := 1
          end
    end
t.stop()
try
   t.val()
catch s
   Stopped ⇒ x += 2; x
end
```

Here the spawned thread t blocks until it is killed by the call to $t.stop()$; it sets x to 1 in the `finally` clause before exiting. In this case, the call to $t.val()$ will throw $\mathrm{Stopped}$, which is caught, causing 2 to be added to x and returning 3.

Note that there is a race in the above code, so the `try` block in t may not have been entered when $t.stop()$ is called, causing x to be 2 at the end of execution. Note also that the call to $t.stop()$ occurs asynchronously; in the absence of the call to $t.val()$, the spawning thread would not have waited for t to complete.

32.7 Placing Threads

A thread can be placed in a particular region by using an `at` expression:

```
(v, w) = (a_i,
          at a.region(j) do
             a_j
          end)
```

In this example, two implicit threads are created; the first computes a_i locally, the second computes a_j in the region where the j^{th} element of a resides, specified by $a.region(j)$. The expression after `at` must return a value of type Region, and the block immediately following `do` is run in that region; the result of the block is the result of the at expression as a whole. Often it is more graceful to use the `also do` construct (described in Section 13.12) in these cases:

```
do
   v := a_i
also at a.region(j) do
   w := a_j
end
```

We can also use `at` with a `spawn` expression:

```
v = spawn at a.region(i) do
        a_i
    end
w = spawn at v.region() do
        v.val() · 17
    end
```

Finally, note that it is possible to use an `at` expression within a block:

```
do
  v := a_i
  at a.region(j) do
    w := a_j
  end
  x = v + w
end
```

We can think of this as the degenerate case of `also do`: a thread group is created with a single implicit thread running the contents of the `at` expression in the given region; when this thread completes control returns to the original location.

Note that the regions given in an `at` expression are non-binding: the Fortress implementation may choose to run the computations elsewhere—for example, thread migration might not be possible within an `atomic` expression, or load balancing might cause code to be executed in a different region. In general, however, implementations should attempt to respect thread placement annotations when they are given.

32.8 Use and Definition of Generators

Several expressions in Fortress make use of *generator lists* (given by the nonterminal *GeneratorList* in the Fortress grammar defined in Appendix G) to express parallel iteration (see Section 13.17). A generator list binds a series of variables to the values produced by a series of objects with the Generator trait. A generator list is simply syntactic sugar for a nested series of invocations of methods on these objects. *All the parallelism provided by a particular generator is specified by its definitions for the methods of the* Generator *trait.* In general, the library code for a generator dictates the parallel structure of computations involving that generator.

The definition of trait Generator has very simple functionality at its core:

```
trait Generator⟦E⟧
  size : ℤ64
  generate⟦R extends Monoid⟦R, ⊕⟧, opr ⊕⟧(body : E → R) : R
  join⟦N⟧(other : Generator⟦N⟧) : Generator⟦(E, N)⟧ =
    SimplePairGenerator⟦E, N⟧(self, other)
end
```

The mechanics of object generation are embedded entirely in the *generate* method. This method takes one argument, the *body* function. The *generate* method invokes *body* once for each object which is to be generated, passing the generated object as an argument. Note that *body* returns a value in some Monoid R; the results of the calls to *body* are combined using the monoid operator $\oplus$. This *reduction* may include any number of occurrences of the identity of the monoid—in particular, a generator may generate no elements, in which case it will never invoke *body* and will simply the identity.

A simple definition of a Generator need only define the *size* field and the *generate* method:

$$
\begin{array}{lll}
\mathcal{T}[\![R, \boxplus]\!][\ \ ] & body & = body \\
\mathcal{T}[\![R, \boxplus]\!][\ x \leftarrow g, gs\ ] & body & = g.generate[\![R, \boxplus]\!](\texttt{fn}\ () \Rightarrow \mathcal{T}[\![R, \boxplus]\!][gs]body) \\
\mathcal{T}[\![R, \boxplus]\!][\ p\texttt{,}\ gs\ ] & body & = \texttt{if}\ p\ \texttt{then}\ (\mathcal{T}[\![R, \boxplus]\!][gs]body)\ \texttt{else}\ \mathrm{Identity}[\![\boxplus]\!]\ \texttt{end}
\end{array}
$$

Figure 32.1: Naive and simple desugaring of generator lists using only the $generate$ method.

```
value object BlockedRange(lo: ℤ64, hi: ℤ64, b: ℤ64) extends Generator⟦ℤ64⟧
  size : ℤ64 = hi − lo + 1
  generate⟦R extends Monoid⟦R, ⊕⟧, opr ⊕⟧(body : ℤ64 → R) : R =
    if size ≤ max(b, 1) then
      r : R = Identity⟦⊕⟧
      i : ℤ64 = lo
      if i ≤ hi then
        label done do
          while true do
            r := r ⊕ body(i)
            if i ≥ hi then exit done with () end
            i += 1
          end
        end
      end
      r
    else
      mid = ⌈lo/2⌉ + ⌊hi/2⌋
      BlockedRange(lo, mid, b).generate(body)⊕
          BlockedRange(mid + 1, hi, b).generate(body)
    end
end
```

This example generates the integers between lo and hi inclusive. It does this using *recursive subdivision*. Recursive subdivision is the recommended technique for exposing large amounts of parallelism in a Fortress program because it adjusts easily to varying machine sizes and can be dynamically load balanced. In this example we divide the range in half if it is larger than the block size b; these two halves are computed in parallel (recall that the arguments to an operator are evaluated in parallel). If the range is smaller than b, then it is enumerated serially using a `while` loop, accumulating the result r as it goes.

The remainder of this section describes in detail the desugaring of generator lists and expressions with generators into invocations of the $generate$ and $join$ methods of the generators in the generator list. It then outlines how method overloading may be used to specialize the behavior of particular combinations of generators and reductions.

32.8.1 Simple Desugaring of Expressions with Generators

Each expression with generators is desugared into the following general form:

$$wrapper(\texttt{fn}\ () \Rightarrow \mathcal{T}[\![R, \boxplus]\!][\ gs\ ]body)$$

where the desugaring must provide appropriate instantiations of $wrapper$, $body$, and the reduction static parameters, R and $\boxplus$. A simple and easily-understood desugaring "$\mathcal{T}[\![R, \boxplus]\!][\ gs\ ]body$" for generator lists is shown in Figure 32.1.

The desugaring of *GeneratorList* takes three parameters: a block of static parameters, R and $\boxplus$, the actual generator list (which we enclose in square brackets), and the $body$ expression which should be used. Here and in subsequent desugarings, v in $v \leftarrow g$ can stand either for a single variable or for a tuple of variables. We convert the

expr	type	wrapper	body	$[\![R, \boxplus]\!]$
$\sum_{gs} e$	R	$\sum$	e	$[\![R, +]\!]$
$lv := e, gs$	$()$	*noReduction*	$lv := e$	$[\![\mathrm{NoReduction}, \oplus]\!]$
$\langle\, e \mid gs \,\rangle$	$\mathrm{List}[\![E]\!]$	*closeList*	*singletonOpen*(e)	$[\![\mathrm{OpenList}, +\!+]\!]$

Figure 32.2: Desugaring of expressions with generators. Top to bottom: big operators (here $\sum$ is used as an example; the appropriate library function is called on the right-hand side), assignments, and comprehensions (here list comprehensions are shown; with the exception of array comprehensions, other comprehensions are similar to list comprehensions).

provided *GeneratorList* into a nested series of calls to $generate$. For example, when we perform the desugaring $\mathcal{T}[\![\mathbb{Z}64, +]\!][x \leftarrow xs, y \leftarrow ys, x \neq y](x \cdot y)$ we obtain the following code:

$$
\begin{array}{l}
xs.generate[\![\mathbb{Z}64, +]\!](\texttt{fn}\ x \Rightarrow \\
\quad ys.generate[\![\mathbb{Z}64, +]\!](\texttt{fn}\ y \Rightarrow \\
\quad\quad \texttt{if}\ x \neq y\ \texttt{then}\ (x \cdot y)\ \texttt{else}\ \mathrm{Identity}[\![+]\!]\ \texttt{end}))
\end{array}
$$

Some example desugarings of expressions with generators are shown in Figure 32.2 for big operators, assignments, and list comprehensions (set and multiset comprehensions are similar to list comprehensions).

The simplest desugarings are the ones for big operators such as $\sum$. The type of the traversal corresponds to the type of the result. The body expression used is exactly the body expression of the big operator. The wrapper function is named by the big operator itself. For example, the $\sum$ operator has the following declaration:

$$\texttt{opr}\ \textstyle\sum[\![\, R\ \texttt{extends}\ \mathrm{CommutativeMonoid}[\![R, +]\!]]\!](rhs : () \to R) \colon R = rhs()$$

Assignments desugar in a manner similar to big operators. However, they make use of the special $\mathrm{NoReduction}$ type, which is a singleton type which extends $\mathrm{CommutativeMonoid}[\![\mathrm{NoReduction}, \oplus]\!]$. We can think of $\mathrm{NoReduction}$ as composing the writes of the assignments in parallel. The wrapper $noReduction$ is defined as follows:

$$
\begin{array}{l}
noReduction(rhs : () \to \mathrm{NoReduction}) \colon () = \texttt{do} \\
\quad rhs() \\
\quad () \\
\texttt{end}
\end{array}
$$

Lists also desugar in a similar way. The desugaring given in Figure 32.2 makes use of the $\mathrm{OpenList}$ type—such a list is constructed with an updatable tail cell, permitting partially-constructed lists to be appended in constant time using the $+\!+$ (`DOUBLE_PLUS`) operator. The $closeList$ operation converts the result into an ordinary non-updatable list.

An array comprehension simply desugars into a factory function call and a series of assignments:

$$
\begin{array}{l}
[\ i_1 = e_1 \mid gs_1 \\
\ \ i_2 = e_2 \mid gs_2 \\
\ \ \ldots \\
\ \ i_n = e_n \mid gs_n\ ]
\end{array}
\quad\longrightarrow\quad
\begin{array}{l}
\texttt{do}\ a = array() \\
\quad a[i_1] := e_1, gs_1 \\
\quad a[i_2] := e_2, gs_2 \\
\quad \ldots \\
\quad a[i_n] := e_n, gs_n \\
\quad a \\
\texttt{end}
\end{array}
$$

The desugaring of a `for` loop depends upon the set of reduction variables. We conceptually desugar the `for` loop with reduction variables, $r_1, r_2, \ldots, r_n$, reduced using the reduction operator $\oplus$ for type $(T_1, T_2, \ldots T_n)$ as follows:

$$\texttt{for}\ gs\ \texttt{do}\ block\ \texttt{end}$$
$$\longrightarrow$$

$$(r_1, r_2, \ldots r_n) \oplus= \mathcal{T}[\![(T_1, T_2, \ldots T_n), \oplus]\!][gs]\ (\texttt{do}$$
$$(r_1, r_2, \ldots r_n) : (T_1, T_2, \ldots T_n) := \text{Identity}[\![\oplus]\!]$$
$$\textit{block}$$
$$(r_1, r_2, \ldots r_n)$$
$$\texttt{end})$$

In practice, a tuple type is not a monoid. If there is only one reduction variable, this is not a problem. If there are no reduction variables, we simply use the type NoReduction used in desugaring assignments. When there are multiple reduction variables, we use nested applications of types extending the trait ReductionPair. These types encode the common properties of the variables being reduced. Recall that every reduction variable must at least have type Monoid, so it is not difficult to guarantee that ReductionPair itself also extends Monoid.

32.8.2 Accounting for Dependencies among Generators

The naive desugaring for generator lists in Figure 32.1 assumes there are always data dependencies among generators. The actual desugaring makes use of the $join$ method in the Generator trait to group together generators that have no data dependencies. The goal is to permit library code to define more efficient merged generators for generator pairs. For example, it is possible for the $join$ method to take the generator list $i \leftarrow 1\,\#\,100, j \leftarrow 2\,\#\,200$ and generate a blocked two-dimensional traversal. This could then be joined with $k \leftarrow 3\,\#\,300$ to obtain a three-dimensional blocked traversal.

However, most generators will simply make use of the default definition of $join$ which calls $\text{SimplePairGenerator}$:

$\texttt{object}\ \text{SimplePairGenerator}[\![\, A, B \,]\!](\mathit{outer} : \text{Generator}[\![\, A \,]\!], \mathit{inner} : \text{Generator}[\![\, B \,]\!])$
$\quad\quad \texttt{extends}\ \text{Generator}[\![\, (A, B) \,]\!]$
$\quad \mathit{size} : \mathbb{Z}64 = \mathit{outer}.\mathit{size} \cdot \mathit{inner}.\mathit{size}$
$\quad \mathit{generate}[\![\, R\ \texttt{extends}\ \text{Monoid}[\![\, R, \oplus \,]\!], \texttt{opr}\ \oplus \,]\!](\mathit{body} : (A, B) \to R) : R =$
$\quad\quad \mathit{outer}.\mathit{generate}(\texttt{fn}\ (a : A) \Rightarrow \mathit{inner}.\mathit{generate}(\texttt{fn}\ (b : B) \Rightarrow \mathit{body}(a, b)))$
$\quad \mathit{join}[\![\, N \,]\!](\mathit{other} : \text{Generator}[\![\, N \,]\!]) : \text{Generator}[\![\, ((A, B), N) \,]\!] =$
$\quad\quad \text{SimpleMapGenerator}(\mathit{outer}.\mathit{join}(\mathit{inner}.\mathit{join}(\mathit{other})),$
$\quad\quad\quad\quad (\texttt{fn}\ (a, (b, n)) \Rightarrow ((a, b), n)))$
$\texttt{end}$

Note how $\text{SimplePairGenerator}$ itself overrides the $join$ method. When we attempt to join an existing pair of joined generators, we first attempt to $join$ the $inner$ generator of the pair with the $other$ generator (the new innermost generator) passed in. This means that every generator will have the opportunity to combine with both its left and right neighbors if neither has a dependency which prevents it. Note that we use a $\text{SimpleMapGenerator}$, which simply applies a function to the result of another generator, to re-nest the tuples produced by the nested $join$ operation.

Which pairs of adjacent traversals are combined using $join$? This question is complicated by examples such as $i \leftarrow 1:100, j \leftarrow 1:100, k \leftarrow i:100, l \leftarrow j:100$. We can either combine i and j traversals, or we can combine j and k traversals. In the former case we can also combine k and l traversals. The Fortress compiler is free to choose any grouping subject to the following constraints:

- Two generators may not be combined using $join$ if the second is data dependent upon the first.
- Generator order must be preserved when invoking $join$.
- When a chain of three or more generators is joined, the traversals must be combined left-associatively.

We can obtain a simple greedy desugaring which joins together traversals in accordance with the above rules by simply adding the following desugaring rule which takes precedence over those given in Figure 32.1 when each variable bound in v_1 does not occur free in g_2.

$$\mathcal{T}[\![R, \boxplus]\!][\, v_1 \leftarrow g_1, v_2 \leftarrow g_2, gs \,]\ \mathit{body} \quad = \quad \mathcal{T}[\![R, \boxplus]\!][\, (v_1, v_2) \leftarrow g_1.\mathit{join}(g_2), gs \,]\ \mathit{body}$$

32.8.3 Using Overloading to Adapt Generators and Traversals

Overloaded instances of the *generate* method can be used to adapt a generator to the particular properties of the reduction being performed. For example, a commutative monoid need only maintain a single variable *result* containing the reduced value so far:

```
value object BlockedRange(lo: ℤ64, hi: ℤ64, b: ℤ64) extends Generator⟦ℤ64⟧
  ...
  generate⟦R extends CommutativeMonoid⟦R, ⊕⟧, opr ⊕⟧(body : ℤ64 → R) : R = do
    result : R = Identity⟦⊕⟧
    traverse(l, u) =
      if u − l + 1 ≤ max(b, 1) then
        i : ℤ64 = l
        while i ≤ u do
          t = body(i)
          atomic result := result ⊕ t
          i += 1
        end
        ()
      else
        mid = ⌈l/2⌉ + ⌊u/2⌋
        (traverse(l, mid), traverse(mid + 1, u))
        ()
      end
    traverse(lo, hi)
    result
  end
end
```

The choice of whether to apply this transformation is left up to the author of the generator; when many iterations run in parallel the *result* variable becomes a scalability bottleneck and this technique should not be used.

Various other properties of the reduction operator can be exploited:

- Idempotent reductions permit redundant computation. For example, when computing the maximum element of a set it might be simpler to enumerate set elements more than once.
- On the other hand, sometimes a more efficient non-idempotent operator can be used for a reduction if the generator promises never to produce duplicates—this fact can be used to advantage in set, multiset, or map comprehensions.
- If the reduction operator has a zero, this can be used to exit early from a partial computation. This requires that the body expression have no visible side effects such as writes or `io` actions.

At the moment, the author of a Generator is responsible for taking advantage of opportunities such as these. In future, we expect some standardized support for efficient versions of various traversals based on experience with the definitions provided here.

32.8.4 Making a Serial Version of a Generator or Distribution

A generator g can be made sequential simply by calling the builtin function *sequential* as follows:

$v \leftarrow sequential(g)$

Several builtin generators (such as those for array indices) have an associated distribution. For these generators, $sequential$ function simply re-distributes the underlying object as follows:

$$sequential(r) = \text{Sequential}.distribute(r)$$

As a convenient shorthand, the $sequential$ function is also defined to work for distributions themselves. The complete signature for the overloadings of $sequential$ is as follows:

$$sequential[\![\, E \,]\!](g : \text{Generator}[\![E]\!]) : \text{Generator}[\![E]\!]$$
$$sequential[\![\, E \,]\!](d : \text{Distribution}) : \text{Distribution}$$

The $sequential$ function has special meaning to the Fortress implementation; there is no need to distinguish reduction variables in loops for which generator is surrounded by a direct call to $sequential$.

Note that at the moment there is no way to tell the compiler for performance reasons that we really mean it when we ask for sequentiality, as opposed to saying that we should preserve sequential semantics. Future versions of this specification may use the Local distribution for this purpose, or provide additional functions on generators which guarantee serial execution (rather than simply providing sequential semantics).

Chapter 33

Overloaded Functional Declarations

Fortress allows multiple functional declarations to be in scope of a particular program point. We call this overloading. Chapter 15 describes how to determine which overloaded declarations are applicable to a particular functional call, and when several are applicable, how to select the most specific one among them. In this chapter, we give a set of restrictions on overloaded declarations that guarantee there exists a most specific declaration for any given functional call. These rules are complicated by the presence of coercion, which may enlarge the set of declarations that are applicable to a functional call, as discussed in Chapter 17.

33.1 Principles of Overloading

Fortress allows multiple functional declarations of the same name to be declared in a single scope. However, recall from Chapter 7 the following shadowing rules:

- dotted method declarations shadow top-level function declarations with the same name, and
- dotted method declarations provided by a trait or object declaration shadow functional method declarations with the same name that are provided by a different trait or object declaration.

Also, note that a trait or object declaration must not have a functional method declaration and a dotted method declaration with the same name, either directly or by inheritance. Therefore, top-level functions can overload with other top-level functions and functional methods, dotted methods with other dotted methods, and functional methods with other functional methods and top-level functions.

Overloading functional declarations allows the benefits of polymorphic declarations. However, with these benefits comes the potential for ambiguous calls at run time. Fortress places restrictions on the *declarations* of functionals to eliminate the *possibility* of ambiguous call at run time, whether or not these calls actually appear in the program.

Furthermore, these restrictions are checked statically. In fact, the restrictions on overloading in Fortress allow the compiler to identify the statically most specific declaration for a particular call. Therefore an implementation strategy may be used in which the statically most specific declaration is identified statically, and the runtime dispatch mechanism need only consider dispatching among that declaration plus declarations that are more specific than that declaration (proof of this is given in Section B.2).

Rather than describe the overloadings that are forbidden in Fortress, this chapter outlines several criteria for valid functional overloading. At any given program point, there may be a set of overloaded declarations that are in scope. Fortress determines whether there is a possibility for ambiguous calls from this set by comparing declarations pairwise. The following three sections describe the rules to accept a pair of overloaded functional declarations. If a pair

of overloaded declarations satisfies any one of the three rules, it is considered valid overloading. In addition, the overloaded declarations must have static parameters that are identical (up to α-equivalence). Also, valid overloading for declarations that contain keyword or varargs parameters is determined by analyzing the expansion of these declarations into declarations without such parameters, as described in Section 15.4.

Section 33.2 states the *Subtype Rule*, which stipulates that the parameter type of one declaration be a subtype of the parameter type of the other. In this case, there is no possibility of ambiguous calls, because one declaration is more specific than the other. This section also places a restriction on the return types of the overloaded declarations to ensure that type safety is not violated. Section 33.3 defines the *Incompatibility Rule* that, if satisfied by a pair of declarations, guarantees that neither declaration is applicable to the same functional call. In Section 33.4, the *More Specific Rule* requires the existence of a declaration that is more specific than both overloaded declarations in the situation that both are applicable to a given call.

In the remainder of this chapter we build on the terminology and notation defined in Chapter 15 and Chapter 17.

33.2 Subtype Rule

If the parameter type of one declaration is a subtype of the parameter type of another then there is no possibility of ambiguous calls because the most specific declaration will be dispatched to. This is the basis of the Subtype Rule. The Subtype Rule also requires a relationship between the return types of the two declarations. Without such a requirement, a program may be statically well typed but have a runtime error because the return type of a dynamically resolved functional is not a subtype of the return type of the statically resolved functional.

The Subtype Rule for Functions and Functional Methods: Suppose that $f(P) : U$ and $f(Q) : V$ are two distinct function or functional method declarations both visible at some point Z in a Fortress program (Z need not be the site of a call). If $P \prec Q$ and $U \preceq V$ then $f(P)$ and $f(Q)$ are valid overloadings.

The Subtype Rule for Dotted Methods: Suppose that $P_0.f(P) : U$ and $Q_0.f(Q) : V$ are two distinct dotted method declarations provided by a trait or object C. If $(P_0, P) \prec (Q_0, Q)$ and $U \preceq V$ then $P_0.f(P)$ and $Q_0.f(Q)$ are valid overloadings.

33.3 Incompatibility Rule

The basic idea behind the Incompatibility Rule is that if there is no call to which two overloaded declarations are both applicable then there is no potential for ambiguous calls. In such a case, we say that the declarations are incompatible. Without coercion, incompatibility is equivalent to exclusion. However, the presence of coercion complicates the definition of incompatibility. To formally define incompatibility we first define the following notation. For types T and U, we say that T and U *do not share coercions*, and write $T \between U$, if any type that coerces to T excludes any type that coerces to U:

$$T \between U \iff \forall A, B : A \rightarrow T \wedge B \rightarrow U \implies A \Diamond B.$$

We say that T *is incompatible with* U, and write $T \blacklozenge U$, if T and U exclude, reject each other, and do not share

coercions:

$$\begin{aligned} T \blacklozenge U &\iff T \Diamond U \wedge T \not\rightarrow U \wedge U \not\rightarrow T \wedge T \not\wr U \\ &\iff T \Diamond U \\ &\quad \wedge\ (\forall A : A \rightarrow T \implies A \Diamond U) \\ &\quad \wedge\ (\forall B : B \rightarrow U \implies B \Diamond T) \\ &\quad \wedge\ (\forall A, B : A \rightarrow T \wedge B \rightarrow U \implies A \Diamond B) \end{aligned}$$

Note that if $T \blacklozenge U$ then no type is substitutable for both T and U.

The Incompatibility Rule for Functions and Functional Methods: Suppose that $f(P)$ and $f(Q)$ are two distinct function or functional method declarations both visible at some point Z in a Fortress program (Z need not be the site of a call). If $P \blacklozenge Q$ then $f(P)$ and $f(Q)$ are valid overloadings.

The Incompatibility Rule for Dotted Methods: Suppose that $P_0.f(P)$ and $Q_0.f(Q)$ are two distinct dotted method declarations provided by a trait or object C. If $P \blacklozenge Q$ then $P_0.f(P)$ and $Q_0.f(Q)$ are valid overloadings.

33.4 More Specific Rule

If neither the Subtype Rule nor the Incompatibility Rule holds for a pair of overloaded declarations then they may still be valid overloadings if the More Specific Rule is satisfied. The More Specific Rule requires that for any two declarations there exists a third applicable declaration that is at least as specific as both.

This rule is complicated by the fact that functions and functional methods can overload. Recall that functional methods can be viewed semantically as top-level functions, as described in Section 9.2. However, treating functional methods as top-level functions for determining valid overloading is too restrictive. In the following example:

```
trait ℤ
    opr −(self): ℤ
end
trait ℝ
    opr −(self): ℝ
end
```

if the functional methods were interpreted as top-level functions then this program would define two top-level functions with parameter types $\mathbb{Z}$ and $\mathbb{R}$. These declarations would be statically rejected as invalid overloadings because there is no relation between $\mathbb{Z}$ and $\mathbb{R}$; another trait may extend them both without declaring its own version of the functional method which may lead to an ambiguous call at run time. To allow such overloading, we define different restrictions on overloaded function declarations and overloaded functional method declarations. When function and functional method declarations are overloaded, the more restrictive rule for function declarations is used. This rule follows.

The More Specific Rule for Functions and Functional Methods: Suppose that $f(P)$ and $f(Q)$ are two function or functional method declarations both visible at some point Z in a Fortress program (Z need not be the site of a call) such that neither P nor Q is a subtype of the other and P and Q are not incompatible with one another. Let $\mathcal{S}$ be the set of types that P defines coercions from and $\mathcal{T}$ be the set of types that Q defines coercions from. $f(P)$ and $f(Q)$ are valid overloadings if all of the following hold:

- either $P \Diamond Q$ or there is a declaration $f(P \cap Q)$ visible at Z, and
- either $P \lhd Q$ or $Q \lhd P$ or for all $P' \in \mathcal{S}$ and $Q' \in \mathcal{T}$ one of the two conditions holds:

- $P' \diamondsuit Q'$, or
- there is a declaration $f(P' \cap Q')$ visible at Z.

Recall that $P \cap Q$ is the intersection of types P and Q as defined in Section 8.8. If for some type S we have $S \preceq P$ and $S \preceq Q$ then $S \preceq (P \cap Q)$, but it's not necessarily the case that $S = (P \cap Q)$ since another type may be more specific than both P and Q. For example, suppose the following:

```
trait S comprises {U, V} end
trait T comprises {V, W} end
trait U extends S excludes W end
trait V extends {S, T} end
trait W extends T end
```

$f(s : S) = 1$
$f(t : T) = 1$
$f(v : V) = 1$

Because of the `comprises` clauses of S and T and the `excludes` clause of U, any subtype of both S and T must be a subtype of V. Thus, $V = S \cap T$, and the declaration $f(V)$ "disambiguates" $f(S)$ and $f(T)$, i.e., it is applicable to and more specific for any call to which both $f(S)$ and $f(T)$ are applicable.

This requirement should not be difficult to obey, especially because the compiler can give useful feedback. First example:

$foo(x : \mathrm{Number}, y : \mathbb{Z}64) = \ldots$
$foo(x : \mathbb{Z}64, y : \mathrm{Number}) = \ldots$

Assuming that $\mathbb{Z}64 \prec \mathrm{Number}$, the compiler reports that these two declarations are a problem because of ambiguity and suggests that a new declaration for $foo(\mathbb{Z}64, \mathbb{Z}64)$ would resolve the ambiguity. Second example:

$bar(x : \mathrm{Printable}) = \ldots$
$bar(x : \mathrm{Throwable}) = \ldots$

Assuming that $\mathrm{Printable}$ and $\mathrm{Throwable}$ are neither comparable by the subtyping relation nor disjoint, the compiler reports that these two declarations are a problem because $\mathrm{Printable}$ and $\mathrm{Throwable}$ are incomparable but possibly overlapping types. As a result, these two declarations are statically rejected.

The More Specific Rule for Dotted Methods: Suppose that $P_0.f(P)$ and $Q_0.f(Q)$ are two dotted method declarations provided by a trait or object C such that neither (P_0, P) nor (Q_0, Q) is a subtype of the other and P and Q are not incompatible with one another. Let $\mathcal{S}$ be the set of types that P defines coercions from and $\mathcal{T}$ be the set of types that Q defines coercions from. $P_0.f(P)$ and $Q_0.f(Q)$ are valid overloadings if all of the following hold:

- either $P \diamondsuit Q$ or there is a declaration $R_0.f(P \cap Q)$ provided by C with $R_0 \preceq (P_0 \cap Q_0)$, and
- either $P \triangleleft Q$ or $Q \triangleleft P$ or for all $P' \in \mathcal{S}$ and $Q' \in \mathcal{T}$ one of the two conditions holds:
  - $P' \diamondsuit Q'$, or
  - there is a declaration $R_0.f(P' \cap Q')$ provided by C with $R_0 \preceq (P_0 \cap Q_0)$.

Unlike for functions and functional methods, the More Specific Rule for dotted methods only applies to dotted methods that are provided by the same trait or object. This is possible because two dotted methods are applicable to a given call $A_0.f(A)$ only if they are both provided by the trait or object A_0. This is not the case for functional methods as the following rule shows.

The More Specific Rule for Functional Methods: Suppose that $f(P)$ and $f(Q)$ are two functional method declarations occurring in trait or object declarations such that neither P nor Q is a subtype of the other and P and Q are not incompatible with one another. Let $f(P)$ and $f(Q)$ have self parameters at i and j respectively. Also, let $\mathcal{S}$ be the set of types that P defines coercions from and $\mathcal{T}$ be the set of types that Q defines coercions from. $f(P)$ and $f(Q)$ are valid overloadings if all of the following hold:

- $i = j$
- either $P \lozenge Q$ or if there exists a trait or object C that provides both $f(P)$ and $f(Q)$ then $P \neq Q$ and there is a declaration $f(P \cap Q)$ provide by C, and
- either $P \lhd Q$ or $Q \lhd P$ or for all $P' \in \mathcal{S}$ and $Q' \in \mathcal{T}$ one of the two conditions holds:
  - $P' \lozenge Q'$, or
  - there is a declaration $f(P' \cap Q')$ provided by C.

Verifying the More Specific Rule for functional methods can be thought of as a two step process. First there must be no ambiguity caused by the position of the self parameter. To guarantee this, overloaded declarations with different self parameter positions must be incompatible with one another. Second, functional method declarations that create the potential for ambiguity because neither is more specific than the other must be accompanied by a third disambiguating declaration that is more specific than both. Notice that the second step is similar to the overloading requirements placed on dotted methods.

33.5 Coercion and Overloading Resolution

The restrictions on overloaded declarations given in this chapter are sufficient to prove the following two facts:

1. If no declaration is applicable to a static call but there is a declaration that is applicable with coercion then there exists a single most specific declaration that is applicable with coercion to the static call.
2. If any declaration is applicable to a static call then there exists a single most specific declaration that is applicable to the static call and a single most specific declaration that is applicable to the corresponding dynamic call.

Moreover, we can prove that the most specific declaration that is applicable to a dynamic call is more specific than the most specific declaration that is applicable to the corresponding static call.

Appendix B formally proves that the restrictions discussed in the previous sections guarantee the static resolution of coercion (described in Section 17.5) is well defined for functions (the case for methods is analogous). Also in Appendix B is a proof that the restrictions placed on overloaded function declarations are sufficient to guarantee no undefined nor ambiguous calls at run time (again, the case for methods is analogous).

Chapter 34

Operator Declarations

An operator declaration may appear anywhere a top-level function or method declaration may appear. Operator declarations are like other function or method declarations in all respects except that an operator declaration has the special reserved word `opr` and has an operator name (see Section 16.1 for a discussion of valid operator names) instead of an identifier. The precise placement of the operator name within the declaration depends on the fixity of the operator. Like other functionals, operators may have overloaded declarations (see Chapter 15 for a discussion of overloading). These overloadings may be of the same or differing fixities.

Syntax:

OpDecl	::=	*FnDecl*
FnDecl	::=	*AbsFnDecl*
	|	*FnDef*
FnDef	::=	*FnMod** *FnHeader* = *Expr*
FnHeader	::=	*OpHeader*
OpHeader	::=	`opr` *Op* [*StaticParams*] *ValParam* [*IsType*] *FnClauses*
	|	`opr` [*StaticParams*] *ValParam Op* [*IsType*] *FnClauses*
	|	`opr` [*StaticParams*] *LeftEncloser ValParams RightEncloser* [:= *ValParam*] [*IsType*] *FnClauses*

An operator declaration has one of seven forms: infix/multifix operator declaration, prefix operator declaration, postfix operator declaration, nofix operator declaration, bracketing operator declaration, subscripting operator method declaration, and subscripted assignment operator method declaration. Each is invoked according to specific rules of syntax. An operator method declaration should be a functional method declaration, a subscripting operator method declaration, or a subscripted assignment operator method declaration.

34.1 Infix/Multifix Operator Declarations

An infix/multifix operator declaration has the special reserved word `opr` and then an operator name where a functional declaration would have an identifier. The declaration must not have any keyword parameters, and must be capable of accepting at least two arguments. It is permissible to use a varargs parameter; in fact, this is a good way to define a multifix operator. Static parameters (described in Chapter 11) may also be present, between the operator and the parameter list.

An expression consisting of an infix operator applied to an expression will invoke an infix/multifix operator declaration. The compiler considers all infix/multifix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals. If the expression is actually multifix, the invocation will pass more than two arguments.

An infix/multifix operator declaration may also be invoked by a prefix or nofix (but not a postfix) operator application if the declaration is applicable.

Note that superscripting (`^`) may be defined using an infix operator declaration even though it has very high precedence and cannot be used as a multifix operator. (An operator declaration for superscripting should have exactly two value parameters.)

Example:

$\texttt{opr MAX}[\![T\ \texttt{extends}\ \mathrm{Rational}]\!](x:T, y:T):T = \texttt{if}\ x > y\ \texttt{then}\ x\ \texttt{else}\ y\ \texttt{end}$

34.2 Prefix Operator Declarations

A prefix operator declaration has the special reserved word `opr` and then an operator name where a functional declaration would have an identifier. The declaration must have one value parameter, which must not be a keyword parameter or varargs parameter. Static parameters may also be present, between the operator and the parameter list.

An expression consisting of a prefix operator applied to an expression will invoke a prefix operator declaration. The compiler considers all prefix and infix/multifix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Example:

$\texttt{opr}\ \tilde{}(x:\mathrm{Widget}):\mathrm{Widget} = x.\mathit{invert}()$

34.3 Postfix Operator Declarations

A postfix operator declaration has the special reserved word `opr` where a functional declaration would have an identifier; the operator name itself *follows* the parameter list. The declaration must have one value parameter, which must not be a keyword parameter or varargs parameter. Static parameters may also be present, between the special reserved word `opr` and the parameter list.

An expression consisting of a postfix operator applied to an expression will invoke a postfix operator declaration. The compiler considers all postfix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Example:

$\texttt{opr}\ (n:\mathrm{Integer})!\ = \prod[i \leftarrow 1:n]i \quad (*\ \text{factorial}\ *)$

34.4 Nofix Operator Declarations

A nofix operator declaration has the special reserved word `opr` and then an operator name where a functional declaration would have an identifier. The declaration must have no parameters.

An expression consisting only of a nofix operator will invoke a nofix operator declaration. The compiler considers all nofix and infix/multifix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Uses for nofix operators are rare, but those rare examples are very useful. For example, the colon operator is used to construct subscripting ranges, and it is the nofix declaration of `:` that allows a lone `:` to be used as a subscript:

$\texttt{opr}\ :() = \mathrm{ImplicitRange}$

34.5 Bracketing Operator Declarations

A bracketing operator declaration has the special reserved word `opr` where a functional declaration would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by the brackets being defined. A bracketing operator declaration may have any number of parameters, keyword parameters, and varargs parameters in the value parameter list. Static parameters may also be present, between the special reserved word `opr` and the parameter list. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets.

An expression consisting of zero or more comma-separated expressions surrounded by a bracket pair will invoke a bracketing operator declaration. The compiler considers all bracketing operator declarations for that type of bracket pair that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals. For example, the expression $\langle p, q \rangle$ might invoke the following bracketing method declaration:

($*$ angle bracket notation for inner product $*$)
`opr` $\langle x : \mathrm{Vector}, y : \mathrm{Vector} \rangle = \sum [i \leftarrow x.indices] x_i \cdot y_i$

($*$ vector space norm (may not be the most efficient) $*$)
`opr` $\|x : \mathrm{Vector}\| = sqrt \langle x, x \rangle$

34.6 Subscripting Operator Method Declarations

A subscripting operator method declaration has the special reserved word `opr` where a method declaration would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by a pair of brackets. A subscripting operator method declaration may have any number of value parameters, keyword parameters, and varargs parameters in that value parameter list. Static parameters may also be present, between the special reserved word `opr` and the parameter list. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets; in particular, the square brackets ordinarily used for indexing may be used.

An expression consisting of a subexpression immediately followed (with no intervening whitespace) by zero or more comma-separated expressions surrounded by brackets will invoke a subscripting operator method declaration. Methods for the expression preceding the bracketed expression list are considered. The compiler considers all subscripting operator method declarations that are both accessible and applicable, and the most specific method declaration is chosen according to the usual overloading rules. For example, the expression foo_p might invoke the following subscripting method declaration because expressions in the square brackets are rendered as subscripts:

($*$ subscripting method $*$)
`opr` $[x : \mathrm{BizarroIndex}] =$ `self`$.bizarroFetch(x)$

34.7 Subscripted Assignment Operator Method Declarations

A subscripted assignment operator method declaration has the special reserved word `opr` where a method declaration would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by a pair of brackets; this is then followed by the operator `:=` and then a second value parameter list in parentheses, which must contain exactly one non-keyword value parameter. A subscripted assignment operator method declaration may have any number of value parameters within the brackets; these value parameters may include keyword parameters and varargs parameters. A result type may appear after the second value parameter list, but it must be $()$. Static parameters may also be present, between the special reserved word `opr` and the first parameter list. Any paired Unicode brackets

may be so defined *except* ordinary parentheses and white square brackets; in particular, the square brackets ordinarily used for indexing may be used.

An assignment expression consisting of an expression immediately followed (with no intervening whitespace) by zero or more comma-separated expressions surrounded by brackets, followed by the assignment operator `:=`, followed by another expression, will invoke a subscripted assignment operator method declaration. Methods for the expression preceding the bracketed expression list are considered. The compiler considers all subscript operator method declarations that are both accessible and applicable, and the most specific method declaration is chosen according to the usual overloading rules. When a compound assignment operator (described in Section 13.8) is used with a subscripting operator and a subscripted assignment operator, for example $a_3 \mathrel{+}= k$, both a subscripting operator declaration and a subscripted assignment operator declaration are required. For example, the assignment $foo_p := myWidget$ might invoke the following subscripted assignment method declaration:

$(*$ subscripted assignment method $*)$
`opr` $[x : \text{BizarroIndex}] := (newValue : \text{Widget}) =$ `self`$.bizarroInstall(x, newValue)$

34.8 Conditional Operator Declarations

A *conditional operator* is a binary operator (other than '`:`') that is immediately followed by '`:`'; see Section 16.6. A conditional operator expression $x@:y$ is syntactic sugar for $x@(\texttt{fn}\ () \Rightarrow y)$; that is, the right-hand operand is converted to a "thunk" (zero-parameter function) that then becomes the right-hand operand of the corresponding unconditional operator. Therefore a conditional operator is simply implemented as an overloading of the operator that accepts a thunk.

It is also permitted for a conditional operator to have a preceding as well as a following colon. A conditional operator expression $x:@:y$ is syntactic sugar for $(\texttt{fn}\ () \Rightarrow x)@(\texttt{fn}\ () \Rightarrow y)$; that is, each operand is converted to a thunk. This mechanism is used, for example, to define the results-comparison operator `:~:`, which takes exceptions into account.

The conditional $\wedge$ and $\vee$ operators for boolean values, for example, are implemented as methods in this manner:

`opr` $\wedge(\texttt{self}, other : \text{Boolean}) = \texttt{if self then}\ other\ \texttt{else}\ false\ \texttt{end}$
`opr` $\wedge(\texttt{self}, other : () \to \text{Boolean}) = \texttt{if self then}\ other()\ \texttt{else}\ false\ \texttt{end}$
`opr` $\vee(\texttt{self}, other : \text{Boolean}) = \texttt{if self then}\ true\ \texttt{else}\ other\ \texttt{end}$
`opr` $\vee(\texttt{self}, other : () \to \text{Boolean}) = \texttt{if self then}\ true\ \texttt{else}\ other()\ \texttt{end}$

34.9 Big Operator Declarations

A *big operator* such as $\sum$ or $\prod$ is declared as a usual operator declaration. See Section 32.8.1 for an example declaration of a big operator. A big operator application is called a *reduction expression* and described in Section 13.18.

Chapter 35

Dimensions and Units Declarations

Syntax:

DimUnitDecl ::= `dim` *Id* [= *DimRef*] [`default` *Unit*]
| (`unit` | `SI_unit`) *Id*$^+$ [: *DimRef*] [= *Expr*]
| `dim` *Id* [= *DimRef*] (`unit` | `SI_unit`) *Id*$^+$ [= *Expr*]

35.1 Dimensions Declarations

Dimensions may be explicitly declared; every declared dimension must be declared at the top level of a program component, not within a block expression or trait. Other dimensions may be constructed by multiplying and dividing other dimensions, as described in Chapter 18. An explicitly declared dimension may be a *base dimension* (with no definition specified) or a *derived dimension* (with a definition specified in the form of an initialization expression).

The set of all dimensions has the algebraic structure of a free abelian group. The identity element of this group is the dimension Unity, which represents dimensionlessness.

For every two dimensions D and E, there is a dimension DE (which may also be written $D \cdot E$), corresponding to the product of the dimensions D and E and a dimension D/E, corresponding to the quotient of the dimensions D and E. The syntactic sugar $1/D$ is equivalent to Unity/D for all dimensions D. A dimension can be raised to a rational power where both the numerator and the denominator of the rational power must be a valid `nat` *parameter* instantiation (as described in Section 11.2); D^0 is the same as Unity, D^1 is the same as D, and D^{m+n} is the same as $D^m D^n$. The syntactic sugar D^{-n} is the same as Unity/D^n.

Here are some examples of base dimension declarations:

```
dim Length
dim Mass
dim Time
dim ElectricCurrent
```

Here are some examples of computed dimensions:

$\mathrm{Length}/\mathrm{Time}$
$\mathrm{Velocity}/\mathrm{Time}$
$\mathrm{Length} \cdot \mathrm{Mass}/\mathrm{Time}^2$
$\mathrm{Length}\ \mathrm{Mass}\ \mathrm{Time}^{-2}$
$\mathrm{ElectricCurrent}/\mathrm{Length}^2$

and here some of these computed dimensions are given names through the use of derived dimension declarations:

```
dim Velocity = Length/Time
dim Acceleration = Velocity/Time
dim CurrentDensity = ElectricCurrent/Length^2
```

35.2 Units Declarations

Every unit belongs to exactly one dimension, which is the type of the unit. A dimension may have more than one unit, but one of these units may be singled out as the *default unit* for that dimension by adding a `default` clause:

```
dim Length default meter
dim Mass default kilogram
dim Time default second
```

The default unit is used when a numerical type is multiplied by a dimension to produce a new type (see Chapter 18). If no default clause is specified for a base dimension, then it has no default unit. If no default clause is specified for a derived dimension, then it has a default unit if and only if all the dimensions mentioned in its initialization expression have defaults, in which case its default unit is calculated using the initialization expression with each dimension replaced by its default unit.

Some units are explicitly declared; every declared unit must be declared at the top level of a program component, not within a block expression or trait. Other units may be constructed by multiplying and dividing other units. An explicitly declared unit may be a *base unit* (with no definition specified) or a *derived unit* (with a definition specified in the form of an initialization expression).

The set of all units, like the set of all dimensions, has the algebraic structure of a free abelian group. The identity element of this group is the unit dimensionless, of dimension Unity. Note that there may be other units of dimension Unity, such as radian and steradian, but only dimensionless is the identity of the group of all units. (Note that there is a straightforward homomorphism of units onto dimensions, wherein every unit is mapped to its dimension.)

Here are some examples of base unit declarations:

```
unit meter : Length
unit kilogram : Mass
unit second : Time
unit ampere : ElectricCurrent
```

Here are some examples of computed units:

```
meter/second
(meter/second)/second
meter · kilogram/second^2
meter kilogram second^-2
ampere/meter^2
```

and here some computed units are given names through the use of derived unit declarations:

```
unit newton: Force = meter · kilogram/second^2
unit joule: Energy = newton meter
unit pascal: Pressure = newton/meter^2
```

In the preceding examples, the initialization expression for each unit is itself a unit. It is also permitted for the initialization expression to be a dimensioned numerical value, in which case the unit being declared is related to the unit of the dimensioned numerical value by a numerical conversion factor.

As with an ordinary variable declaration, one may omit the dimension for a unit if there is an initialization expression; the dimension of the declared unit is the dimension of the unit of the expression.

Every unit can be reduced to a canonical value as follows. A base unit is multiplied by the value 1; a `unit` parameter is multiplied by the value 1; a defined unit is replaced by its initialization expression and then every unit in that expression is replaced by its canonical form; and finally all arithmetic is performed so as to reduce the units to a single unit and the numerical values to a single numerical value. A dimensioned value with unit U is convertible by the `in` operator to a value with unit V if the canonical values for U and V have the same unit; the conversion involves multiplying the numerical value by the ratio of the numerical value of the canonical form of V to the numerical value of the canonical form of U.

For example, given the declarations:

`dim` Length
`unit` meter: Length; `unit` meters = meter
`unit` kilometer: Length = 10^3meter; `unit` kilometers = kilometer
`unit` inch: Length = 2.54×10^{-2}meter; `unit` inches = inch
`unit` foot: Length = 12 inch; `unit` feet = foot
`unit` mile: Length = 5280 foot; `unit` miles = mile

then one can say 3 miles `in` kilometers and the `in` operator will multiply the numerical value 3 by the amount of $((2.54 \times 10^{-2})(12)(5280)/10^3)$, or $25146/15625$.

Notice the subtle difference between these two declarations:

`unit` radian = meter/meter
`unit` radian = 1 meter/meter

The first declaration defines radian to be equivalent to dimensionless, and so a value with unit radian can be used anywhere a dimensionless value can be used, and vice versa. The second declaration defines radian to be convertible to dimensionless but not equivalent, and so it is necessary to use the `in` operator (or multiplication and division by radian) to convert between values in radians and truly dimensionless values.

35.3 Abbreviating Dimension and Unit Declarations

For convenience, three forms of syntactic sugar are provided when declaring dimensions and units. First, in a `unit` declaration one may mention more than one name before the colon, and the extra names are defined to be synonyms for the first name; thus

`unit` foot feet ft: Length

means exactly the same thing as

`unit` foot: Length
`unit` feet: Length = foot
`unit` ft: Length = foot

Second, instead of the reserved word `unit` one may use the reserved word `SI_unit`, which has the effect of defining not only the specified names but also names with the various SI prefixes attached. If more than one name is specified, then the last name is assumed to be a symbol and has symbol prefixes (such as M and n) attached; all other names have the full prefixes (such as mega and nano) attached. Thus

`SI_unit` $name_1$ $name_2$ $name_3$: ...

may be regarded as an abbreviation for

`unit` $name_1\ name_2\ name_3$: ...
`unit` yotta$name_1$ yotta$name_2$ Y$name_3 = 10^{24} name_1$
`unit` zetta$name_1$ zetta$name_2$ Z$name_3 = 10^{21} name_1$
`unit` exa$name_1$ exa$name_2$ E$name_3 = 10^{18} name_1$
`unit` peta$name_1$ peta$name_2$ P$name_3 = 10^{15} name_1$
`unit` tera$name_1$ tera$name_2$ T$name_3 = 10^{12} name_1$
`unit` giga$name_1$ giga$name_2$ G$name_3 = 10^{9} name_1$
`unit` mega$name_1$ mega$name_2$ M$name_3 = 10^{6} name_1$
`unit` kilo$name_1$ kilo$name_2$ k$name_3 = 10^{3} name_1$
`unit` hecto$name_1$ hecto$name_2$ h$name_3 = 10^{2} name_1$
`unit` deka$name_1$ deka$name_2$ da$name_3 = 10 name_1$
`unit` deci$name_1$ deci$name_2$ d$name_3 = 10^{-1} name_1$
`unit` centi$name_1$ centi$name_2$ c$name_3 = 10^{-2} name_1$
`unit` milli$name_1$ milli$name_2$ m$name_3 = 10^{-3} name_1$
`unit` micro$name_1$ micro$name_2$ $\mu name_3 = 10^{-6} name_1$
`unit` nano$name_1$ nano$name_2$ n$name_3 = 10^{-9} name_1$
`unit` pico$name_1$ pico$name_2$ p$name_3 = 10^{-12} name_1$
`unit` femto$name_1$ femto$name_2$ f$name_3 = 10^{-15} name_1$
`unit` atto$name_1$ atto$name_2$ a$name_3 = 10^{-18} name_1$
`unit` zepto$name_1$ zepto$name_2$ z$name_3 = 10^{-21} name_1$
`unit` yocto$name_1$ yocto$name_2$ y$name_3 = 10^{-24} name_1$

where μ is the Unicode character U+00B5 MICRO SIGN. Third, a `dim` declaration and a `unit` or `SI_unit` declaration may be collapsed into a single declaration by writing the `unit` or `SI_unit` declaration in place of the `default` clause in the `dim` declaration and omitting the colon and dimension from the `unit` declaration. Thus

`dim` Length `SI_unit` meter meters m
`dim` Power = Energy/Time `SI_unit` watt watts W = joule/second

is understood to abbreviate

`dim` Length `default` meter; `SI_unit` meter meters m: Length
`dim` Power = Energy/Time `default` watt; `SI_unit` watt watts W: Power = joule/second

In this way the names of the seven SI base units, along with all possible plural and prefixed forms, may be concisely defined as follows:

`dim` Length `SI_unit` meter meters m
`dim` Mass `default` kilogram; `SI_unit` gram grams g: Mass
`dim` Time `SI_unit` second seconds s
`dim` ElectricCurrent `SI_unit` ampere amperes A
`dim` Temperature `SI_unit` kelvin kelvins K
`dim` AmountOfSubstance `SI_unit` mole moles mol
`dim` LuminousIntensity `SI_unit` candela candelas cd

Note the subtle difference in the declaration of Mass that allows the default unit to be kilogram rather than gram.

35.4 Absorbing Units

Syntax:

StaticParam ::= *Id* [*Extends*] [`absorbs unit`]
| `unit` *Id* [: *DimRef*] [`absorbs unit`]

The declaration of a type parameter or a `unit` parameter for a parameterized trait may contain a clause "`absorbs unit`"; at most one static parameter of a trait may have this clause. An instance of a parameterized trait with a static parameter that "`absorbs unit`" may be multiplied or divided by a unit, the result being another instance of that parameterized trait in which the static argument corresponding to the unit-absorbing parameter has been multiplied or divided by the unit.

A few examples should make this clear. Given the declaration

$\texttt{trait}\ \mathrm{Vector}[\![\mathrm{EltType}\ \texttt{extends}\ \mathrm{Number}\ \texttt{absorbs unit}, \texttt{nat}\ \mathit{len}]\!] \ldots \texttt{end}$

then $\mathrm{Vector}[\![\mathrm{Float}, 3]\!]\mathrm{meter}$ means the same as $\mathrm{Vector}[\![\mathrm{Float\ meter}, 3]\!]$, and $\mathrm{Vector}[\![\mathrm{Float}, 3]\!]/\mathrm{second}$ means the same as $\mathrm{Vector}[\![\mathrm{Float}/\mathrm{second}, 3]\!]$. Similarly, given the declaration

$\texttt{trait}\ \mathrm{Float}[\![\texttt{unit}\ U\ \texttt{absorbs unit}, \texttt{nat}\ e, \texttt{nat}\ s]\!] \ldots \texttt{end}$

then $\mathrm{Float}[\![\mathrm{meter}, 11, 53]\!]/\mathrm{second}$ means the same as $\mathrm{Float}[\![\mathrm{meter}/\mathrm{second}, 11, 53]\!]$, and

$$\mathrm{Float}[\![\mathrm{dimensionless}, 8, 24]\!]\mathrm{meter\ kilogram}/\mathrm{second}^2$$

means the same as

$$\mathrm{Float}[\![\mathrm{dimensionless\ meter\ kilogram}/\mathrm{second}^2, 8, 24]\!],$$

which is the same as

$$\mathrm{Float}[\![\mathrm{meter\ kilogram}/\mathrm{second}^2, 8, 24]\!].$$

This is the mechanism by which meaning is given to the multiplication and division of library-defined types by units.

Chapter 36

Support for Domain-Specific Languages

To support syntax for domain-specific languages, and to allow the Fortress language to grow with time, programmers can extend the basic syntax of Fortress in their programs. Such extensions are possible through the use of *syntax expanders*. Syntax expanders must be defined in the top level of a simple component.

36.1 Definitions of Syntax Expanders

Syntax:

Decl	::=	*ExternalSyntax*
ExternalSyntax	::=	`syntax` *OpenExpander Id CloseExpander* $=$ *Expr*
OpenExpander	::=	*Id* | *LeftEncloser*
CloseExpander	::=	*Id* | *RightEncloser* | `end`

The definition of a syntax expander starts with the special reserved word `syntax`, followed by an *opening delimiter*, followed by a *contents parameter*, followed by a *terminating delimiter*, followed by an $=$, and a subexpression. The opening delimiter must be either an identifier or the opening member of an enclosing operator. The contents parameter must be an identifier (see Section 5.15). The terminating delimiter must be either an identifier, the terminating member of an enclosing operator, or the special reserved word `end`. If either the opening delimiter or the closing delimiter is *part* of an enclosing operator, the opening and closing delimiters must both be parts of enclosing operators, and they must match, or it is a static error. Because delimiters conceptually delimit blocks, just as `do` and `end` delimit blocks, delimiters of syntax expanders are rendered as special reserved words. The subexpression of a syntax expander has type $\mathrm{Fortress.Ast.SyntaxTree}$. This $\mathrm{SyntaxTree}$ must be that of a Fortress expression. Here is an example:

$$\texttt{syntax sql } e \texttt{ end} = parseSQL(e)$$

where $parseSQL$ is a function that takes an argument of type $\mathrm{Fortress.Lang.SourceAssembly}$ (a sequence of Unicode characters and abstract syntax trees), interprets it as an SQL query, and returns an expression with type $\mathrm{SyntaxTree}$ consisting of constructor calls to SQL syntax nodes (defined in some SQL library).

At a use site, all characters between the opening delimiter and the terminating delimiter are turned into a $\mathrm{SourceAssembly}$ (see Section 36.4 for a more detailed description of how this conversion is achieved). The resulting $\mathrm{SourceAssembly}$ is bound to the contents parameter of the syntax expander. The use site is then expanded by evaluating the body of the expander. Every use site of a syntax expander must occur in an expression context, or it is a static error.

For example, we could define $parseSQL$ so that a use site such as:

```
sql
  SELECT spectral_class FROM stars
end
```

would be expanded into the following Fortress SyntaxTree:

```
Call(Empty,
     List(VarRef(Identifier("SqlQuery")),
          Call(Empty,
               List(VarRef(Identifier("Select")),
                    String("spectral_class"))),
          Call(Empty,
               List(VarRef(Identifier("From")),
                    String("stars")))))
```

(The Empty lists passed to Calls are the lists of static parameters to these calls). Note that this SyntaxTree corresponds to the following Fortress concrete syntax:

```
SqlQuery(Select("spectral_class"), From("stars"))
```

36.2 Declarations of Syntax Expanders

Syntax:

AbsDecl	::=	*AbsExternalSyntax*
AbsExternalSyntax	::=	`syntax` *OpenExpander Id CloseExpander*

A *declaration* of a syntax expander is syntactically identical to the definition of a syntax expander, except that $=$ and the body of the expander (i.e., the expression following the $=$ sign in an expander definition) are elided. Syntax expander declarations must occur only in APIs. A component that exports an API A must provide, for each syntax expander declaration d in A, a syntax expander definition with a header identical to d.

36.3 Restrictions on Delimiters

Consider the set S of syntax expander declarations imported by a component or API B, along with the syntax expanders defined or declared in B directly. Every expander in S must have a distinct opening delimiter, or it is a static error. Moreover, the terminating delimiter of each syntax expander must be distinct from every opening delimiter of every syntax expander in S, or it is a static error.

36.4 Processing Syntax Expanders

In a given component, only syntax expander declarations appearing in APIs imported by the component may be used. A component exporting an API A that includes a syntax expander must not import APIs that are not also imported by A, or it is a static error. This restriction ensures that all names in scope of the definition of the syntax expander are also in scope of any component importing the syntax expander in A. Additionally, all use sites of a syntax expander must occur in simple components other than the defining component. Furthermore, multiple components containing syntax expanders must not be cyclically linked. These restrictions avoid pathologies with nontermination during expansion. Finally, to maintain proper separation of test code, a syntax expander definition is statically forbidden from referring to variables or functions that have modifier `test` .

Because syntax expanders are defined at the top level of program components, and because they are syntactically distinguished, they can be identified before scanning or parsing (but after ASCII conversion). Use sites are then identified and expanded before parsing occurs.

Syntax expansion takes a sequence of Unicode characters and yields a sequence of Unicode characters and syntax trees, where all the syntax expanders have been replaced by syntax trees. This result has type $\mathrm{SourceAssembly}$. Syntax expansion proceeds from left to right as follows. First, the source is scanned for opening delimiters of syntax expanders, stopping at the leftmost one. We call this scanning *Fortress-source scanning*.

If the opening delimiter of a syntax expander is encountered during Fortress-source scanning, the source is scanned rightward until the first occurrence of either an opening delimiter of some syntax expander (possibly another use of the same expander), or the terminating delimiter of that syntax expander, is found. (If no matching terminating delimiter is found in the remainder of the program, it is a static error.) We call this scanning *expander scanning*.

If an opening delimiter is encountered before the terminating delimiter, there is a *nested use site* of another syntax expander. The nested use site is processed, and then expander scanning continues rightward of that use site. Thus, the processing of syntax expanders is recursive, syntax expanders may be nested arbitrarily, and expanders are processed from the leftmost-innermost occurrence outward and rightward.

When the terminating delimiter is encountered during expander scanning, the scanning is terminated, the resulting $\mathrm{SourceAssembly}$ is bound to the contents parameter of the expander, and then the body of the expander is evaluated. The result of evaluating this body has type $\mathrm{SyntaxTree}$, and it is placed into the resulting $\mathrm{SourceAssembly}$ in place of the expander in the scanned source.

36.4.1 Introduced Variable Names

Often, when expanding concrete syntax for a domain-specific language, it is useful to introduce variable binding constructs into the resulting $\mathrm{SyntaxTree}$. It is required that such bindings, in general, respect the rules of hygiene and referential transparency [7]. Several aspects of the Fortress semantics allow the library programmer to ensure referential transparency of syntax expanders:

1. A component exporting an API A that includes a syntax expander must not import APIs that are not also imported by A.
2. Syntax expanders must expand to $\mathrm{SyntaxTrees}$ of expressions.
3. Shadowing of identifiers is not allowed.

Thus, syntax expanders cannot expand into new top-level identifiers. Moreover, provided that the library programmer is careful to ensure that all top-level identifier references appearing in expanded code are fully qualified identifiers exported by APIs, all top-level identifiers referred to in the definition of a syntax expander are visible at all use sites.

To ensure hygiene, all variables bound in a $\mathrm{SyntaxTree}$ resulting from an expansion are renamed, following the `syntax-case` system of Dybvig et al. [9].

36.4.2 Comments and Syntax Expansion

Comments are *not* recognized before syntax expansion (the embedded syntax may have its own commenting syntax): during expander scanning, an opening or terminating delimiter that occurs in what appears to be a Fortress comment is nonetheless recognized as an opening or terminating delimiter for that expander. Comments can be viewed as uses of a syntax expander, with special opening and terminating delimiters.

36.5 Expanders for Fortress

As the above examples demonstrate, it is often useful to denote Fortress abstract syntax using Fortress concrete syntax. A special set of syntax expanders are defined in the API Fortress.Syntax for every nonterminal in Fortress grammar, defined in Appendix G. The name of each expander consists of the name of the nonterminal in lowercase. The terminating symbol for each nonterminal is the special reserved word `end`. For example:

```
expr
  x + y
end
```

expands to the SyntaxTree:

```
Call(Empty,
     VarRef(Identifier("+")),
     VarRef(Identifier("x")),
     VarRef(Identifier("y")))
```

When one of these syntax expanders parses a binding construct, the bound identifier is replaced with an identifier resulting from a call to the function *gensym*, which yields an identifier distinct from all other identifiers bound in any component installed in the same fortress, or in any other fortress, anywhere, throughout all time past, present, and future. All variable references captured by the original identifier are replaced with references to the new identifier.

For convenience, a syntax expander with opening delimiter $\ll$ and terminating delimiter $\gg$ behaves identically to the `expr` expander when Fortress.Syntax is imported.

Part V

Fortress APIs and Documentation for Library Writers

Chapter 37

Algebraic Constraints

The traits in this component are used to describe properties of traits and their associated operators. These traits provide very few concrete methods, but specify abstract methods and `property` declarations. For this reason, the complete code for these traits is presented here, rather than just the APIs.

37.1 Predicates and Equivalence Relations

A *predicate* is an operator that produces a boolean result. A binary predicate may be identified with a mathematical relation, where the predicate returns $true$ in exactly those cases that its two operands satisfy the relation; therefore we use the mathematical terminology usually associated with relations to describe the properties of binary predicates.

```
trait UnaryPredicate⟦T extends UnaryPredicate⟦T, ∼⟧, opr ∼⟧
  opr ∼(self): Boolean
end
```

A unary predicate is a prefix operator that takes one argument and returns a boolean value ($true$ or $false$). Note that $\sim$ is a static parameter, used here as a "variable" name for an operator.

```
trait BinaryPredicate⟦T extends BinaryPredicate⟦T, ∼⟧, opr ∼⟧
  opr ∼(self, other: T): Boolean
end
```

A binary predicate is an infix operator that takes two arguments and returns a boolean value. Thus, for example, any trait T that extends $\mathrm{BinaryPredicate}[\![T, \sqsubset]\!]$ necessarily has an infix method for the operator $\sqsubset$, and that operator returns a boolean value.

```
trait Reflexive⟦T extends Reflexive⟦T, ∼⟧, opr ∼⟧
      extends { BinaryPredicate⟦T, ∼⟧ }
  property ∀(a: T) (a ∼ a)
end
```

A *reflexive* predicate always returns $true$ when its operands are the same. Because this fact is expressed as a `property` declaration, the behavior of such an operator can be checked for correctness by unit testing.

```
trait Irreflexive⟦T extends Irreflexive⟦T, ∼⟧, opr ∼⟧
      extends { BinaryPredicate⟦T, ∼⟧ }
```

```
  property ∀(a: T) ¬(a ∼ a)
end
```

An *irreflexive* predicate always returns $false$ when its operands are the same. (Note that it is possible for a predicate to be neither reflexive nor irreflexive.)

```
trait Symmetric⟦T extends Symmetric⟦T, ∼⟧, opr ∼⟧
      extends { BinaryPredicate⟦T, ∼⟧ }
  property ∀(a: T, b: T) (a ∼ b) ↔ (b ∼ a)
end
```

A *symmetric* predicate doesn't care in which order its arguments are presented; the result is the same either way.

```
trait Transitive⟦T extends Transitive⟦T, ∼⟧, opr ∼⟧
      extends { BinaryPredicate⟦T, ∼⟧ }
  property ∀(a: T, b: T, c: T) ((a ∼ b) ∧ (b ∼ c)) → (a ∼ c)
end
```

A *transitive* predicate has the property that if a is related to b and b is related to c, then a is related to c.

```
trait EquivalenceRelation⟦T extends EquivalenceRelation⟦T, ∼⟧, opr ∼⟧
       extends { Reflexive⟦T, ∼⟧, Symmetric⟦T, ∼⟧, Transitive⟦T, ∼⟧ }
end
```

An *equivalence relation* is any predicate that is reflexive, symmetric, and transitive. You can think of an equivalence relation as describing a way to separate a set of items into categories, such that each item belongs to exactly one category; the predicate is $true$ of two items if and only if they are in the same category.

```
trait IdentityEquality⟦T extends IdentityEquality⟦T⟧⟧
      extends { EquivalenceRelation⟦T, =⟧ }
  opr =(self, other: T): Boolean = (self ≡ other)
end
```

This trait provides a concrete implementation of the operator $=$ (defining it to behave the same as the operator $\equiv$) and states that $=$ is an equivalence relation over instances of the type T.

```
trait UnaryPredicateSubstitutionLaws⟦T extends UnaryPredicateSubstitutionLaws⟦T, ∼, ≃⟧,
                                     opr ∼, opr ≃⟧
      extends { UnaryPredicate⟦T, ∼⟧, BinaryPredicate⟦T, ≃⟧ }
  property ∀(a: T, a′: T) (a ≃ a′) →: ((∼ a) ↔ (∼ a′))
end
```

This handy trait states that the unary predicate $\sim$ is consistent under substitutions described by the relation $\simeq$ (which is typically, but not always, an equivalence relation); that is, the result produced by $\sim$ is unchanged if its argument is replaced by some other value that is equivalent.

```
trait BinaryPredicateSubstitutionLaws⟦T extends BinaryPredicateSubstitutionLaws⟦T, ∼, ≃⟧,
                                     opr ∼, opr ≃⟧
      extends { BinaryPredicate⟦T, ∼⟧, BinaryPredicate⟦T, ≃⟧ }
  property ∀(a: T, a′: T) (a ≃ a′) →: ∀(b: T) (a ∼ b) ↔ (a′ ∼ b)
  property ∀(b: T, b′: T) (b ≃ b′) →: ∀(a: T) (a ∼ b) ↔ (a ∼ b′)
end
```

This equally handy trait states that the binary predicate $\sim$ is consistent under substitutions described by the relation $\simeq$ (which is typically, but not always, an equivalence relation); that is, the result produced by $\sim$ is unchanged if either

argument is replaced by some other value that is equivalent. (It is then easy to prove that the result is unchanged even when *both* arguments are replaced by equivalent values.)

37.2 Partial and Total Orders

```
trait Antisymmetric⟦T extends AntiSymmetric⟦T, ∼⟧, opr ∼⟧
      extends { BinaryPredicate⟦T, ∼⟧, EquivalenceRelation⟦T, =⟧,
                BinaryPredicateSubstitutionLaws⟦T, ∼, =⟧ }
   property ∀(a: T, b: T) ((a ∼ b) ∧ (b ∼ a)) :→: (a = b)
end
```

A binary predicate $\sim$ is *antisymmetric* if and only if two conditions are true: (a) $\sim$ is consistent under substitutions described by the predicate $=$, which must be an equivalence relation; (b) whenever $\sim$ holds true for a pair of arguments and for those same arguments in reverse order, those arguments are equivalent as specified by $=$.

```
trait PartialOrder⟦T extends PartialOrder⟦T, ≼⟧, opr ≼⟧
      extends { Reflexive⟦T, ≼⟧, Antisymmetric⟦T, ≼⟧, Transitive⟦T, ≼⟧ }
end
```

A *partial order* is a binary predicate that is reflexive, antisymmetric, and transitive.

```
trait StrictPartialOrder⟦T extends StrictPartialOrder⟦T, ≺⟧, opr ≺⟧
      extends { Irreflexive⟦T, ≺⟧, Antisymmetric⟦T, ≺⟧, Transitive⟦T, ≺⟧ }
end
```

A *strict partial order* is a binary predicate that is irreflexive, antisymmetric, and transitive. (Thus it differs from an ordinary partial order in being irreflexive rather than reflexive.)

It is easy to prove that, because $\prec$ is irreflexive and antisymmetric, that $a \prec b$ and $a = b$ cannot both be true. (If they were both true, then because antisymmetry requires that $\prec$ obey substitution laws, $a \prec a$ would be true—but that contradicts the fact that $\prec$ is irreflexive.) It is then easy to prove that $a \prec b$ and $b \prec a$ cannot both be true. (If they were both true, then by antisymmetry $a = b$ must be true—but $a \prec b$ and $a = b$ cannot both be true.)

```
trait TotalOrder⟦T extends TotalOrder⟦T, ≼⟧, opr ≼⟧
      extends { PartialOrder⟦T, ≼⟧ }
   property ∀(a: T, b: T) (a ≼ b) ∨ (b ≼ a)
end
```

A *total order* is a partial order in which every pair of operands must be related by the predicate $\preccurlyeq$ in one order or the other; thus no two values are ever unordered with respect to each other.

```
trait StrictTotalOrder⟦T extends StrictTotalOrder⟦T, ≺⟧, opr ≺⟧
      extends { StrictPartialOrder⟦T, ≺⟧ }
   property ∀(a: T, b: T) (a ≺ b) ∨ (b ≺ a) ∨ (a = b)
end
```

A *strict total order* is a strict partial order in which every pair of operands must be related, either by equality $=$ or by the predicate $\prec$ in one order or the other; thus no two values are ever unordered with respect to each other.

For a strict total order *at least* one of $a \prec b$ and $a = b$ and $b \prec a$ is true; but a strict total order is also a strict partial order, for which *at most* one of $a \prec b$ and $a = b$ and $b \prec a$ is true. Therefore a strict total order obeys the *law of trichotomy*: for any a and b, *exactly* one of $a \prec b$ and $a = b$ and $b \prec a$ is true.

```
trait PartialOrderOperators⟦T extends PartialOrderOperators⟦T, ≺, ≼, ≽, ≻, CMP⟧,
                                opr ≺, opr ≼, opr ≽, opr ≻, opr CMP⟧
    extends { StrictPartialOrder⟦T, ≺⟧, PartialOrder⟦T, ≼⟧,
              PartialOrder⟦T, ≽⟧, StrictPartialOrder⟦T, ≻⟧ }
  opr CMP(self, other: T): Comparison
  opr ≺(self, other: T): Boolean = (a CMP b) ≡ LessThan
  opr ≼(self, other: T): Boolean = (a CMP b) ≡ LessThan ∨ (a CMP b) ≡ EqualTo
  opr =(self, other: T): Boolean = (a CMP b) ≡ EqualTo
  opr ≽(self, other: T): Boolean = (a CMP b) ≡ GreaterThan ∨ (a CMP b) ≡ EqualTo
  opr ≻(self, other: T): Boolean = (a CMP b) ≡ GreaterThan
  property ∀(a: T, b: T) ((a CMP b) ≡ LessThan) ↔ ((b CMP a) ≡ GreaterThan)
  property ∀(a: T, b: T) ((a CMP b) ≡ EqualTo) ↔ ((b CMP a) ≡ EqualTo)
  property ∀(a: T, b: T) ((a CMP b) ≡ Unordered) ↔ ((b CMP a) ≡ Unordered)
  property ∀(a: T, b: T) (a ≺ b) ↔ ((a CMP b) ≡ LessThan)
  property ∀(a: T, b: T) (a ≼ b) ↔ ((a CMP b) ≡ LessThan ∨ (a CMP b) ≡ EqualTo)
  property ∀(a: T, b: T) (a = b) ↔ ((a CMP b) ≡ EqualTo)
  property ∀(a: T, b: T) (a ≽ b) ↔ ((a CMP b) ≡ GreaterThan ∨ (a CMP b) ≡ EqualTo)
  property ∀(a: T, b: T) (a ≻ b) ↔ ((a CMP b) ≡ GreaterThan)
end
```

For practical programming, we assume that partial order operators come in groups of five: a "less than" predicate, a "less than or equal to" predicate, a "greater than or equal to" predicate, a "greater than" predicate, and a "comparison" operator that returns one of the four values LessThan, EqualTo, GreaterThan, and Unordered. The trait PartialOrderOperators declares such a set of five operators and describes the necessary algebraic constraints among them and the equality predicate $=$. It also provides default definitions for the predicates (including $=$) in terms of the comparison operator.

```
trait TotalOrderOperators⟦T extends TotalOrderOperators⟦T, ≺, ≼, ≽, ≻, CMP⟧,
                              opr ≺, opr ≼, opr ≽, opr ≻, opr CMP⟧
    extends { PartialOrderOperators⟦T, ≺, ≼, ≽, ≻, CMP⟧,
              StrictTotalOrder⟦T, ≺⟧, TotalOrder⟦T, ≼⟧,
              TotalOrder⟦T, ≽⟧, StrictTotalOrder⟦T, ≻⟧ }
  opr CMP(self, other: T): TotalComparison
end
```

Total order operators likewise come in groups of five: a "less than" predicate, a "less than or equal to" predicate, a "greater than or equal to" predicate, a "greater than" predicate, and a "comparison" operator that returns one of the three values LessThan, EqualTo, and GreaterThan. The trait TotalOrderOperators declares such a set of five operators and describes the necessary algebraic constraints among them and the equality predicate $=$. For a total order, the comparison operator CMP never returns Unordered.

```
trait PartialOrderBasedOnLE⟦T extends PartialOrderBasedOnLE⟦T, ≺, ≼, ≽, ≻, CMP⟧,
                                opr ≺, opr ≼, opr ≽, opr ≻, opr CMP⟧
    extends { PartialOrderOperators⟦T, ≺, ≼, ≽, ≻, CMP⟧ }
  opr CMP(self, other: T): Comparison =
    if a ≼ b then
      if b ≼ a then EqualTo else LessThan end
    else
      if b ≼ a then GreaterThan else Unordered end
    end
  opr ≺(self, other: T): Boolean = (self ≼ other) ∧ ¬(other ≼ self)
  opr =(self, other: T): Boolean = (self ≼ other) ∧ (other ≼ self)
  opr ≽(self, other: T): Boolean = (other ≼ self)
```

```
    opr ≻(self, other: T): Boolean = (other ≺ self)
end
```

The trait PartialOrderBasedOnLE specifies a partial order by assuming that the "less than or equal to" predicate is already defined and providing definitions for the comparison operator, the "less than" predicate, the equality predicate, the "greater than or equal to" predicate, and the "greater than" predicate in terms of the "less than or equal to" predicate.

```
trait TotalOrderBasedOnLE⟦T extends TotalOrderBasedOnLE⟦T, ≺, ≼, ≽, ≻, CMP⟧,
                                    opr ≺, opr ≼, opr ≽, opr ≻, opr CMP⟧
      extends { TotalOrderOperators⟦T, ≺, ≼, ≽, ≻, CMP⟧ }
    opr CMP(self, other: T): TotalComparison =
      if a ≼ b then (if b ≼ a then EqualTo else LessThan end) else GreaterThan end
    opr ≺(self, other: T): Boolean = ¬(other ≼ self)
    opr =(self, other: T): Boolean = (self ≼ other) ∧ (other ≼ self)
    opr ≽(self, other: T): Boolean = (other ≼ self)
    opr ≻(self, other: T): Boolean = (other ≺ self)
end
```

The trait TotalOrderBasedOnLE specifies a total order by assuming that the "less than or equal to" predicate is already defined and providing definitions for the comparison operator, the "less than" predicate, the equality predicate, the "greater than or equal to" predicate, and the "greater than" predicate in terms of the "less than or equal to" predicate.

```
trait TotalOrderBasedOnLT⟦T extends TotalOrderBasedOnLT⟦T, ≺, ≼, ≽, ≻, CMP⟧,
                                    opr ≺, opr ≼, opr ≽, opr ≻, opr CMP⟧
      extends { TotalOrderOperators⟦T, ≺, ≼, ≽, ≻, CMP⟧ }
    opr CMP(self, other: T): TotalComparison =
      if a ≺ b then LessThan elif b ≺ a then GreaterThan else EqualTo end
    opr ≼(self, other: T): Boolean = ¬(other ≺ self)
    opr =(self, other: T): Boolean = (self ≼ other) ∧ (other ≼ self)
    opr ≽(self, other: T): Boolean = (other ≼ self)
    opr ≻(self, other: T): Boolean = (other ≺ self)
end
```

The trait TotalOrderBasedOnLT specifies a total order by assuming that the "less than" predicate is already defined and providing definitions for the comparison operator, the "less than or equal to" predicate, the equality predicate, the "greater than or equal to" predicate, and the "greater than" predicate in terms of the "less than" predicate.

```
value object MaximalElement⟦opr ≼⟧ end

trait HasMaximalElement⟦T extends HasMaximalElement⟦T, ≼⟧, opr ≼⟧
      extends { PartialOrder⟦T, ≼⟧ }
      where { T coerces MaximalElement⟦≼⟧ }
    property ∀(a : T) a ≼ MaximalElement⟦≼⟧
end
```

The HasMaximalElement trait specifies that a partial order has a maximal element, that is, one to which every other element is related by the ordering predicate $\preccurlyeq$. The maximal element may be identified by coercing the object named MaximalElement⟦≼⟧ to type T.

```
value object MinimalElement⟦opr ≼⟧ end

trait HasMinimalElement⟦T extends HasMinimalElement⟦T, ≼⟧, opr ≼⟧
      extends { PartialOrder⟦T, ≼⟧ }
      where { T coerces MinimalElement⟦≼⟧ }
    property ∀(a : T) MinimalElement⟦≼⟧ ≼ a
end
```

The HasMinimalElement trait specifies that a partial order has a minimal element, that is, one to which every other element is related by the ordering predicate $\preccurlyeq$. The minimal element may be identified by coercing the object named MinimalElement$[\![\preceq]\!]$ to type T.

```
trait LexicographicPartialOrder[T extends LexicographicPartialOrder[T, ⊏, ⊑, ≡, ⊒, ⊐, TCMP,
                                                                     X, ≺, ≼, ≃, ≽, ≻, XCMP],
                                opr ⊏, opr ⊑, opr ≡, opr ⊒, opr ⊐, opr TCMP,
                                X extends TotalOrderOperators[X, ≺, ≼, ≃, ≽, ≻, XCMP],
                                opr ≺, opr ≼, opr ≃, opr ≽, opr ≻, opr XCMP]
    extends { PartialOrderOperators[LexicographicPartialOrder[T, ⊏, ⊑, ≡, ⊒, ⊐, TCMP,
                                                               X, ≺, ≼, ≃, ≽, ≻, XCMP],
                                    ⊏, ⊑, ⊒, ⊐, TCMP] }
        where { T extends ZeroBasedIndexing[T, X] }
  opr TCMP(self, other: LexicographicPartialOrder[T, ⊏, ⊑, ≡, ⊒, ⊐, TCMP, X, ≺, ≼, ≃, ≽, ≻, XCMP]) :
        Comparison =
    ( BIG LEXICO (self_i XCMP other_i)) LEXICO (|self| CMP |other|)
                i←self.indices∩other.indices
end
```

The Comparison trait provides an associative operator LEXICO whose principal use is in defining lexicographic order on sequences of elements, which may be partial or total depending on whether the ordering of the elements is partial or total.

A set of lexicographic partial order operators $\sqsubset, \sqsubseteq, \equiv, \sqsupseteq, \sqsupset,$ TCMP may be defined in terms of a partial order on the elements of the sequence with operators $\prec, \preccurlyeq, \simeq, \succcurlyeq, \succ,$ XCMP. All that is really necessary is to define the lexicographic sequence comparison operator TCMP in terms of the element comparison operator XCMP; this is easily expressed by using the associative LEXICO operator to reduce the results of elementwise comparisons to a single value. (If the sequences to be compared are of unequal length, then the shorter sequence is compared to a prefix of the longer sequence, and if they are equal, then the longer sequence is considered to be greater than the shorter sequence. This rule is implemented by an additional application of the LEXICO operator to the result of comparing the lengths of the sequences.)

```
trait LexicographicTotalOrder[T extends LexicographicTotalOrder[T, ⊏, ⊑, ≡, ⊒, ⊐, TCMP,
                                                                 X, ≺, ≼, ≃, ≽, ≻, XCMP],
                              opr ⊏, opr ⊑, opr ≡, opr ⊒, opr ⊐, opr TCMP,
                              X extends TotalOrderOperators[X, ≺, ≼, ≃, ≽, ≻, XCMP],
                              opr ≺, opr ≼, opr ≃, opr ≽, opr ≻, opr XCMP]
    extends { LexicographicPartialOrder[T, ⊏, ⊑, ≡, ⊒, ⊐, TCMP, X, ≺, ≼, ≃, ≽, ≻, XCMP],
              TotalOrderBasedOnLE[LexicographicTotalOrder[T, ⊏, ⊑, ≡, ⊒, ⊐, TCMP,
                                                          X, ≺, ≼, ≃, ≽, ≻, XCMP],
                                  ⊏, ⊑, ⊒, ⊐, TCMP] }
  opr TCMP(self, other: LexicographicTotalOrder[T, ⊏, ⊑, ≡, ⊒, ⊐, TCMP, X, ≺, ≼, ≃, ≽, ≻, XCMP]) :
        TotalComparison =
    ( BIG LEXICO (self_i XCMP other_i)) LEXICO (|self| CMP |other|)
                i←self.indices∩other.indices
end
```

Similarly, a set of lexicographic total order operators $\sqsubset, \sqsubseteq, \equiv, \sqsupseteq, \sqsupset,$ TCMP may be defined in terms of a total order on the elements of the sequence with operators $\prec, \preccurlyeq, \simeq, \succcurlyeq, \succ,$ XCMP.

37.3 Operators and Their Properties

For some types, such as the integers $\mathbb{Z}$ or the rationals $\mathbb{Q}$, results are always exact, and algebraic properties can be expected to be obeyed exactly. For other types, such as floating-point numbers, results are not always numerically exact, and algebraic properties can be expected to be obeyed only approximately. For example, given three floating-point values a and b and c, it may well be that $a + (b + c)$ is not equal to $(a + b) + c$; but we would expect their values to be reasonably close—unless, of course, overflow occurred in one expression but not the other.

In order to address the difficulties of such approximate computation, many of the traits described in this section come in two varieties: approximate and exact. The $+$ operator on integers or rationals is correctly described by the trait $\mathrm{Associative}$, and the $+$ operator on floating-point numbers is correctly described by the trait $\mathrm{ApproximatelyAssociative}$. An important distinction is that the predicate used to test acceptability of exact algebraic properties is $=$, which is required to be an equivalence relation and therefore transitive, but a type-dependent binary predicate (typically $\approx$) may be used to test acceptability of approximate algebraic properties, and this predicate is required only to be reflexive and symmetric.

```
trait UnaryOperator⟦T extends UnaryOperator⟦T, ⊙⟧, opr ⊙⟧
  opr ⊙(self): T
end
```

A *unary operator* is a prefix operator that takes one argument and returns a value of the same type. Note that $\odot$ is a static parameter, used here as a "variable" name for an operator.

```
trait BinaryOperator⟦T extends BinaryOperator⟦T, ⊙⟧, opr ⊙⟧
  opr ⊙(self, other: T): T
end
```

A *binary operator* is an infix operator that takes two arguments of the same type and returns a value of that type. Thus, for example, any trait T that extends $\mathrm{BinaryPredicate}[\![T, +]\!]$ necessarily has an infix method for the operator $+$, and that operator takes two operands of type T and returns a value of type T.

```
trait IdentityOperator⟦T extends IdentityOperator⟦T⟧⟧
      extends { UnaryOperator⟦T, IDENTITY⟧ }
  opr IDENTITY(self): T = self
  property ∀(a: T) (IDENTITY a) ≡ a
end
```

The trait $\mathrm{IdentityOperator}$ provides a definition of the unary `IDENTITY` operator, which simply returns its argument. The trait Object extends $\mathrm{IdentityOperator}[\![\mathrm{Object}]\!]$, so the `IDENTITY` operator is defined for every type whatsoever. (This operator may not be terribly useful for applications programming, but it has technical uses for specifying contracts and algebraic properties in libraries. It is used, for example, when defining the trait $\mathrm{BooleanAlgebra}$ in terms of the trait Ring: because every value is its own inverse with respect to the "exclusive OR" operator in a Boolean Algebra, `IDENTITY` is the appropriate additive inverse operator for use with the trait Ring in this connection.)

```
trait ApproximatelyCommutative⟦T extends ApproximatelyCommutative⟦T, ⊙, ≈⟧, opr ⊙, opr ≈⟧
      extends { BinaryOperator⟦T, ⊙⟧, Reflexive⟦T, ≈⟧, Symmetric⟦T, ≈⟧ }
  property ∀(a: T, b: T) (a ⊙ b) :≈: (b ⊙ a)
end
```

The trait $\mathrm{ApproximatelyCommutative}$ requires the operator $\odot$ to be *approximately commutative*; that is, reversing the operands produces a result that is considered to be "close enough" as determined by the specified $\approx$ predicate.

```
trait Commutative⟦T extends Commutative⟦T, ⊙⟧, opr ⊙⟧
      extends { ApproximatelyCommutative⟦T, ⊙, =⟧, EquivalenceRelation⟦T, =⟧ }
end
```

The trait Commutative requires the operator $\odot$ to be *commutative*; that is, reversing the operands produces an equal result.

```
trait ApproximatelyAssociative⟦T extends ApproximatelyAssociative⟦T, ⊙, ≈⟧, opr ⊙, opr ≈⟧
      extends { BinaryOperator⟦T, ⊙⟧, Reflexive⟦T, ≈⟧, Symmetric⟦T, ≈⟧ }
   property ∀(a: T, b: T, c: T) ((a ⊙ b) ⊙ c) :≈: (a ⊙ (b ⊙ c))
end
```

The trait ApproximatelyAssociative requires the operator $\odot$ to be *approximately associative*; that is, the expressions $(a \odot b) \odot c$ and $a \odot (b \odot c)$ always produce results that are "close enough" to each other as determined by the specified $\approx$ predicate.

```
trait Associative⟦T extends Associative⟦T, ⊙⟧, opr ⊙⟧
      extends { ApproximatelyAssociative⟦T, ⊙, =⟧, EquivalenceRelation⟦T, =⟧ }
end
```

The trait Associative requires the operator $\odot$ to be *associative*; that is, the expressions $(a \odot b) \odot c$ and $a \odot (b \odot c)$ always produce equal results.

```
trait IdempotentBinaryOperator⟦T extends IdempotentBinaryOperator⟦T, ⊙⟧, opr ⊙⟧
      extends { BinaryOperator⟦T, ⊙⟧, EquivalenceRelation⟦T, =⟧ }
   property ∀(a: T) (a ⊙ a) :=: a
end
```

An idempotent binary operator has the property that if its two arguments are the same then the result is equal to each argument. For example, `MAX` and `MIN` are idempotent, as are $\wedge$ and $\vee$ applied to boolean arguments and $\cap$ and $\cup$ applied to sets; but $+$ applied to integers is not idempotent because $1 + 1$ does not produce 1, and $\oplus$ applied to boolean arguments is not idempotent because $true \oplus true$ produces $false$. The property of idempotency is sometimes of interest when performing reductions such as $\underset{i \leftarrow 1:n}{\texttt{MAX}}\ a_i$.

```
trait HasLeftZeroes⟦T extends HasLeftZeroes⟦T, ⊙, isLeftZero⟧, opr ⊙, ident isLeftZero⟧
      extends { BinaryOperator⟦T, ⊙⟧ }
   isLeftZero(): Boolean
   property ∀(a: T, b: T) a.isLeftZero() →: ((a ⊙ b) = a)
end
```

A value e is a *left zero* for a binary operator $\odot$ if the result of $\odot$ always equals e whenever e is the left-hand operand. For example, $-\infty$ is a left zero for the `MIN` operator on floating-point values, and $\texttt{7FFFFFFF}_{16}$ is a left zero for the `MAX` operator on values of type $\mathbb{Z}32$. The purpose of this trait is to specify a method that says whether a given element is a left zero for $\odot$.

```
trait HasRightZeroes⟦T extends HasRightZeroes⟦T, ⊙, isRightZero⟧, opr ⊙, ident isRightZero⟧
      extends { BinaryOperator⟦T, ⊙⟧ }
   isRightZero(): Boolean
   property ∀(a: T, b: T) b.isRightZero() →: ((a ⊙ b) = b)
end
```

A value e is a *right zero* for a binary operator $\odot$ if the result of $\odot$ always equals e whenever e is the right-hand operand. For example, $-\infty$ is a right zero (as well as a left zero) for the `MIN` operator on floating-point values, and $\texttt{7FFFFFFF}_{16}$ is a right zero (as well as a left zero) for the `MAX` operator on values of type $\mathbb{Z}32$. By way of contrast, 0 is a left zero for the arithmetic shift operator on integers, but is not a right zero. The purpose of this trait is to specify a method that says whether a given element is a right zero for $\odot$.

```
trait ApproximatelyLeftDistributive⟦T extends ApproximatelyLeftDistributive⟦T, ⊗, ⊕, ≈⟧,
```

$$\texttt{opr}\ \otimes, \texttt{opr}\ \oplus, \texttt{opr}\ \approx]\!]$$

$\texttt{extends}\ \{\ \text{BinaryOperator}[\![T, \otimes]\!], \text{BinaryOperator}[\![T, \oplus]\!], \text{Reflexive}[\![T, \approx]\!], \text{Symmetric}[\![T, \approx]\!]\ \}$

$\texttt{property}\ \forall(a\colon T, b\colon T, c\colon T)\ (a \otimes (b \oplus c)) :\approx: ((a \otimes b) \oplus (a \otimes c))$

$\texttt{end}$

The trait ApproximatelyLeftDistributive requires the operator $\otimes$ to be *approximately left distributive* over the operator $\oplus$; that is, the expressions $a \otimes (b \oplus c)$ and $(a \otimes b) \oplus (a \otimes c)$ always produce results that are "close enough" to each other as determined by the specified $\approx$ predicate.

$\texttt{trait}\ \text{LeftDistributive}[\![T\ \texttt{extends}\ \text{LeftDistributive}[\![T, \otimes, \oplus]\!], \texttt{opr}\ \otimes, \texttt{opr}\ \oplus]\!]$

$\texttt{extends}\ \{\ \text{ApproximatelyLeftDistributive}[\![T, \otimes, \oplus, =]\!], \text{EquivalenceRelation}[\![T, =]\!]\ \}$

$\texttt{end}$

The trait LeftDistributive requires the operator $\otimes$ to be *left distributive* over the operator $\oplus$; that is, the expressions $a \otimes (b \oplus c)$ and $(a \otimes b) \oplus (a \otimes c)$ always produce equal results.

$\texttt{trait}\ \text{ApproximatelyRightDistributive}[\![T\ \texttt{extends}\ \text{ApproximatelyRightDistributive}[\![T, \otimes, \oplus, \approx]\!],$
$\texttt{opr}\ \otimes, \texttt{opr}\ \oplus, \texttt{opr}\ \approx]\!]$

$\texttt{extends}\ \{\ \text{BinaryOperator}[\![T, \otimes]\!], \text{BinaryOperator}[\![T, \oplus]\!], \text{Reflexive}[\![T, \approx]\!], \text{Symmetric}[\![T, \approx]\!]\ \}$

$\texttt{property}\ \forall(a\colon T, b\colon T, c\colon T)\ ((a \oplus b) \otimes c) :\approx: ((a \otimes c) \oplus (b \otimes c))$

$\texttt{end}$

The trait ApproximatelyRightDistributive requires the operator $\otimes$ to be *approximately right distributive* over the operator $\oplus$; that is, the expressions $(a \oplus b) \otimes c$ and $(a \otimes c) \oplus (b \otimes c)$ always produce results that are "close enough" to each other as determined by the specified $\approx$ predicate.

$\texttt{trait}\ \text{RightDistributive}[\![T\ \texttt{extends}\ \text{RightDistributive}[\![T, \otimes, \oplus]\!], \texttt{opr}\ \otimes, \texttt{opr}\ \oplus]\!]$

$\texttt{extends}\ \{\ \text{ApproximatelyRightDistributive}[\![T, \otimes, \oplus, =]\!], \text{EquivalenceRelation}[\![T, =]\!]\ \}$

$\texttt{end}$

The trait RightDistributive requires the operator $\otimes$ to be *right distributive* over the operator $\oplus$; that is, the expressions $(a \oplus b) \otimes c$ and $(a \otimes c) \oplus (b \otimes c)$ always produce equal results.

$\texttt{trait}\ \text{ApproximatelyDistributive}[\![T\ \texttt{extends}\ \text{ApproximatelyDistributive}[\![T, \otimes, \oplus, \approx]\!],$
$\texttt{opr}\ \otimes, \texttt{opr}\ \oplus, \texttt{opr}\ \approx]\!]$

$\texttt{extends}\ \{\ \text{ApproximatelyLeftDistributive}[\![T, \otimes, \oplus, \approx]\!],$
$\text{ApproximatelyRightDistributive}[\![T, \otimes, \oplus, \approx]\!]\ \}$

$\texttt{end}$

The trait ApproximatelyDistributive requires the operator $\otimes$ to be both approximately left distributive and approximately right distributive over the operator $\oplus$.

$\texttt{trait}\ \text{Distributive}[\![T\ \texttt{extends}\ \text{Distributive}[\![T, \otimes, \oplus]\!], \texttt{opr}\ \otimes, \texttt{opr}\ \oplus]\!]$

$\texttt{extends}\ \{\ \text{ApproximatelyDistributive}[\![T, \otimes, \oplus, =]\!], \text{LeftDistributive}[\![T, \otimes, \oplus]\!], \text{RightDistributive}[\![T, \otimes, \oplus]\!]\ \}$

$\texttt{end}$

The trait Distributive requires the operator $\otimes$ to be both left distributive and right distributive over the operator $\oplus$.

$\texttt{trait}\ \text{HasLeftIdentity}[\![T\ \texttt{extends}\ \text{HasLeftIdentity}[\![T, \odot]\!], \texttt{opr}\ \odot]\!]$

$\texttt{extends}\ \{\ \text{BinaryOperator}[\![T, \odot]\!], \text{EquivalenceRelation}[\![T, =]\!]\ \}$

$\mathit{isLeftIdentity}()\colon \text{Boolean}$

$\texttt{property}\ \forall(a\colon T, b\colon T)\ a.\mathit{isLeftIdentity}() \rightarrow: ((a \odot\ b) = b)$

$\texttt{end}$

A value e is a *left identity* for a binary operator $\odot$ if the result of $\odot$ always equals the right-hand operand whenever e is the left-hand operand. For example, 0 is a left identity for the $+$ operator on integers and the empty set is a left identity for the $\cup$ operator on sets.

```
trait HasRightIdentity⟦T extends HasRightIdentity⟦T, ⊙⟧, opr ⊙⟧
    extends { BinaryOperator⟦T, ⊙⟧, EquivalenceRelation⟦T, =⟧ }
  isRightIdentity(): Boolean
  property ∀(a: T, b: T) b.isRightIdentity() →: ((a ⊙ b) = a)
end
```

A value e is a *right identity* for a binary operator $\odot$ if the result of $\odot$ always equals the right-hand operand whenever e is the left-hand operand. For example, 0 is a right identity (as well as a left identity) for the $+$ operator on integers and the empty set is a right identity (as well as a left identity) for the $\cup$ operator on sets. By way of contrast, 1 is a right identity for division of rationals but not a left identity.

```
value object Identity⟦opr ⊙⟧ end

trait HasIdentity⟦T extends HasIdentity⟦T, ⊙⟧, opr ⊙⟧
    extends { HasLeftIdentity⟦T, ⊙⟧, HasRightIdentity⟦T, ⊙⟧ }
    where { T coerces Identity⟦⊙⟧ }
  property ∀(a: T) (a ⊙ Identity⟦⊙⟧) = a
  property ∀(a: T) (Identity⟦⊙⟧ ⊙ a) = a
  property ∀(a: T) a.isLeftIdentity() ↔ (a = Identity⟦⊙⟧)
  property ∀(a: T) a.isRightIdentity() ↔ (a = Identity⟦⊙⟧)
end
```

If the same value is both a left identity and a right identity for $\odot$, then it may be called simply an *identity*—in fact, *the* identity, for it is unique and may be obtained by coercing the object named $\mathrm{Identity}⟦\odot⟧$ to type T.

```
value object Zero⟦opr ⊗⟧ end

trait ApproximateZeroAnnihilation⟦T extends ApproximateZeroAnnihilation⟦T, ⊗, ≈⟧,
                                  opr ⊗, opr ≈⟧
    extends { BinaryOperator⟦T, ⊗⟧, Reflexive⟦T, ≈⟧, Symmetric⟦T, ≈⟧ }
  property ∀(a: T) (Zero⟦⊗⟧ ⊗ a) :≈: Zero⟦⊗⟧
  property ∀(a: T) (a ⊗ Zero⟦⊗⟧) :≈: Zero⟦⊗⟧
end
```

An operator $\otimes$ obeys *approximate zero annihilation* if and only if there is an element (call it z) that when used as either operand of $\otimes$ causes the result to be "close enough" to z as determined by the specified $\approx$ predicate. This zero element may be obtained by coercing the object named $\mathrm{Zero}⟦\otimes⟧$ to type T.

```
trait ZeroAnnihilation⟦T extends ZeroAnnihilation⟦T, ⊗⟧, opr ⊗⟧
    extends { ApproximateZeroAnnihilation⟦T, ⊗, =⟧, EquivalenceRelation⟦T, =⟧ }
end
```

An operator $\otimes$ obeys *zero annihilation* if and only if there is an element (call it z) that when used as either operand of $\otimes$ causes the result to equal z. (It follows that z is both a left idempotent element and a right idempotent element for $\otimes$. However, the trait $\mathrm{ZeroAnnihilation}⟦T, \otimes⟧$ intentionally does not extend the traits $\mathrm{HasLeftZeroes}⟦T, \otimes, name⟧$ and $\mathrm{HasRightZeroes}⟦T, \otimes, name⟧$ because there is not always a practical requirement for methods that determine whether any value is left idempotent or right idempotent; sometimes it suffices to know only that one particular element, produced by coercing $\mathrm{Zero}⟦\otimes⟧$ to type T, has that property.)

```
trait UnaryOperatorSubstitutionLaws⟦T extends UnaryOperatorSubstitutionLaws⟦T, ⊙, =⟧,
                                    opr ⊙, opr ≃⟧
```

```
    extends { UnaryOperator⟦T, ⊙⟧, BinaryPredicate⟦T, ≃⟧ }
  property ∀(a: T, a′: T) (a ≃ a′) →: (⊙a) ≃ (⊙a′)
end
```

This peculiarly spiffy trait states that the unary operator $\odot$ is consistent under substitutions described by the relation $\simeq$ (which is typically, but not always, an equivalence relation); that is, the result produced by $\odot$ is unchanged if its argument is replaced by some other value that is equivalent.

```
trait BinaryOperatorSubstitutionLaws⟦T extends BinaryOperatorSubstitutionLaws⟦T, ⊙, =⟧,
                                     opr ⊙, opr ≃⟧
    extends { BinaryOperator⟦T, ⊙⟧, BinaryPredicate⟦T, ≃⟧ }
  property ∀(a: T, a′: T) (a ≃ a′) →: ∀(b: T) (a ⊙ b) ≃ (a′ ⊙ b)
  property ∀(b: T, b′: T) (b ≃ b′) →: ∀(a: T) (a ⊙ b) ≃ (a ⊙ b′)
end
```

This equally spiffy trait states that the binary operator $\odot$ is consistent under substitutions described by the relation $\simeq$ (which is typically, but not always, an equivalence relation); that is, the result produced by $\odot$ is unchanged if either argument is replaced by some other value that is equivalent. (It is then easy to prove that the result is unchanged even when *both* arguments are replaced by equivalent values.)

37.4 Monoids, Groups, Rings, and Fields

```
trait ApproximateMonoid⟦T extends ApproximateMonoid⟦T, ⊙, ≈⟧, opr ⊙, opr ≈⟧
    extends { ApproximatelyAssociative⟦T, ⊙, ≈⟧, HasIdentity⟦T, ⊙⟧ }
end
```

An *approximate monoid* is a set of values with an approximately associative binary operator $\odot$ that has an identity. For example, floating-point multiplication has identity 1 and is approximately associative.

```
trait Monoid⟦T extends Monoid⟦T, ⊙⟧, opr ⊙⟧
    extends { ApproximateMonoid⟦T, ⊙, =⟧, Associative⟦T, ⊙⟧ }
end
```

A *monoid* is a set of values with an associative binary operator $\odot$ that has an identity. For example, trait String extends Monoid⟦String, ||⟧ where $\|$ is the string concatenation operator. Note that string concatenation is associative but not commutative and that the empty string is the identity for string concatenation, so coercing Identity⟦ ||, ⟧ to type String produces the empty string.

```
trait ApproximateCommutativeMonoid⟦ T extends ApproximateCommutativeMonoid⟦T, ⊕, ≈⟧,
                                    opr ⊕, opr ≈⟧
    extends { ApproximateMonoid⟦T, ⊕, ≈⟧, ApproximatelyCommutative⟦T, ⊕, ≈⟧ }
end
```

An *approximate commutative monoid* is an approximate monoid whose binary operator is also approximately commutative. For example, floating-point multiplication has identity 1 and is approximately associative and also approximately (indeed, exactly) commutative.

```
trait CommutativeMonoid⟦T extends CommutativeMonoid⟦T, ⊕⟧, opr ⊕⟧
    extends { ApproximateCommutativeMonoid⟦T, ⊕, =⟧, Monoid⟦T, ⊕⟧, Commutative⟦T, ⊕⟧ }
end
```

A *commutative monoid* is a monoid whose binary operator is also commutative. For example, the operator $\wedge$ on boolean values is associative and commutative and has identity $true$; likewise, the operator $\vee$ on boolean values is associative and commutative and has identity $false$.

```
trait ApproximatelyHasInverses⟦T extends ApproximatelyHasInverses⟦T, ⊙, ⊘, ≈⟧,
                                        opr ⊙, opr ⊘, opr ≈⟧
    extends { HasIdentity⟦T, ⊙⟧, UnaryOperator⟦T, ⊘⟧, BinaryOperator⟦T, ⊘⟧ }
  property ∀(a: T) (a ⊙ (⊘ a)) :≈: Identity⟦⊙⟧
  property ∀(a: T) ((⊘ a) ⊙ a) :≈: Identity⟦⊙⟧
  property ∀(a: T, b: T) (a ⊘ b) :≈: (a ⊙ (⊘ b))
end
```

A set of values with a binary operator $\odot$ has *approximate inverses* if and only if the operator has an identity and for every value a there is another value a' such that the result of applying $\odot$ to a and a' (in either order) is "close enough" to the identity. The unary operator $\oslash$ returns the approximate inverse of its argument; as a notational convenience, it may also be used as a binary operator.

```
trait HasInverses⟦T extends HasInverses⟦T, ⊙, ⊘⟧, opr ⊙, opr ⊘⟧
    extends { ApproximatelyHasInverses⟦T, ⊙, ⊘, =⟧ }
end
```

A set of values with a binary operator $\odot$ has *inverses* if and only if the operator has an identity and for every value a there is another value a' such that the result of applying $\odot$ to a and a' (in either order) equals the identity. The unary operator $\oslash$ returns the inverse of its argument; as a notational convenience, it may also be used as a binary operator. A standard example is the operator $+$ on integers; the identity is 0, and the unary operator $-$ returns the additive inverse of its argument, such that $a + (-a) = 0$ and $(-a) + a = 0$. Moreover, $-$ may be used as a binary operator: $a - b$ means $a + (-b)$.

```
trait ApproximateGroup⟦T extends ApproximateGroup⟦T, ⊙, ⊘, ≈⟧, opr ⊙, opr ⊘, opr ≈⟧
    extends { ApproximateMonoid⟦T, ⊙, ≈⟧, ApproximatelyHasInverses⟦T, ⊙, ⊘, ≈⟧ }
end
```

An *approximate group* is an approximate monoid that has approximate inverses. For example, a floating-point representation of quaternions with multiplication as the binary operator would form an approximate group.

```
trait Group⟦T extends Group⟦T, ⊙, ⊘⟧, opr ⊙, opr ⊘⟧
    extends { ApproximateGroup⟦T, ⊙, ⊘, =⟧, Monoid⟦T, ⊙⟧, HasInverses⟦T, ⊙, ⊘⟧ }
end
```

A *group* is monoid that has inverses.

```
trait ApproximateAbelianGroup⟦T extends ApproximateAbelianGroup⟦T, ⊕, ⊖, ≈⟧,
                                        opr ⊕, opr ⊖, opr ≈⟧
    extends { ApproximateGroup⟦T, ⊕, ⊖, ≈⟧,
              ApproximateCommutativeMonoid⟦T, ⊕, ≈⟧ }
end
```

An *approximate Abelian group* is an approximate group whose binary operator is also approximately commutative.

```
trait AbelianGroup⟦T extends AbelianGroup⟦T, ⊕, ⊖⟧, opr ⊕, opr ⊖⟧
    extends { ApproximateAbelianGroup⟦T, ⊕, ⊖, =⟧,
              Group⟦T, ⊕, ⊖⟧, CommutativeMonoid⟦T, ⊕⟧ }
end
```

An *Abelian group* is group whose binary operator is also commutative. (The term "Abelian" is traditionally used instead of "commutative" when discussing groups, in tribute to mathematician Niels Henrik Abel.) For example, the integers with the binary addition operator $+$, unary negation operator $-$, and identity 0 form an Abelian group. As another example, the boolean values with the binary exclusive OR operator $\oplus$ and unary negation operator `IDENTITY` form an Abelian group; the value *false* is the identity for $\oplus$.

```
trait ApproximateSemiRing⟦T extends ApproximateSemiRing⟦T, ⊕, ⊗, ≈⟧,
                                   opr ⊕, opr ⊗, opr ≈⟧
    extends { ApproximateCommutativeMonoid⟦T, ⊕, ≈⟧,
              ApproximateMonoid⟦T, ⊗, ≈⟧,
              ApproximatelyDistributive⟦T, ⊗, ⊕, ≈⟧,
              ApproximateZeroAnnihilation⟦T, ⊗, ≈⟧ }
  property cast⟦T⟧(Zero⟦⊗⟧) ≈ cast⟦T⟧(Identity⟦⊕⟧)
end
```

An *approximate semiring* is a set of values that has two binary operators $\oplus$ and $\otimes$, such that (a) the values form an approximate commutative monoid with $\oplus$; (b) the values form an approximate monoid with $\otimes$; (c) $\otimes$ is approximately distributive over $\oplus$; and (d) $\otimes$ obeys approximate zero annihilation, where the zero for $\otimes$ is the same as the identity for $\oplus$.

```
trait SemiRing⟦T extends SemiRing⟦T, ⊕, ⊗⟧, opr ⊕, opr ⊗⟧
    extends { ApproximateSemiRing⟦T, ⊕, ⊗, =⟧,
              CommutativeMonoid⟦T, ⊕⟧,
              Monoid⟦T, ⊗⟧,
              Distributive⟦T, ⊗, ⊕⟧,
              ZeroAnnihilation⟦T, ⊗⟧ }
end
```

A *semiring* is a set of values that has two binary operators $\oplus$ and $\otimes$, such that (a) the values form a commutative monoid with $\oplus$; (b) the values form a monoid with $\otimes$; (c) $\otimes$ is distributive over $\oplus$; and (d) $\otimes$ obeys zero annihilation, where the zero for $\otimes$ is the same as the identity for $\oplus$.

```
trait ApproximateRing⟦T extends ApproximateRing⟦T, ⊕, ⊖, ⊗, ≈⟧,
                               opr ⊕, opr ⊖, opr ⊗, opr ≈⟧
    extends { ApproximateAbelianGroup⟦T, ⊕, ⊖, ≈⟧,
              ApproximateSemiRing⟦T, ⊕, ⊗, ≈⟧ }
end
```

An *approximate ring* is an approximate semiring that also has a unary operator $\ominus$ that returns inverses for the $\oplus$ operator so that the values form an approximate group with $\oplus$ and $\ominus$.

```
trait Ring⟦T extends Ring⟦T, ⊕, ⊖, ⊗⟧, opr ⊕, opr ⊖, opr ⊗⟧
    extends { ApproximateRing⟦T, ⊕, ⊖, ⊗, =⟧,
              AbelianGroup⟦T, ⊕, ⊖⟧,
              SemiRing⟦T, ⊕, ⊗⟧ }
end
```

A *ring* is a semiring that also has a unary operator $\ominus$ that returns inverses for the $\oplus$ operator so that the values form a group with $\oplus$ and $\ominus$.

```
trait ApproximateCommutativeRing⟦T extends ApproximateCommutativeRing⟦T, ⊕, ⊖, ⊗, ≈⟧,
                                          opr ⊕, opr ⊖, opr ⊗, opr ≈⟧
    extends { ApproximateRing⟦T, ⊕, ⊖, ⊗, ≈⟧,
              ApproximatelyCommutative⟦T, ⊗, ≈⟧ }
end
```

An *approximate commutative ring* is an approximate ring for which the binary operator $\otimes$ is also approximately commutative.

```
trait CommutativeRing⟦T extends CommutativeRing⟦T, ⊕, ⊖, ⊗⟧,
                               opr ⊕, opr ⊖, opr ⊗⟧
```

```
    extends { ApproximateCommutativeRing⟦T, ⊕, ⊖, ⊗, =⟧,
              Ring⟦T, ⊕, ⊖, ⊗⟧,
              Commutative⟦T, ⊗⟧ }
end
```

A *commutative ring* is a ring for which the binary operator $\otimes$ is also commutative.

```
trait ApproximateDivisionRing⟦T extends ApproximateDivisionRing⟦T, U, ⊕, ⊖, ⊗, ⊘, ≈⟧,
                              U extends T,
                              opr ⊕, opr ⊖, opr ⊗, opr ⊘, opr ≈⟧
    extends { ApproximateRing⟦T, ⊕, ⊖, ⊗, ≈⟧,
              ApproximateGroup⟦U, ⊗, ⊘, ≈⟧ }
  property ¬ instanceOf⟦U⟧(cast⟦T⟧(Zero⟦⊕⟧))
  property ∀(a: T) a ≠ Zero⟦⊕⟧ → instanceOf⟦U⟧(a)
```

An *approximate division ring* is an approximate ring for which the binary operator $\otimes$ also has approximate inverses, so that the values other than the zero of $\otimes$ form an approximate group with $\otimes$ and $\oslash$.

```
trait DivisionRing⟦T extends DivisionRing⟦T, U, ⊕, ⊖, ⊗, ⊘⟧,
                   U extends T,
                   opr ⊕, opr ⊖, opr ⊗, opr ⊘⟧
    extends { ApproximateDivisionRing⟦T, U, ⊕, ⊖, ⊗, ⊘, =⟧,
              Ring⟦T, ⊕, ⊖, ⊗⟧,
              Group⟦U, ⊗, ⊘⟧ }
end
```

A *division ring* is a ring for which the binary operator $\otimes$ also has inverses, so that the values other than the zero of $\otimes$ form a group with $\otimes$ and $\oslash$.

```
trait ApproximateField⟦T extends ApproximateField⟦T, U, ⊕, ⊖, ⊗, ⊘, ≈⟧,
                       U extends T,
                       opr ⊕, opr ⊖, opr ⊗, opr ⊘, opr ≈⟧
    extends { ApproximateCommutativeRing⟦T, ⊕, ⊖, ⊗, ≈⟧,
              ApproximateDivisionRing⟦T, U, ⊕, ⊖, ⊗, ⊘, ≈⟧ }
end
```

An *approximate field* is an approximate commutative ring that is also an approximate division ring: the binary operator $\otimes$ is approximately commutative and also has approximate inverses, so that the values other than the zero of $\otimes$ form an approximate Abelian group with $\otimes$ and $\oslash$.

```
trait Field⟦T extends Field⟦T, U, ⊕, ⊖, ⊗, ⊘⟧, U extends T, opr ⊕, opr ⊖, opr ⊗, opr ⊘⟧
    extends { ApproximateField⟦T, U, ⊕, ⊖, ⊗, ⊘, =⟧,
              CommutativeRing⟦T, ⊕, ⊖, ⊗⟧,
              DivisionRing⟦T, U, ⊕, ⊖, ⊗, ⊘⟧ }
end
```

A *field* is a commutative ring that is also a division ring: the binary operator $\otimes$ is commutative and also has inverses, so that the values other than the zero of $\otimes$ form an Abelian group with $\otimes$ and $\oslash$.

37.5 Boolean Algebras

```
value object ComplementBound⟦opr ⊙⟧ end
trait HasComplements⟦T extends HasComplements⟦T, ⊙, ∼⟧, opr ⊙, opr ∼⟧
```

```
      extends { BinaryOperator⟦T, ⊙⟧, UnaryOperator⟦T, ∼⟧, EquivalenceRelation⟦T, =⟧ }
      where { T coerces ComplementBound⟦⊙⟧ }
  property ∀(a: T) (a ⊙ (∼ a)) :=: ComplementBound⟦⊙⟧
end
```

A set of values with a binary operator $\odot$ has *complements* if and only if there is a specific value e such that for every value a there is another value a' such that the result of applying $\odot$ to a and a' (in either order) equals the specified value e. This value may be obtained by coercing the object named $\mathrm{ComplementBound}[\![\odot]\!]$ to type T. The unary operator $\sim$ returns the complement of its argument with respect to the operator $\odot$.

Note that the trait HasComplements is similar to the trait HasInverses, but HasComplements does not require that that specified element be an identity of the binary operator.

```
trait DeMorgan⟦T extends DeMorgan⟦T, ⅄, ⅂, ∼⟧, opr ⅄, opr ⅂, opr ∼⟧
      extends { BinaryOperator⟦T, ⅄⟧, BinaryOperator⟦T, ⅂⟧,
                UnaryOperator⟦T, ∼⟧, EquivalenceRelation⟦T, =⟧ }
  property ∀(a: T, b: T) (∼ (a ⅂ b)) :=: ((∼ a) ⅄ (∼ b))
end
```

This trait expresses De Morgan's law for two binary operators $⅄$ and $⅂$ and a unary operator $\sim$: the expressions $\sim (a \curlyvee b)$ and $(\sim a) \curlywedge (\sim b)$ produce equal results.

```
trait BooleanAlgebra⟦T extends BooleanAlgebra⟦T, ⅄, ⅂, ∼, ⩒⟧, opr ⅄, opr ⅂, opr ∼, opr ⩒⟧
      extends { Commutative⟦T, ⅄⟧, Associative⟦T, ⅄⟧,
                Commutative⟦T, ⅂⟧, Associative⟦T, ⅂⟧,
                IdempotentBinaryOperator⟦T, ⅄⟧,
                IdempotentBinaryOperator⟦T, ⅂⟧,
                HasIdentity⟦T, ⅄⟧, HasIdentity⟦T, ⅂⟧,
                HasComplements⟦T, ⅂, ∼⟧, HasComplements⟦T, ⅄, ∼⟧,
                Distributive⟦T, ⅄, ⅂⟧, Distributive⟦T, ⅂, ⅄⟧,
                DeMorgan⟦T, ⅄, ⅂, ∼⟧, DeMorgan⟦T, ⅂, ⅄, ∼⟧,
                Ring⟦T, ⩒, IDENTITY, ⅄⟧ }
  property ∀(a: T) (∼ (∼ a)) :=: a
  opr ⩒(self, other: T): T = (self ⅄ (∼ other)) ⅂ ((∼ self) ⅄ other)
  property cast⟦T⟧(Identity⟦⩒⟧) = cast⟦T⟧(Identity⟦⅂⟧)
  property cast⟦T⟧(ComplementBound⟦⅄⟧) = cast⟦T⟧(Identity⟦⅂⟧)
  property cast⟦T⟧(ComplementBound⟦⅂⟧) = cast⟦T⟧(Identity⟦⅄⟧)
end
```

A *boolean algebra* is an algebraic structure consisting of a set of values, three binary operators $⅄$, $⅂$, and $⩒$, and a unary operator $\sim$, such that the operators obey a number of specific properties:

- $⅄$ is commutative, associative, and idempotent, and has a unique identity
- $⅂$ is commutative, associative, and idempotent, and has a unique identity
- $⅄$ has complements with respect to $\sim$
- $⅂$ has complements with respect to $\sim$
- $⅄$ and $⅂$ distribute over each other
- De Morgan's law applies to $⅄$, $⅂$, and $\sim$, and also to $⅂$, $⅄$, and $\sim$
- The values form a ring with $⩒$ as the addition operator, IDENTITY as the additive inverse operator, and $⅄$ as the multiplication operator

A default definition is provided for the $\underline{\vee}$ operator in terms of $\curlywedge$, $\curlyvee$, and $\sim$.

The type $\mathrm{Boolean}$ is the most familiar example of a boolean algebra. The power set of a set (that is, the set of all subsets of the set) also forms a boolean algebra with operators $\cap$, $\cup$, set complement, and symmetric set difference; the empty set is the identity for $\cup$, and the original set is the identity for $\cap$.

Chapter 38

Numbers

38.1 The Trait Fortress.Standard.RationalQuantity

The standard types for rational numbers such as $\mathbb{Q}$ and $\mathbb{Q}^*$ and $\mathbb{Q}^\#_\leq$ are defined in terms of a single trait $\mathrm{RationalQuantity}$ that handles dimensions and units as well as performing a case analysis to distinguish rather a particular expression is guaranteed not to produce infinities, $0/0$, or numbers of a particular sign.

The trait $\mathrm{RationalQuantity}$ takes seven static parameters; the first is a dimensional unit, and the others are booleans specifying whether an instance of the trait can possibly be $-\infty$, a finite rational less than zero, zero, a finite rational greater than zero, $+\infty$, or $0/0$. This allows the standard rational types to be represented as follows:

`type` $\mathbb{Q} = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{false}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}]\!]$
`type` $\mathbb{Q}_< = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{false}, \mathit{true}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}]\!]$
`type` $\mathbb{Q}_\leq = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{false}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}, \mathit{false}]\!]$
`type` $\mathbb{Q}_\geq = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{false}, \mathit{false}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}]\!]$
`type` $\mathbb{Q}_> = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true}, \mathit{false}, \mathit{false}]\!]$
`type` $\mathbb{Q}_\neq = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{false}, \mathit{true}, \mathit{false}, \mathit{true}, \mathit{false}, \mathit{false}]\!]$
`type` $\mathbb{Q}^* = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{false}]\!]$
`type` $\mathbb{Q}^*_< = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}]\!]$
`type` $\mathbb{Q}^*_\leq = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}, \mathit{false}]\!]$
`type` $\mathbb{Q}^*_\geq = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{false}, \mathit{false}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{false}]\!]$
`type` $\mathbb{Q}^*_> = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true}, \mathit{true}, \mathit{false}]\!]$
`type` $\mathbb{Q}^*_\neq = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{true}, \mathit{true}, \mathit{false}]\!]$
`type` $\mathbb{Q}^\# = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}]\!]$
`type` $\mathbb{Q}^\#_< = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true}]\!]$
`type` $\mathbb{Q}^\#_\leq = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{false}, \mathit{true}]\!]$
`type` $\mathbb{Q}^\#_\geq = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{false}, \mathit{false}, \mathit{true}, \mathit{true}, \mathit{true}, \mathit{true}]\!]$
`type` $\mathbb{Q}^\#_> = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true}, \mathit{true}, \mathit{true}]\!]$
`type` $\mathbb{Q}^\#_\neq = \mathrm{RationalQuantity}[\![\mathrm{dimensionless}, \mathit{true}, \mathit{true}, \mathit{false}, \mathit{true}, \mathit{true}, \mathit{true}]\!]$

Here is the detailed description of $\mathrm{RationalQuantity}$, showing the details of the type calculations:

```
trait RationalQuantity⟦unit U absorbs unit, bool ninf, bool lt, bool eq, bool gt, bool pinf, bool nan⟧
    extends { RationalQuantity⟦U, ninf′, lt′, eq′, gt′, pinf′, nan′⟧
              where { bool ninf′, bool lt′, bool eq′, bool gt′, bool pinf′, bool nan′,
                      ninf → ninf′, lt → lt′, eq → eq′, gt → gt′, pinf → pinf′, nan → nan′ },
              Field⟦ RationalQuantity⟦U, ninf, lt, eq, gt, pinf, nan⟧,
                     RationalQuantity⟦U, ninf, lt, false, gt, pinf, nan⟧, +, −, ·, /⟧
              where { lt ∧ eq ∧ gt ∧ ¬ninf ∧ ¬pinf ∧ ¬nan, U = dimensionless },
              Field⟦ RationalQuantity⟦U, ninf, lt, eq, gt, pinf, nan⟧,
                     RationalQuantity⟦U, ninf, lt, false, gt, pinf, nan⟧, +, −, ×, /⟧
              where { lt ∧ eq ∧ gt ∧ ¬ninf ∧ ¬pinf ∧ ¬nan, U = dimensionless },
              Field⟦ RationalQuantity⟦U, ninf, lt, eq, gt, pinf, nan⟧,
                     RationalQuantity⟦U, ninf, lt, false, gt, pinf, nan⟧, +, −, juxtaposition, /⟧
              where { lt ∧ eq ∧ gt ∧ ¬ninf ∧ ¬pinf ∧ ¬nan, U = dimensionless },
              AbelianGroup⟦RationalQuantity⟦U, ninf, lt, eq, gt, pinf, nan⟧, +, −⟧,
              TotalOrderOperators⟦RationalQuantity⟦U, ninf, lt, eq, gt, pinf, nan⟧, <, ≤, ≥, >, CMP⟧
              where { ¬nan } }
      where { ninf ∨ lt ∨ eq ∨ gt ∨ pinf ∨ nan }
  coercion (_: Identity⟦+⟧) = 0
  coercion (_: Identity⟦·⟧) = 1
  coercion (_: Identity⟦×⟧) = 1
  coercion (_: Identity⟦juxtaposition⟧) = 1
  coercion (_: Zero⟦×⟧) = 0
  coercion (x: IntegerQuantity⟦U, ninf, lt, eq, gt, pinf, nan⟧)
  opr juxtaposition ⟦unit U′, bool ninf′, bool lt′, bool eq′, bool gt′, bool pinf′, bool nan′⟧
        (self, other: RationalQuantity⟦U′, ninf′, lt′, eq′, gt′, pinf′, nan′⟧):
        RationalQuantity⟦ U U′,
                          ninf ∧ pinf′ ∨ ninf ∧ gt′ ∨ lt ∧ pinf′ ∨ pinf ∧ ninf′ ∨ pinf ∧ lt′ ∨ gt ∧ ninf′,
                          lt ∧ gt′ ∨ gt ∧ lt′,
                          eq ∧ (lt′ ∨ eq′ ∨ gt′) ∨ (lt ∨ eq ∨ gt) ∧ eq′,
                          lt ∧ lt′ ∨ gt ∧ gt′,
                          ninf ∧ ninf′ ∨ ninf ∧ lt′ ∨ lt ∧ ninf′ ∨ pinf ∧ pinf′ ∨ pinf ∧ gt′ ∨ gt ∧ pinf′,
                          nan ∨ nan′ ∨ ninf ∧ eq′ ∨ pinf ∧ eq′ ∨ eq ∧ ninf′ ∨ eq ∧ pinf′⟧
  opr +(self): RationalQuantity⟦U, ninf, lt, eq, gt, pinf, nan⟧
  opr +⟦bool ninf′, bool lt′, bool eq′, bool gt′, bool inf′, bool nan′⟧
        (self, other: RationalQuantity⟦U, ninf′, lt′, eq′, gt′, pinf′, nan′⟧):
        RationalQuantity⟦ U,
                          ninf ∧ (ninf′ ∨ lt′ ∨ eq′ ∨ gt′) ∨ (ninf ∨ lt ∨ eq ∨ gt) ∧ ninf′,
                          lt ∧ (lt′ ∨ eq′ ∨ gt′) ∨ (lt ∨ eq ∨ gt) ∧ lt′,
                          lt ∧ gt′ ∨ eq ∧ eq′ ∨ gt ∧ lt′,
                          gt ∧ (lt′ ∨ eq′ ∨ gt′) ∨ (lt ∨ eq ∨ gt) ∧ gt′,
                          pinf ∧ (lt′ ∨ eq′ ∨ gt′ ∨ pinf′) ∨ (lt ∨ eq ∨ gt ∨ pinf) ∧ pinf′,
                          nan ∨ nan′ ∨ ninf ∧ pinf′ ∨ pinf ∧ ninf′⟧
  opr −(self): RationalQuantity⟦U, pinf, gt, eq, lt, ninf, nan⟧
  opr −⟦bool ninf′, bool lt′, bool eq′, bool gt′, bool pinf′, bool nan′⟧
        (self, other: RationalQuantity⟦U, ninf′, lt′, eq′, gt′, pinf′, nan′⟧):
        RationalQuantity⟦ U,
                          ninf ∧ (lt′ ∨ eq′ ∨ gt′ ∨ pinf′) ∨ (ninf ∨ lt ∨ eq ∨ gt) ∧ pinf′,
                          lt ∧ (lt′ ∨ eq′ ∨ gt′) ∨ (lt ∨ eq ∨ gt) ∧ gt′,
                          lt ∧ lt′ ∨ eq ∧ eq′ ∨ gt ∧ gt′,
                          gt ∧ (lt′ ∨ eq′ ∨ gt′) ∨ (lt ∨ eq ∨ gt) ∧ lt′,
                          pinf ∧ (ninf′ ∨ lt′ ∨ eq′ ∨ gt′) ∨ (lt ∨ eq ∨ gt ∨ pinf) ∧ ninf′,
                          nan ∨ nan′ ∨ ninf ∧ ninf′ ∨ pinf ∧ pinf′⟧
```

opr $\cdot[\![\texttt{unit}\ U', \texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf', \texttt{bool}\ nan']\!]$
$(\texttt{self}, other\colon \text{RationalQuantity}[\![U', ninf', lt', eq', gt', pinf', nan']\!])\colon$
$\text{RationalQuantity}[\![\, U\ U',$
$ninf \wedge pinf' \vee ninf \wedge gt' \vee lt \wedge pinf' \vee pinf \wedge ninf' \vee pinf \wedge lt' \vee gt \wedge ninf',$
$lt \wedge gt' \vee gt \wedge lt',$
$eq \wedge (lt' \vee eq' \vee gt') \vee (lt \vee eq \vee gt) \wedge eq',$
$lt \wedge lt' \vee gt \wedge gt',$
$ninf \wedge ninf' \vee ninf \wedge lt' \vee lt \wedge ninf' \vee pinf \wedge pinf' \vee pinf \wedge gt' \vee gt \wedge pinf',$
$nan \vee nan' \vee ninf \wedge eq' \vee pinf \wedge eq' \vee eq \wedge ninf' \vee eq \wedge pinf']\!]$
opr $/(\texttt{self})\colon \text{RationalQuantity}[\![1/U, eq, lt, ninf \vee pinf, gt, eq, nan]\!]$
opr $/[\![\texttt{unit}\ U', \texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf', \texttt{bool}\ nan']\!]$
$(\texttt{self}, other\colon \text{RationalQuantity}[\![U', ninf', lt', eq', gt', pinf', nan']\!])\colon$
$\text{RationalQuantity}[\![\, U/U',$
$ninf \wedge eq' \vee ninf \wedge gt' \vee lt \wedge eq' \vee pinf \wedge lt',$
$lt \wedge gt' \vee gt \wedge lt',$
$eq \wedge (lt' \vee gt') \vee (lt \vee eq \vee gt) \wedge (ninf' \vee pinf')',$
$lt \wedge lt' \vee gt \wedge gt',$
$ninf \wedge lt' \vee pinf \wedge eq' \vee pinf \wedge gt' \vee gt \wedge eq',$
$nan \vee nan' \vee (ninf \vee pinf) \wedge (ninf' \vee pinf') \vee eq \wedge eq']\!]$
opr $<[\![\texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf', \texttt{bool}\ nan']\!]$
$(\texttt{self}, other\colon \text{RationalQuantity}[\![U, ninf', lt', eq', gt', pinf', nan']\!])\colon \text{Boolean}$
opr $\leq[\![\texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf', \texttt{bool}\ nan']\!]$
$(\texttt{self}, other\colon \text{RationalQuantity}[\![U, ninf', lt', eq', gt', pinf', nan']\!])\colon \text{Boolean}$
opr $=[\![\texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf', \texttt{bool}\ nan']\!]$
$(\texttt{self}, other\colon \text{RationalQuantity}[\![U, ninf', lt', eq', gt', pinf', nan']\!])\colon \text{Boolean}$
opr $\geq[\![\texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf', \texttt{bool}\ nan']\!]$
$(\texttt{self}, other\colon \text{RationalQuantity}[\![U, ninf', lt', eq', gt', pinf', nan']\!])\colon \text{Boolean}$
opr $>[\![\texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf', \texttt{bool}\ nan']\!]$
$(\texttt{self}, other\colon \text{RationalQuantity}[\![U, ninf', lt', eq', gt', pinf', nan']\!])\colon \text{Boolean}$
opr $\texttt{CMP}[\![\texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf']\!]$
$(\texttt{self}, other\colon \text{RationalQuantity}[\![U, ninf', lt', eq', gt', pinf', false]\!])\colon \text{TotalComparison}$
opr $\texttt{CMP}[\![\texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf']\!]$
$(\texttt{self}, other\colon \text{RationalQuantity}[\![U, ninf', lt', eq', gt', pinf', true]\!])\colon \text{Comparison}$
opr $\texttt{MAX}[\![\texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf', \texttt{bool}\ nan']\!]$
$(\texttt{self}, other\colon \text{RationalQuantity}[\![U, ninf', lt', eq', gt', pinf', nan']\!])\colon$
$\text{RationalQuantity}[\![\, U,$
$ninf \wedge ninf',$
$lt \wedge (ninf' \vee lt') \vee (ninf \vee lt) \wedge lt',$
$eq \wedge (ninf' \vee lt' \vee eq') \vee (ninf \vee lt \vee eq) \wedge eq',$
$gt \wedge (ninf' \vee lt' \vee eq' \vee gt') \vee (ninf \vee lt \vee eq \vee gt) \wedge gt',$
$pinf \wedge (ninf' \vee lt' \vee eq' \vee gt' \vee pinf') \vee (ninf \vee lt \vee eq \vee gt \vee pinf) \wedge pinf',$
$nan \vee nan']\!]$
opr $\texttt{MIN}[\![\texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf', \texttt{bool}\ nan']\!]$
$(\texttt{self}, other\colon \text{RationalQuantity}[\![U, ninf', lt', eq', gt', pinf', nan']\!])\colon$
$\text{RationalQuantity}[\![\, U,$
$ninf \wedge (ninf' \vee lt' \vee eq' \vee gt' \vee pinf') \vee (ninf \vee lt \vee eq \vee gt \vee pinf) \wedge ninf',$
$lt \wedge (lt' \vee eq' \vee gt' \vee pinf') \vee (lt \vee eq \vee gt \vee pinf) \wedge lt',$
$eq \wedge (eq' \vee gt' \vee pinf') \vee (eq \vee gt \vee pinf) \wedge eq',$
$gt \wedge (gt' \vee pinf') \vee (gt \vee pinf) \wedge gt',$
$pinf \wedge pinf',$
$nan \vee nan']\!]$
opr $\texttt{MAXNUM}[\![\texttt{bool}\ ninf', \texttt{bool}\ lt', \texttt{bool}\ eq', \texttt{bool}\ gt', \texttt{bool}\ pinf', \texttt{bool}\ nan']\!]$

$(\texttt{self}, \mathit{other}\colon \mathrm{RationalQuantity}[\![U, \mathit{ninf}', \mathit{lt}', \mathit{eq}', \mathit{gt}', \mathit{pinf}', \mathit{nan}']\!])\colon$
$\mathrm{RationalQuantity}[\![\, U,$
$\mathit{ninf} \wedge (\mathit{nan}' \vee \mathit{ninf}') \vee (\mathit{nan} \vee \mathit{ninf}) \wedge \mathit{ninf}',$
$\mathit{lt} \wedge (\mathit{nan}' \vee \mathit{ninf}' \vee \mathit{lt}') \vee (\mathit{nan} \vee \mathit{ninf} \vee \mathit{lt}) \wedge \mathit{lt}',$
$\mathit{eq} \wedge (\mathit{nan}' \vee \mathit{ninf}' \vee \mathit{lt}' \vee \mathit{eq}') \vee (\mathit{nan} \vee \mathit{ninf} \vee \mathit{lt} \vee \mathit{eq}) \wedge \mathit{eq}',$
$\mathit{gt} \wedge (\mathit{nan}' \vee \mathit{ninf}' \vee \mathit{lt}' \vee \mathit{eq}' \vee \mathit{gt}') \vee (\mathit{nan} \vee \mathit{ninf} \vee \mathit{lt} \vee \mathit{eq} \vee \mathit{gt}) \wedge \mathit{gt}',$
$\mathit{pinf} \wedge (\mathit{nan}' \vee \mathit{ninf}' \vee \mathit{lt}' \vee \mathit{eq}' \vee \mathit{gt}' \vee \mathit{pinf}') \vee$
$(\mathit{nan} \vee \mathit{ninf} \vee \mathit{lt} \vee \mathit{eq} \vee \mathit{gt} \vee \mathit{pinf}) \wedge \mathit{pinf}',$
$\mathit{nan} \wedge \mathit{nan}'\,]\!]$

$\texttt{opr MINNUM}[\![\texttt{bool } \mathit{ninf}', \texttt{bool } \mathit{lt}', \texttt{bool } \mathit{eq}', \texttt{bool } \mathit{gt}', \texttt{bool } \mathit{pinf}', \texttt{bool } \mathit{nan}']\!]$
$(\texttt{self}, \mathit{other}\colon \mathrm{RationalQuantity}[\![U, \mathit{ninf}', \mathit{lt}', \mathit{eq}', \mathit{gt}', \mathit{pinf}', \mathit{nan}']\!])\colon$
$\mathrm{RationalQuantity}[\![\, U,$
$\mathit{ninf} \wedge (\mathit{ninf}' \vee \mathit{lt}' \vee \mathit{eq}' \vee \mathit{gt}' \vee \mathit{pinf}' \vee \mathit{nan}') \vee$
$(\mathit{ninf} \vee \mathit{lt} \vee \mathit{eq} \vee \mathit{gt} \vee \mathit{pinf} \vee \mathit{nan}) \wedge \mathit{ninf}',$
$\mathit{lt} \wedge (\mathit{lt}' \vee \mathit{eq}' \vee \mathit{gt}' \vee \mathit{pinf}' \vee \mathit{nan}') \vee (\mathit{lt} \vee \mathit{eq} \vee \mathit{gt} \vee \mathit{pinf} \vee \mathit{nan}) \wedge \mathit{lt}',$
$\mathit{eq} \wedge (\mathit{eq}' \vee \mathit{gt}' \vee \mathit{pinf}' \vee \mathit{nan}') \vee (\mathit{eq} \vee \mathit{gt} \vee \mathit{pinf} \vee \mathit{nan}) \wedge \mathit{eq}',$
$\mathit{gt} \wedge (\mathit{gt}' \vee \mathit{pinf}' \vee \mathit{nan}') \vee (\mathit{gt} \vee \mathit{pinf} \vee \mathit{nan}) \wedge \mathit{gt}',$
$\mathit{pinf} \wedge (\mathit{pinf}' \vee \mathit{nan}') \vee (\mathit{pinf} \vee \mathit{nan}) \wedge \mathit{pinf}',$
$\mathit{nan} \wedge \mathit{nan}'\,]\!]$

$\texttt{opr } |\texttt{self}| : \mathrm{RationalQuantity}[\![U, \mathit{false}, \mathit{false}, \mathit{eq}, \mathit{lt} \vee \mathit{gt}, \mathit{ninf} \vee \mathit{pinf}, \mathit{nan}]\!]$
$\mathit{signum}(\texttt{self})\colon \mathrm{RationalQuantity}[\![U, \mathit{false}, \mathit{lt}, \mathit{eq}, \mathit{gt}, \mathit{false}, \mathit{nan}]\!]$
$\mathit{numerator}(\texttt{self})\colon \mathrm{IntegerQuantity}[\![U, \mathit{false}, \mathit{ninf} \vee \mathit{lt}, \mathit{eq}, \mathit{gt} \vee \mathit{pinf}, \mathit{false}, \mathit{nan}]\!]$
$\mathit{denominator}(\texttt{self})\colon \mathrm{IntegerQuantity}[\![\mathrm{dimensionless}, \mathit{false}, \mathit{false}, \mathit{ninf} \vee \mathit{pinf}, \mathit{lt} \vee \mathit{eq} \vee \mathit{gt}, \mathit{false}, \mathit{nan}]\!]$
$\mathit{floor}(\texttt{self})\colon \mathrm{IntegerQuantity}[\![U, \mathit{ninf}, \mathit{lt}, \mathit{eq} \vee \mathit{gt}, \mathit{gt}, \mathit{pinf}, \mathit{nan}]\!]$
$\texttt{opr } \lfloor \texttt{self} \rfloor\colon \mathrm{IntegerQuantity}[\![U, \mathit{ninf}, \mathit{lt}, \mathit{eq} \vee \mathit{gt}, \mathit{gt}, \mathit{pinf}, \mathit{nan}]\!]$
$\mathit{ceiling}(\texttt{self})\colon \mathrm{IntegerQuantity}[\![U, \mathit{ninf}, \mathit{lt}, \mathit{lt} \vee \mathit{eq}, \mathit{gt}, \mathit{pinf}, \mathit{nan}]\!]$
$\texttt{opr } \lceil \texttt{self} \rceil\colon \mathrm{IntegerQuantity}[\![U, \mathit{ninf}, \mathit{lt}, \mathit{lt} \vee \mathit{eq}, \mathit{gt}, \mathit{pinf}, \mathit{nan}]\!]$
$\mathit{round}(\texttt{self})\colon \mathrm{IntegerQuantity}[\![U, \mathit{ninf}, \mathit{lt}, \mathit{lt} \vee \mathit{eq} \vee \mathit{gt}, \mathit{gt}, \mathit{pinf}, \mathit{nan}]\!]$
$\mathit{truncate}(\texttt{self})\colon \mathrm{IntegerQuantity}[\![U, \mathit{ninf}, \mathit{lt}, \mathit{lt} \vee \mathit{eq} \vee \mathit{gt}, \mathit{gt}, \mathit{pinf}, \mathit{nan}]\!]$
$\mathit{realpart}(\texttt{self})\colon \mathrm{RationalQuantity}[\![U, \mathit{ninf}, \mathit{lt}, \mathit{eq}, \mathit{gt}, \mathit{pinf}, \mathit{nan}]\!]$
$\mathit{imagpart}(\texttt{self})\colon \mathrm{RationalQuantity}[\![U, \mathit{false}, \mathit{false}, \mathit{true}, \mathit{false}, \mathit{false}, \mathit{nan}]\!]$
$\mathit{check}(\texttt{self})\colon \mathbb{Q}\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}^*(\texttt{self})\colon \mathbb{Q}^*\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}_<(\texttt{self})\colon \mathbb{Q}_<\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}_\le(\texttt{self})\colon \mathbb{Q}_\le\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}_\ge(\texttt{self})\colon \mathbb{Q}_\ge\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}_>(\texttt{self})\colon \mathbb{Q}_>\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}_\neq(\texttt{self})\colon \mathbb{Q}_\neq\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}^*_<(\texttt{self})\colon \mathbb{Q}^*_<\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}^*_\le(\texttt{self})\colon \mathbb{Q}^*_\le\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}^*_\ge(\texttt{self})\colon \mathbb{Q}^*_\ge\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}^*_>(\texttt{self})\colon \mathbb{Q}^*_>\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}^*_\neq(\texttt{self})\colon \mathbb{Q}^*_\neq\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}^\#_<(\texttt{self})\colon \mathbb{Q}^\#_<\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}^\#_\le(\texttt{self})\colon \mathbb{Q}^\#_\le\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}^\#_\ge(\texttt{self})\colon \mathbb{Q}^\#_\ge\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}^\#_>(\texttt{self})\colon \mathbb{Q}^\#_>\ \texttt{throws}\ \mathrm{CastException}$
$\mathit{check}^\#_\neq(\texttt{self})\colon \mathbb{Q}^\#_\neq\ \texttt{throws}\ \mathrm{CastException}$

`end`

For descriptions of the methods, see Section 25.1.

38.2 The Trait Fortress.Standard.TotalComparison

The comparison operator `CMP`, when applied to values belonging to a total order, typically returns a value of type TotalComparison. The three values of type TotalComparison are called LessThan, EqualTo, and GreaterThan.

This trait supports an associative operator `LEXICO` that is useful for supporting lexicographic comparison of ordered sequences; the trick is to compare the sequences elementwise and then to use the `LEXICO` operator to reduce the sequence of comparison results. Note that EqualTo is the identity for `LEXICO`, and all other comparison values are left zeroes for `LEXICO`.

```
value trait TotalComparison
    extends { Comparison,
              Associative⟦TotalComparison, LEXICO⟧,
              HasIdentity⟦TotalComparison, LEXICO⟧,
              HasLeftZeroes⟦TotalComparison, LEXICO, isLeftZeroForLEXICO⟧ }
    comprises { LessThan, EqualTo, GreaterThan }
  opr LEXICO(self, other: TotalComparison): TotalComparison
  isLeftZeroForLEXICO(self): Boolean
  opr ≡(self, other: TotalComparison): Boolean
  getter hashCode(): ℤ64
  toString(): String
end
LessThan: TotalComparison
EqualTo: TotalComparison
GreaterThan: TotalComparison
```

38.2.1 opr LEXICO(self, *other*: TotalComparison): TotalComparison

The operator `LEXICO` returns its right argument if the left argument is EqualTo; otherwise it returns its left argument. The `LEXICO` operator as applied to total comparison values may be described by this table:

LEXICO	LessThan	EqualTo	GreaterThan
LessThan	LessThan	LessThan	LessThan
EqualTo	LessThan	EqualTo	GreaterThan
GreaterThan	GreaterThan	GreaterThan	GreaterThan

38.2.2 *isLeftZeroForLEXICO*(self): Boolean

This method returns *false* for EqualTo and *true* for all other total comparison values.

38.2.3 opr ≡(self, *other*: TotalComparison): Boolean

Two total comparison values are strictly equivalent if and only if they are the same.

38.2.4 `getter` *hashCode*(): ℤ64

38.2.5 *toString*(): String

The *toString* method returns either "`LessThan`" or "`EqualTo`" or "`GreaterThan`" as appropriate.

38.3 Top-level Total Comparison Values

38.3.1 LessThan: TotalComparison
38.3.2 EqualTo: TotalComparison
38.3.3 GreaterThan: TotalComparison

The immutable variables LessThan, EqualTo, and GreaterThan have as their values the three total comparison values that respectively signify whether a left-hand comparand is less than, equal to, or greater than a right-hand comparand. They are top-level variables declared in the Fortress standard libraries.

38.4 The Trait Fortress.Standard.Comparison

When the comparison operator `CMP` is applied to values belonging to a partial order, rather than a total order, it typically returns a value of type Comparison, which includes the three values LessThan, EqualTo, and GreaterThan of type TotalComparison and also a fourth value, Unordered.

This trait, like trait TotalComparison, supports an associative operator `LEXICO` that is useful for supporting lexicographic comparison of ordered sequences; the trick is to compare the sequences elementwise and then to use the `LEXICO` operator to reduce the sequence of comparison results. Note that EqualTo is the identity for `LEXICO`, and all other comparison values are left zeroes for `LEXICO`.

```
value trait Comparison
    extends { IdentityEquality⟦Comparison⟧,
              Associative⟦Comparison, LEXICO⟧,
              HasIdentity⟦Comparison, LEXICO⟧,
              HasLeftZeroes⟦Comparison, LEXICO, isLeftZeroForLEXICO⟧ }
    comprises { TotalComparison, Unordered }
  opr LEXICO(self, other: Comparison): Comparison
  isLeftZeroForLEXICO(self): Boolean
  opr ≡(self, other: Comparison): Boolean
  getter hashCode(): ℤ64
  toString(): String
end
```

Unordered: Comparison

38.4.1 `opr` LEXICO(`self`, *other*: Comparison): Comparison

The operator `LEXICO` returns its right argument if the left argument is EqualTo; otherwise it returns its left argument. The `LEXICO` operator as applied to comparison values may be described by this table:

`LEXICO`	LessThan	EqualTo	GreaterThan	Unordered
LessThan	LessThan	LessThan	LessThan	LessThan
EqualTo	LessThan	EqualTo	GreaterThan	Unordered
GreaterThan	GreaterThan	GreaterThan	GreaterThan	GreaterThan
Unordered	Unordered	Unordered	Unordered	Unordered

38.4.2 *isLeftZeroForLEXICO*(`self`): Boolean

This method returns *false* for EqualTo and *true* for all other comparison values.

38.4.3 `opr` ≡(`self`, *other*: Comparison): Boolean

Two comparison values are strictly equivalent if and only if they are the same.

38.4.4 `getter` *hashCode*(): ℤ64

38.4.5 *toString*(): String

The *toString* method returns either "`LessThan`" or "`EqualTo`" or "`GreaterThan`" or "`Unordered`" as appropriate.

38.5 Top-level Comparison Value

38.5.1 Unordered: Comparison

The immutable variable Unordered has as its value the comparison value that signifies that two comparands are unordered. It is a top-level variable declared in the Fortress standard libraries.

Chapter 39

Components and APIs

We define a special Fortress.Components API that provides handles on components and APIs, and operations on them, for use by components themselves (e.g., development environments), allowing components to build and maintain other components, manipulate projects and components as objects, compile projects into components, link components together, deploy components on specific sites over the internet, etc. This API is also used by the Upgradable and Installable APIs. A component implementing this API is installed along with the Fortress standard libraries on every fortress.

Note that Components and Apis can be constructed only from the factory functions provided in the API. The components and APIs so constructed are also installed and accessible via *getComponent*, *preferences* (which returns a list of components implementing a given API, in order of preference), and *getAPI*. Calling *preferences* on an API in the Fortress standard libraries returns a non-empty list of components. In particular, *preferences*(Fortress.Components) returns a non-empty list whose first element is the very component on which the call to *preferences* was made. Conceptually, this component serves as a handle on the enclosing fortress, which might be necessary for the purposes of certain development tools.

The operations on a fortress provided in this API take components and APIs as arguments directly, rather than names of components and APIs as the corresponding shell operations are described. This decision is done for the sake of convenience. Note, however, that a component name may be rebound on a fortress, or even uninstalled, while some process p keeps a reference to a corresponding Component object. This possibility is not problematic because the component corresponding to this object may be simply kept by the fortress until the object is freed in p. Also, note that `upgrade` operations on a compound component are purely functional: they produce new compound components as a result. Thus, the structure of a component as viewed through a Component object does not became stale in the face of upgrades.

We include a method *getSourceFile* on components that returns the source file the component was compiled from. Source files can be included with simple components during compilation as a compiler option. Doing so allows development tools such as graphical debuggers to easily display the locations of errors without the possibility that source code would not be synchronized with compiled code, as can happen in conventional programming models where compiled code is stored in nonencapsulated object files.

```
api Fortress.Components
import File from Fortress.IO
import { List, Set, Date } from Fortress.Util

trait Fortress
  fortressName: Name
  birthDate: Date
  getComponent(componentName: Name): Component
```

```
    getAPI(apiName: Name): Api
    preferences(ofAPI: Api): ⟨Component⟩
    compile(file: File): SimpleComponent
    install(file: File): Component
    install(file: File, prereqs: Set⟦Api⟧): Component
    upgradeAll(componentName: Name, that: Component): ()
    isValidLink(constituents: ⟨Component⟩, exports = Set⟦Api⟧, hide = Set⟦Api⟧): Boolean
    link(result: Name, constituents: ⟨Component⟩, exports = Set⟦Api⟧, hide = Set⟦Api⟧): Component
      requires isValidLink(constituents, exports, hide)
end

object EnclosingFortress extends { Fortress } end

trait FortressElement
    elementName: Name
    vendor: String
    owner: Fortress
    timeStamp: Date
    version: Version
    uninstall(): ()
end

trait Component extends FortressElement
    imports: Set⟦Api⟧
    exports: Set⟦Api⟧
    provides: Set⟦Api⟧
    visibles: Set⟦Api⟧
    constituents: Set⟦Component⟧
    run(args: String ...): ()
    constrain(destination: Name, apis: Set⟦Api⟧): Component
    hide(destination: Name, apis: Set⟦Api⟧): Component
    extract(prereqs: Set⟦Api⟧): File
    isValidUpgrade(that: Component): Boolean
    abstract upgrade(result: Name, that: Component): Component
      requires self.isValidUpgrade(that)
    sourceIsAvailable: Boolean
    getSourceFile(): File
      requires sourceIsAvailable
    runTests(inclusive = Boolean): ()
end

trait Api extends FortressElement
    uses: Set⟦Api⟧
    extraction: File
end

trait Name end

trait SimpleComponent extends Component end

trait Version
    major: ℕ
    minor: ℕ
end

end
```

Chapter 40

Memory Sequences and Binary Words

These are the lowest-level data structures in Fortress, upon which all others are built. Even such conceptually "primitive" data types as $\mathbb{Z}$ and $\mathbb{Z}32$ and $\mathbb{R}64$ are defined in terms of memory sequences and binary words.

Consider, for example, the types $\mathrm{BinaryWord}[\![6]\!]$, $\mathbb{Z}64$, and $\mathbb{N}64$. All three may be regarded as 64-bit data objects. However, $\mathbb{Z}64$ causes the operator $<$ to compare 64-bit words as two's-complement signed integers, $\mathbb{N}64$ causes the operator $<$ to compare 64-bit words as unsigned integers, and $\mathrm{BinaryWord}[\![6]\!]$ does not support the operator $<$ at all—instead it has two methods named $signedLT$ and $unsignedLT$ (which are, of course, conveniently used to implement the operator $<$ for $\mathbb{Z}64$ and $\mathbb{N}64$). Moreover, $\mathbb{Z}64$ and $\mathbb{N}64$ support units and dimensions, but $\mathrm{BinaryWord}$ values do not. The parameterized type $\mathrm{BinaryWord}$ provides methods that are only a modest abstraction of operations supported by typical hardware instruction sets and serves as the lowest-level substrate that allows types such as $\mathbb{Z}64$ to be defined by libraries coded entirely in Fortress.

Similarly, the parameterized traits $\mathrm{LinearSequence}$ and $\mathrm{HeapSequence}$ describe the lowest-level data structures that are array-like or vector-like, capable of little more than one-dimensional indexing. They serve as the lowest-level substrate that allows the complete distributed and multi-dimensional array types to be defined by libraries coded entirely in Fortress.

For convenience, we use the term *binary linear sequence* to refer to a linear sequence of binary words, and the term *binary heap sequence* to refer to a heap sequence of binary words.

$\texttt{type}\ \mathrm{BinaryLinearSequence}[\![\texttt{nat}\ b, \texttt{nat}\ n]\!] = \mathrm{LinearSequence}[\![\mathrm{BinaryWord}[\![b]\!], n]\!]$
$\texttt{type}\ \mathrm{BinaryHeapSequence}[\![\texttt{nat}\ b]\!] = \mathrm{HeapSequence}[\![\mathrm{BinaryWord}[\![b]\!]]\!]$

Most operations on binary words do not depend on *endianness*, that is, in which order the bits are numbered. For operations that do depend on endianness, the parameterized trait $\mathrm{BinaryEndianWord}$ is provided.

It is also sometimes desirable to perform endianness-dependent operations on a linear sequence of binary words. For this purpose the specialized parameterized traits $\mathrm{BinaryLinearEndianSequence}$ and $\mathrm{BinaryEndianLinearEndianSequence}$ are provided; the former has $\mathrm{BinaryWord}$ values as elements, and the latter has $\mathrm{BinaryEndianWord}$ values as elements.

40.1 The Trait Fortress.Core.LinearSequence

A value of type $\text{LinearSequence}[\![T, n]\!]$ is a sequence of n things of type T, where n may be any natural number. Note that its length is statically fixed and may be described by a static expression. The general intent is that such a sequence will reside in a contiguous region of memory, typically belonging to a single processor or processor node, and that any element (indicated by an integer index) may be fetched or updated quickly by that processor or processor node.

If T is not a value type, then $\text{LinearSequence}[\![T, n]\!]$ describes a sequence of references, and a variable of type $\text{LinearSequence}[\![T, n]\!]$ occupies an amount of storage equal to n times the amount of storage required to hold a reference. If T is a value type, then $\text{LinearSequence}[\![T, n]\!]$ describes a sequence of "unboxed" values, and a variable of type $\text{LinearSequence}[\![T, n]\!]$ occupies an amount of storage equal to n times the amount of storage required to hold one value of type T.

Linear sequences, unlike arrays, are not too fancy. The main things you can do with linear sequences are subscripting and subscripted assignment, as well as assignment of entire sequences. They also support the concatenation operator $\|$. For example:

x: $\text{LinearSequence}[\![\text{Thread}, 3]\!]$
y: $\text{LinearSequence}[\![\text{Thread}, 6]\!]$
z: $\text{LinearSequence}[\![\text{Thread}, 5]\!] = x \| y[3\,\#\,2]$

Note in this example that the lengths are statically checkable. The range $3\,\#\,2$ is a range of constant size 2, so $y[3\,\#\,2]$ is known to be of type $\text{LinearSequence}[\![\text{Thread}, 2]\!]$. Indeed, only ranges of static size with element type IndexInt may be used to subscript a linear sequence.

```
value trait LinearSequence⟦T extends Object, nat n⟧ comprises {}
  coercion ⟦nat b, bool bigEndianSequence⟧
      (x: BinaryLinearEndianSequence⟦b, n, bigEndianSequence⟧)
    where { T extends BinaryWord⟦b⟧ }
  coercion ⟦nat b, bool bigEndianBytes, bool bigEndianBits, bool bigEndianSequence⟧
      (x: BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                           n, bigEndianSequence⟧)
    where { T extends BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧ }
  coercion ⟦nat b, bool bigEndianBytes, bool bigEndianBits, bool bigEndianSequence⟧
      (x: BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                           n, bigEndianSequence⟧)
    where { T extends BinaryWord⟦b⟧ }
  opr [j: IndexInt]: T throws { IndexOutOfBoundsException }
  opr ⟦nat k⟧[j: IntegerStatic⟦k⟧]: T where { k < n }
  opr ⟦nat m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧]: LinearSequence⟦T, m⟧
    throws { IndexOutOfBoundsException } where { m ≤ n }
  opr ⟦int a, nat m, int c⟧[r: StaticRange⟦a, m, c⟧]: LinearSequence⟦T, m⟧
    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
  opr [j: IndexInt] := (v: T): LinearSequence⟦T, n⟧ throws { IndexOutOfBoundsException }
  opr ⟦nat k⟧[j: IntegerStatic⟦k⟧] := (v: T): LinearSequence⟦T, n⟧ where { k < n }
  opr ⟦nat m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧] := (v: LinearSequence⟦T, m⟧):
      LinearSequence⟦T, n⟧
    throws { IndexOutOfBoundsException } where { m ≤ n }
  opr ⟦int a, nat m, int c⟧[r: StaticRange⟦a, m, c⟧] := (v: LinearSequence⟦T, m⟧):
      LinearSequence⟦T, n⟧
    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
  update(j: IndexInt, v: T): LinearSequence⟦T, n⟧ throws { IndexOutOfBoundsException }
```

```
    update⟦nat k⟧(j: IntegerStatic⟦k⟧, v: T): LinearSequence⟦T, n⟧ where { k < n }
    update⟦nat m⟧(r: RangeOfStaticSize⟦IndexInt, m⟧, v: LinearSequence⟦T, m⟧):
        LinearSequence⟦T, n⟧
      throws { IndexOutOfBoundsException } where { m ≤ n }
    update⟦int a, nat m, int c⟧(r: StaticRange⟦a, m, c⟧, v: LinearSequence⟦T, m⟧):
        LinearSequence⟦T, n⟧
      where { 0 ≤ a < n, 0 ≤ a + m · c < n }
    opr ‖⟦nat m⟧(self, other: LinearSequence⟦T, m⟧): LinearSequence⟦T, n + m⟧
    getter reverse(): LinearSequence⟦T, n⟧
    getter littleEndian⟦nat b⟧(): BinaryLinearEndianSequence⟦b, n, false⟧
      where { T extends BinaryWord⟦b⟧ }
    getter bigEndian⟦nat b⟧(): BinaryLinearEndianSequence⟦b, n, true⟧
      where { T extends BinaryWord⟦b⟧ }
    getter littleEndian⟦nat b, bool bigEndianBytes, bool bigEndianBits⟧():
        BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits, n, false⟧
      where { T extends BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧ }
    getter bigEndian⟦nat b, bool bigEndianBytes, bool bigEndianBits⟧():
        BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits, n, true⟧
      where { T extends BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧ }
  end
```

40.1.1 `coercion` ⟦`nat` b, `bool` *bigEndianSequence*⟧
(x: BinaryLinearEndianSequence⟦b, n, *bigEndianSequence*⟧)
`where` { T `extends` BinaryWord⟦b⟧ }

40.1.2 `coercion` ⟦`nat` b, `bool` *bigEndianBytes*, `bool` *bigEndianBits*, `bool` *bigEndianSequence*⟧
(x: BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧)
`where` { T `extends` BinaryEndianWord⟦b, *bigEndianBytes*, *bigEndianBits*⟧ }

Any binary linear endian sequence may be coerced to an ordinary binary linear sequence of corresponding element type. The bit values remain the same; all that is lost is the endianness information of the sequence in the static type.

40.1.3 `coercion` ⟦`nat` b, `bool` *bigEndianBytes*, `bool` *bigEndianBits*, `bool` *bigEndianSequence*⟧
(x: BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧)
`where` { T `extends` BinaryWord⟦b⟧ }

A binary linear endian sequence of binary endian words may be coerced to an ordinary binary linear sequence of ordinary binary words. The bit values remain the same; all that is lost is the endianness information of both the sequence and the elements.

40.1.4 `opr` $[j\colon \mathrm{IndexInt}]\colon T$ `throws` $\{\,\mathrm{IndexOutOfBoundsException}\,\}$
40.1.5 `opr` $[\![\texttt{nat}\ k]\!][j\colon \mathrm{IntegerStatic}[\![k]\!]]\colon T$ `where` $\{\,k < n\,\}$

Subscripting returns element j of this linear sequence. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $0 \le j < n$, where n is the length of the linear sequence. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

40.1.6 `opr` $[\![\texttt{nat}\ m]\!][r\colon \mathrm{RangeOfStaticSize}[\![\mathrm{IndexInt}, m]\!]]\colon \mathrm{LinearSequence}[\![T, m]\!]$
`throws` $\{\,\mathrm{IndexOutOfBoundsException}\,\}$ `where` $\{\,m \le n\,\}$
40.1.7 `opr` $[\![\texttt{int}\ a, \texttt{nat}\ m, \texttt{int}\ c]\!][r\colon \mathrm{StaticRange}[\![a, m, c]\!]]\colon \mathrm{LinearSequence}[\![T, m]\!]$
`where` $\{\,0 \le a < n, 0 \le a + m \cdot c < n\,\}$

Subscripting with a range of static size m returns the indicated subsequence of this linear sequence. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $r \subseteq 0 \# n$, where n is the length of the linear sequence. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time. Element k of the result sequence is the same as element $r.lowerBound + k \times r.stride$ of this linear sequence, for all $0 \le k < m$.

40.1.8 `opr` $[j\colon \mathrm{IndexInt}] := (v\colon T)\colon \mathrm{LinearSequence}[\![T, n]\!]$ `throws` $\{\,\mathrm{IndexOutOfBoundsException}\,\}$
40.1.9 `opr` $[\![\texttt{nat}\ k]\!][j\colon \mathrm{IntegerStatic}[\![k]\!]] := (v\colon T)\colon \mathrm{LinearSequence}[\![T, n]\!]$ `where` $\{\,k < n\,\}$

After subscripted value object assignment, element j of the subscripted variable is the same as the given value v, and all other elements are the same as before. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $0 \le j < n$, where n is the length of the linear sequence. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

40.1.10 `opr` $[\![\texttt{nat}\ m]\!][r\colon \mathrm{RangeOfStaticSize}[\![\mathrm{IndexInt}, m]\!]] := (v\colon \mathrm{LinearSequence}[\![T, m]\!])\colon$
$\mathrm{LinearSequence}[\![T, n]\!]$
`throws` $\{\,\mathrm{IndexOutOfBoundsException}\,\}$ `where` $\{\,m \le n\,\}$
40.1.11 `opr` $[\![\texttt{int}\ a, \texttt{nat}\ m, \texttt{int}\ c]\!][r\colon \mathrm{StaticRange}[\![a, m, c]\!]] := (v\colon \mathrm{LinearSequence}[\![T, m]\!])\colon$
$\mathrm{LinearSequence}[\![T, n]\!]$
`where` $\{\,0 \le a < n, 0 \le a + m \cdot c < n\,\}$

After subscripted value object assignment, elements of the subscripted variable selected by r are the same as corresponding elements of v, and all other elements are the same as before; specifically, element $r.lowerBound + k \times r.stride$ of the updated variable is the same as element k of v, for all $0 \le k < m$. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $r \subseteq 0 \# n$, where n is the length of the linear sequence. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time.

40.1.12 $update(j\colon \mathrm{IndexInt}, v\colon T)\colon \mathrm{LinearSequence}[\![T, n]\!]$ `throws` $\{\ \mathrm{IndexOutOfBoundsException}\ \}$
40.1.13 $update[\![\texttt{nat}\ k]\!](j\colon \mathrm{IntegerStatic}[\![k]\!], v\colon T)\colon \mathrm{LinearSequence}[\![T, n]\!]$ `where` $\{\ k < n\ \}$

This is a functional version of subscripted value object assignment: element j of the result is the same as the given value v, and all other elements are the same as before. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $0 \le j < n$, where n is the length of the linear sequence. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

40.1.14 $update[\![\texttt{nat}\ m]\!](r\colon \mathrm{RangeOfStaticSize}[\![\mathrm{IndexInt}, m]\!], v\colon \mathrm{LinearSequence}[\![T, m]\!])\colon$
$\mathrm{LinearSequence}[\![T, n]\!]$
`throws` $\{\ \mathrm{IndexOutOfBoundsException}\ \}$ `where` $\{\ m \le n\ \}$
40.1.15 $update[\![\texttt{int}\ a, \texttt{nat}\ m, \texttt{int}\ c]\!](r\colon \mathrm{StaticRange}[\![a, m, c]\!], v\colon \mathrm{LinearSequence}[\![T, m]\!])\colon$
$\mathrm{LinearSequence}[\![T, n]\!]$
`where` $\{\ 0 \le a < n, 0 \le a + m \cdot c < n\ \}$

This is a functional version of subscripted value object assignment: elements of the result selected by r are the same as corresponding elements of v, and all other elements are the same as before; specifically, element $r.lowerBound + k \times r.stride$ of the result is the same as element k of v, for all $0 \le k < m$. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $0 \le j < n$, where n is the length of the linear sequence. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time.

40.1.16 `opr` $\|\ [\![\texttt{nat}\ m]\!](\texttt{self}, other\colon \mathrm{LinearSequence}[\![T, m]\!])\colon \mathrm{LinearSequence}[\![T, n + m]\!]$

The result is a linear sequence whose length is equal to the sum of the lengths of this linear sequence and the other linear sequence. Element k of the result is the same as element k of this linear sequence if $0 \le k < n$, and is the same as element $k - n$ of the other linear sequence if $n \le k < n + m$.

40.1.17 `getter` $reverse()\colon \mathrm{LinearSequence}[\![T, n]\!]$

The result is a linear sequence such that element k of the result is the same as element $n - 1 - k$ of this linear sequence, for all $0 \le k < n$.

40.1.18 `getter` $littleEndian[\![\texttt{nat}\ b]\!]()\colon \mathrm{BinaryLinearEndianSequence}[\![b, n, false]\!]$
`where` $\{\ T$ `extends` $\mathrm{BinaryWord}[\![b]\!]\ \}$
40.1.19 `getter` $bigEndian[\![\texttt{nat}\ b]\!]()\colon \mathrm{BinaryLinearEndianSequence}[\![b, n, true]\!]$
`where` $\{\ T$ `extends` $\mathrm{BinaryWord}[\![b]\!]\ \}$
40.1.20 `getter` $littleEndian[\![\texttt{nat}\ b, \texttt{bool}\ bigEndianBytes, \texttt{bool}\ bigEndianBits]\!]()\colon$
$\mathrm{BinaryEndianLinearEndianSequence}[\![b, bigEndianBytes, bigEndianBits, n, false]\!]$
`where` $\{\ T$ `extends` $\mathrm{BinaryEndianWord}[\![b, bigEndianBytes, bigEndianBits]\!]\ \}$
40.1.21 `getter` $bigEndian[\![\texttt{nat}\ b, \texttt{bool}\ bigEndianBytes, \texttt{bool}\ bigEndianBits]\!]()\colon$
$\mathrm{BinaryEndianLinearEndianSequence}[\![b, bigEndianBytes, bigEndianBits, n, true]\!]$
`where` $\{\ T$ `extends` $\mathrm{BinaryEndianWord}[\![b, bigEndianBytes, bigEndianBits]\!]\ \}$

These conversion getters allow a linear sequence of (possibly endian) binary words to be treated as a specifically little-endian or specifically big-endian linear sequence. This is especially useful just before invoking an endianness-dependent method, for example $s.littleEndian.countLeadingZeroBits()$.

40.2 Constructing Linear Sequences

40.2.1 *makeLinearSequence*⟦T `extends` Object, `nat` n⟧($item$: T): LinearSequence⟦T, n⟧

A new linear sequence of length n is returned. Every element of the linear sequence is initialized to be the same as the given $item$.

40.2.2 *computeLinearSequence*⟦T `extends` Object, `nat` n⟧(f: IndexInt → T): LinearSequence⟦T, n⟧

A new linear sequence of length n is returned. Element j of the new linear sequence is initialized to a value computed by calling the given function f with argument j.

40.3 The Trait Fortress.Core.HeapSequence

A value of type HeapSequence⟦T⟧ is an array-like object that contains items of type T. The length of a heap sequence is in general not known statically, but can be discovered by asking for its length. A variable of type HeapSequence⟦T⟧ occupies the amount of storage required to hold a reference; this reference refers to an object that occupies an amount of storage greater than or equal to the amount that would be occupied by a variable of type LinearSequence⟦T, n⟧ where n is the length of the heap sequence.

Heap sequences, like linear sequences and unlike arrays, are not too fancy. The main things you can do with heap sequences are subscripting and subscripted assignment. Concatenation is *not* supported, because a basic principle of the low-level types is that none of the operations, other than explicit construction of a heap sequence, does any heap allocation. However, a range of static size may be used to index a heap sequence; the result is a linear sequence.

```
trait HeapSequence⟦T extends Object⟧ extends Object comprises {}
  opr [j: IndexInt]: T throws { IndexOutOfBoundsException }
  opr ⟦nat m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧]: LinearSequence⟦T, m⟧
    throws { IndexOutOfBoundsException }
  opr [j : IndexInt] := (v: T): () throws { IndexOutOfBoundsException }
  opr ⟦nat m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧] := (v: LinearSequence⟦T, m⟧): ()
    throws { IndexOutOfBoundsException }
  reverse(selfStart: IndexInt, length: IndexInt): () throws { IndexOutOfBoundsException }
  opr |self| : IndexInt
end
```

40.3.1 `opr` [j: IndexInt]: T `throws` { IndexOutOfBoundsException }

Subscripting returns element j of this heap sequence. Indexing is zero-origin; an IndexOutOfBoundsException is thrown unless $0 \le j < n$, where n is the length of the heap sequence.

40.3.2 `opr` $[\![$`nat` $m]\!][r\colon \mathrm{RangeOfStaticSize}[\![\mathrm{IndexInt}, m]\!]]\colon \mathrm{LinearSequence}[\![T, m]\!]$
`throws` $\{\,\mathrm{IndexOutOfBoundsException}\,\}$

Subscripting by a range returns the indicated subsequence of this heap sequence. The range must be a range of static size, and the result is returned as a linear sequence (not a heap sequence), so no heap allocation is performed. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $r \subseteq (0 : n-1)$, where n is the length of the heap sequence. Element k of the result linear sequence is the same as element $r.lowerBound + k \times r.stride$ of this heap sequence, for all $0 \le k < m$.

40.3.3 `opr` $[j : \mathrm{IndexInt}] := (v\colon T)\colon ()$ `throws` $\{\,\mathrm{IndexOutOfBoundsException}\,\}$

After subscripted assignment, element j of this heap sequence is the same as the given value v, and all other elements are the same as before. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $0 \le j < n$, where n is the length of the heap sequence.

40.3.4 `opr` $[\![$`nat` $m]\!][r\colon \mathrm{RangeOfStaticSize}[\![\mathrm{IndexInt}, m]\!]] := (v\colon \mathrm{LinearSequence}[\![T, m]\!])\colon ()$
`throws` $\{\,\mathrm{IndexOutOfBoundsException}\,\}$

After subscripted assignment using a range as a subscript, elements of the subscripted variable selected by r are the same as corresponding elements of v, and all other elements are the same as before; specifically, element $r.lowerBound + k \times r.stride$ of the updated variable is the same as element k of v, for all $0 \le k < m$. The range must be a range of static size, and the values to be assigned must be passed as linear sequence of the same size. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $r \subseteq (0 : n-1)$, where n is the length of the heap sequence.

40.3.5 $reverse(\mathit{selfStart}\colon \mathrm{IndexInt}, \mathit{length}\colon \mathrm{IndexInt})\colon ()$ `throws` $\{\,\mathrm{IndexOutOfBoundsException}\,\}$

Elements $\mathit{selfStart}$ through $\mathit{selfStart} + \mathit{length} - 1$, inclusive, are reversed in order, that is, rearranged so that the value originally stored at element $\mathit{selfStart} + j$ becomes element $\mathit{selfStart} + \mathit{length} - 1 - j$, for all $0 \le j < \mathit{length}$. Other elements of this heap sequence are unaffected.

40.3.6 `opr` $|$`self`$|$: $\mathrm{IndexInt}$

The length of this heap sequence is returned. Note that the size of a heap sequence is specified at run time when the heap sequence is created; once a heap sequence has been created, its length does not change.

40.4 Constructing Heap Sequences

40.4.1 *makeHeapSequence*$[\![T$ `extends` Object$]\!](n\colon$ IndexInt, $item\colon T)\colon$ HeapSequence$[\![T]\!]$
`throws` { NegativeLengthException }

A new heap sequence of length n is allocated and returned. A NegativeLengthException is thrown if $n < 0$. Every element of the heap sequence is initialized to be the same as the given $item$.

40.4.2 *computeHeapSequence*$[\![T$ `extends` Object$]\!](n\colon$ IndexInt, $f\colon$ IndexInt $\rightarrow T)\colon$ HeapSequence$[\![T]\!]$
`throws` { NegativeLengthException }

A new heap sequence of length n is allocated and returned. A NegativeLengthException is thrown if $n < 0$. Element j of the new heap sequence is initialized to a value computed by calling the given function f with argument j.

40.5 The Trait Fortress.Core.BinaryWord

A value of type $\mathrm{BinaryWord}[\![b]\!]$ is a binary word of 2^b bits; b may be any natural number, so $\mathrm{BinaryWord}[\![0]\!]$ is a bit, $\mathrm{BinaryWord}[\![3]\!]$ is a byte, $\mathrm{BinaryWord}[\![6]\!]$ is a 64-bit word, and $\mathrm{BinaryWord}[\![10]\!]$ is a 1024-bit word. In fact, for convenience, the type abbreviations Bit and Byte are defined:

```
type Bit = BinaryWord⟦0⟧
type Byte = BinaryWord⟦3⟧
```

The type $\mathrm{BinaryWord}[\![b]\!]$ has $2^{(2^b)}$ distinct values. When the binary word is regarded as an unsigned integer, these values are identified with the nonnegative integers that are less than $2^{(2^b)}$. A binary word may also be regarded as a signed integer: a value that, when regarded as an unsigned integer, is identified with an integer less than $2^{(2^b-1)}$, is identified with that same integer when regarded as a signed integer; but a value that, when regarded as an unsigned integer, is identified with an integer not less than $2^{(2^b-1)}$, is identified with that same integer less $2^{(2^b)}$. (This is the standard "two's complement" representation for signed integers.)

A binary word of one bit can have one of two values, 0 or 1. A binary word of more than one bit has two halves, a high half and a low half, which are binary words of half the size. If v is the unsigned integer value of a binary word of 2^b bits, $b \geq 1$, h is the unsigned integer value of its high half, and l is the unsigned integer value of its low half, then $v = h \cdot 2^{2^{b-1}} + l$.

Operations on binary words include bitwise boolean operations, arithmetic operations, shifts and rotates, population count, and counting of leading and trailing zeros. The type $\mathrm{BinaryWord}[\![b]\!]$ is not "endian" and has no operations that depend on endianness. However, if w is binary word, then $w.\mathit{littleEndian}$ is a little-endian version of w and $w.\mathit{bigEndian}$ is a big-endian version of w; for example, if w is of type $\mathrm{BinaryWord}[\![6]\!]$, then $w.\mathit{littleEndian}_{63}$ is the most significant bit (the sign bit if the word is regarded as a two's-complement integer), and $w.\mathit{bigEndian}_0$ is that same bit.

```
trait BinaryWord⟦nat b⟧ extends { BasicBinaryWordOperations⟦BinaryWord⟦b⟧⟧ }
      comprises {}
      where { b ≤ maxBinaryWordBitLog }
    coercion ⟦int r⟧(x: IntegerStatic⟦r⟧) where { -2^(b-1) ≤ r < 2^b }
    coercion ⟦bool bigEndianBytes, bool bigEndianBits⟧
        (x: BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧)
    coercion ⟦nat b', nat n, bool bigEndianSequence⟧
        (x: BinaryLinearEndianSequence⟦b', n, bigEndianSequence⟧)
      where { 2^b = n · 2^b' }
    coercion ⟦nat b', bool bigEndianBytes, bool bigEndianBits, nat n, bool bigEndianSequence⟧
        (x: BinaryEndianLinearEndianSequence⟦b', bigEndianBytes, bigEndianBits,
                                              n, bigEndianSequence⟧)
      where { 2^b = n · 2^b' }
    bit(m: IndexInt): Bit
    getter lowHalf(): BinaryWord⟦b - 1⟧ where { b > 0 }
    getter highHalf(): BinaryWord⟦b - 1⟧ where { b > 0 }
    opr ‖⟦nat m⟧(self, other: BinaryWord⟦b⟧): BinaryWord⟦b + 1⟧
      where { b < maxBinaryWordBitLog }
    bitShuffle(other: BinaryWord⟦b⟧): BinaryWord⟦b + 1⟧
      where { b < maxBinaryWordBitLog }
    bitUnshuffle(): (BinaryWord⟦b - 1⟧, BinaryWord⟦b - 1⟧) where { b > 0 }
end
```

40.5.1 `coercion` $[\![$`int` $r]\!](x\colon \mathrm{IntegerStatic}[\![r]\!])$ `where` $\{\, -2^{b-1} \leq r < 2^b \,\}$

An static integer may be coerced to a binary word that corresponds to that integer value when interpreted as either a signed integer or an unsigned integer. For example, the type $\mathrm{BinaryWord}[\![3]\!]$ has $2^{(2^3)} = 256$ distinct binary word values; when they are interpreted as signed integers, the integer values range from -128 to 127, and when they are interpreted as unsigned integers, the integer values range from 0 to 255. Therefore any static integer from -128 to 255 may be coerced to type $\mathrm{BinaryWord}[\![3]\!]$.

40.5.2 `coercion` $[\![$`bool` $\mathit{bigEndianBytes}$, `bool` $\mathit{bigEndianBits}]\!]$
$(x\colon \mathrm{BinaryEndianWord}[\![b, \mathit{bigEndianBytes}, \mathit{bigEndianBits}]\!])$

Any binary endian word may be coerced to a plain binary word of the same size and value. In effect, this coercion merely discards the endianness information.

40.5.3 `coercion` $[\![$`nat` b', `nat` n, `bool` $\mathit{bigEndianSequence}]\!]$
$(x\colon \mathrm{BinaryLinearEndianSequence}[\![b', n, \mathit{bigEndianSequence}]\!])$
`where` $\{\, 2^b = n \cdot 2^{b'} \,\}$

A binary linear endian sequence of smaller binary words may be coerced to a single binary word, provided that the length of the linear sequence is an appropriate power of two, so that the total number of bits in the sequence is the same as the total number of bits in the resulting binary word. The manner in which the elements of the sequence are used to form the new binary word value respects the endianness of the sequence, so that element 0 of the sequence supplies the most significant bits of the result if $\mathit{bigEndianSequence}$ is true, but supplies the least significant bits of the result if $\mathit{bigEndianSequence}$ is false.

40.5.4 `coercion` $[\![$`nat` b', `bool` $\mathit{bigEndianBytes}$, `bool` $\mathit{bigEndianBits}$, `nat` n, `bool` $\mathit{bigEndianSequence}]\!]$
$(x\colon \mathrm{BinaryEndianLinearEndianSequence}[\![b', \mathit{bigEndianBytes}, \mathit{bigEndianBits},$
$n, \mathit{bigEndianSequence}]\!])$
`where` $\{\, 2^b = n \cdot 2^{b'} \,\}$

A binary endian linear endian sequence of smaller binary endian words may be coerced to a single binary word, provided that the length of the linear sequence is an appropriate power of two, so that the total number of bits in the sequence is the same as the total number of bits in the resulting binary word. The manner in which the elements of the sequence are used to form the new binary word value respects the endianness of the sequence, so that element 0 of the sequence supplies the most significant bits of the result if $\mathit{bigEndianSequence}$ is true, but supplies the least significant bits of the result if $\mathit{bigEndianSequence}$ is false. In effect, the "bytes and bits" endianness information is simply ignored and discarded.

40.5.5 $bit(m\colon \mathrm{IndexInt})\colon \mathrm{Bit}$

The result is a bit whose value (0 or 1) is equal to $\lfloor v \cdot 2^{-m} \rfloor \bmod 2$ where v is the value of the binary word regarded as an unsigned integer. This formula holds for any value of m; note that if m is negative or greater than $2^b - 1$, the result will always be a 0-bit. Thus the bit method provides a kind of "little-endian" indexing of the bits of a binary word, even a binary word whose type is not intrinsically endian, but it does not require that the bit number identify an actual represented bit of the binary word.

The bit method is particularly useful for describing the behavioral properties of other methods of binary data.

40.5.6 `getter` $lowHalf()\colon \mathrm{BinaryWord}[\![b-1]\!]$ `where` $\{\, b > 0 \,\}$
40.5.7 `getter` $highHalf()\colon \mathrm{BinaryWord}[\![b-1]\!]$ `where` $\{\, b > 0 \,\}$

The getters $lowHalf$ and $highHalf$ each return a binary word of half the size (in bits) of this binary word; $lowHalf$ returns the less significant bits, and $highHalf$ returns the more significant bits.

`property` $\forall(v) \bigwedge_{m \leftarrow 0\#2^{b-1}} v.lowHalf.bit(m) = v.bit(m)$
`property` $\forall(v) \bigwedge_{m \leftarrow 0\#2^{b-1}} v.highHalf.bit(m) = v.bit(m + 2^{b-1})$

40.5.8 `opr` $\|\, [\![\mathtt{nat}\ m]\!](\mathtt{self}, other\colon \mathrm{BinaryWord}[\![b]\!])\colon \mathrm{BinaryWord}[\![b+1]\!]$
`where` $\{\, b < maxBinaryWordBitLog \,\}$

The result of concatenating two binary words of size 2^b is a single binary word of size 2^{b+1}. The left-hand operand becomes the high (more significant) half of the result and the right-hand operand becomes the low (less significant) half of the result.

`property` $\forall(v, w)\ (v \,\|\, w).highHalf = v$
`property` $\forall(v, w)\ (v \,\|\, w).lowHalf = w$

40.5.9 $bitShuffle(other\colon \mathrm{BinaryWord}[\![b]\!])\colon \mathrm{BinaryWord}[\![b+1]\!]$
`where` $\{\, b < maxBinaryWordBitLog \,\}$

The bit-shuffle operation interleaves the bits of two words, as if shuffling cards together (using what magicians call a "perfect shuffle"). The result of shuffling the bits two binary words of size 2^b is a single binary word of size 2^{b+1}. This binary word provides the odd-numbered bits of the result and the other binary word provides the even-numbered bits of the result. For example, shuffling `1111` and `0000` produces `10101010`.

`property` $\forall(v, w, m\colon \mathrm{IndexInt})\ v.bitShuffle(w).bit(m) = (\mathtt{if}\ odd\ m\ \mathtt{then}\ v.bit((m-1)/2)\ \mathtt{else}\ w.bit(m/2))$

40.5.10 $bitUnshuffle()\colon (\mathrm{BinaryWord}[\![b-1]\!], \mathrm{BinaryWord}[\![b-1]\!])$ `where` $\{\, b > 0 \,\}$

This is the inverse of the $bitShuffle$ method: the odd-numbered bits of this binary word are used to form a binary word of half the size, and likewise the even-numbered bits, and a tuple of the two binary words is returned.

`property` $\forall(v, w)\ v.bitShuffle(w).bitUnshuffle() = (v, w)$

40.6 The Trait Fortress.Core.BinaryEndianWord

The type BinaryEndianWord⟦b, *bigEndianBytes*, *bigEndianBits*⟧ is exactly like BinaryWord⟦b⟧ but bears two kinds of endianness information. A BinaryEndianWord may be split into a sequence of smaller words; the result is of type BinaryLinearEndianSequence. The flag *bigEndianBytes* indicates whether subword 0 is the most significant subword (if *bigEndianBytes* is true) or least significant subword (if *bigEndianBytes* is false) of the original binary word. A BinaryEndianWord may also be subscripted to extract a bit or a bit field; the flag *bigEndianBits* indicates whether bit 0 is the most significant bit (if *bigEndianBits* is true) or least significant bit (if *bigEndianBits* is false) of the original binary word. (Yes, it may seem strange for the bit ordering to differ from the subword ordering, but they do differ on a number of architectures, including SPARC.) Extracting a bit produces a Bit, that is, a BinaryWord⟦0⟧. Extracting a bit field of width k produces a BinaryEndianLinearEndianSequence with $n = k$ and $b = 0$; the endianness of the sequence matches the bit-endianness of the original BinaryEndianWord.

```
trait BinaryEndianWord⟦nat b, bool bigEndianBytes, bool bigEndianBits⟧
      extends { BasicBinaryWordOperations⟦BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧, b⟧}
      comprises {}
      where { b ≤ maxBinaryWordBitLog }
    coercion ⟦int r⟧(x: IntegerStatic⟦r⟧) where { −2^(b−1) ≤ r < 2^b }
    opr [j: IndexInt] : BinaryEndianWord⟦1, bigEndianBytes, bigEndianBits⟧
      throws { IndexOutOfBoundsException }
    opr ⟦nat k⟧[j: IntegerStatic⟦k⟧] : BinaryEndianWord⟦1, bigEndianBytes, bigEndianBits⟧
      where { k < 2^b }
    opr ⟦nat m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧] :
        BinaryEndianLinearEndianSequence⟦1, bigEndianBytes, bigEndianBits, m, bigEndianBits⟧
      throws { IndexOutOfBoundsException }
    opr ⟦int a, nat m, int c⟧[r: StaticRange⟦a, m, c⟧] :
        BinaryEndianLinearEndianSequence⟦1, bigEndianBytes, bigEndianBits, m, bigEndianBits⟧
      where { 0 ≤ a < 2^b, 0 ≤ a + m · c < 2^b }
    opr [j: IndexInt] := (v: Bit):
        BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧
      throws { IndexOutOfBoundsException }
    opr ⟦nat k⟧[j: IntegerStatic⟦k⟧] := (v: Bit):
        BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧
      where { k < 2^b }
    opr ⟦nat m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧] := (v: BinaryLinearSequence⟦1, m⟧):
        BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧
      throws { IndexOutOfBoundsException }
    opr ⟦int a, nat m, int c⟧[r: StaticRange⟦a, m, c⟧] := (v: BinaryLinearSequence⟦1, k⟧):
        BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧
      where { 0 ≤ a < 2^b, 0 ≤ a + m · c < 2^b }
    update(j: IndexInt, v: Bit):
        BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧
      throws { IndexOutOfBoundsException }
    update⟦nat k⟧(j: IntegerStatic⟦k⟧, v: Bit):
        BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧
      where { k < 2^b }
    update⟦nat m⟧(r: RangeOfStaticSize⟦IndexInt, m⟧, v: BinaryLinearSequence⟦1, m⟧):
        BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧
      throws { IndexOutOfBoundsException }
    update⟦int a, nat m, int c⟧(r: StaticRange⟦a, m, c⟧, v: BinaryLinearSequence⟦1, m⟧):
        BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧
      where { 0 ≤ a < 2^b, 0 ≤ a + m · c < 2^b }
```

```
    lowHalf(): BinaryEndianWord⟦b − 1, bigEndianBytes, bigEndianBits⟧ where { b > 0 }
    highHalf(): BinaryEndianWord⟦b − 1, bigEndianBytes, bigEndianBits⟧ where { b > 0 }
    opr ‖⟦nat m, bool bigEndianSequence⟧
        (self, other: BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧):
        BinaryEndianLinearEndianSequence⟦b + 1, bigEndianBytes, bigEndianBits,
                                         2, bigEndianSequence⟧
      where { bigEndianSequence = bigEndianBytes }
    opr ‖⟦nat m, bool bigEndianSequence, nat radix, nat q, nat k, nat v⟧
        (self, other: NaturalNumeral⟦m, radix, v⟧):
        BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                         k + 1, bigEndianSequence⟧
      where { bigEndianSequence = bigEndianBytes, radix = 2^q, q · m = k · 2^b }
    opr ‖⟦nat m, bool bigEndianSequence, nat radix, nat q, nat k, nat v⟧
        (other: NaturalNumeral⟦m, radix, v⟧, self):
        BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                         k + 1, bigEndianSequence⟧
      where { bigEndianSequence = bigEndianBytes, radix = 2^q, q · m = k · 2^b }
    bitShuffle(other: BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧):
      BinaryEndianWord⟦b + 1, bigEndianBytes, bigEndianBits⟧
      where { b < maxBinaryWordBitLog }
    bitUnshuffle(): (BinaryEndianWord⟦b − 1, bigEndianBytes, bigEndianBits⟧,
                     BinaryEndianWord⟦b − 1, bigEndianBytes, bigEndianBits⟧)
      where { b > 0 }
    littleEndian(): BinaryEndianWord⟦b, false, false⟧
    bigEndian(): BinaryEndianWord⟦b, true, true⟧
end
```

40.6.1 `coercion` ⟦`int` r⟧(x: IntegerStatic⟦r⟧) `where` $\{ -2^{b-1} \le r < 2^b \}$

An static integer may be coerced to a binary endian word exactly as if it were coerced to a plain binary word of the same size; the endian numbering of the bytes and bits does not affect which binary word value is produced from the static integer.

40.6.2 `opr` [j: IndexInt] : BinaryEndianWord⟦1, *bigEndianBytes*, *bigEndianBits*⟧ `throws` { IndexOutOfBoundsException }

40.6.3 `opr` ⟦`nat` k⟧[j: IntegerStatic⟦k⟧] : BinaryEndianWord⟦1, *bigEndianBytes*, *bigEndianBits*⟧ `where` $\{ k < 2^b \}$

Subscripting returns bit j of this binary endian word. The numbering of the bits is dictated by *bigEndianBits*. Indexing is zero-origin; an IndexOutOfBoundsException is thrown unless $0 \le j < 2^b$. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

`property` $\forall(v) \bigwedge_{j \leftarrow 0 \# 2^b} v_j = (\texttt{if}\ bigEndianBits\ \texttt{then}\ v.bit(2^b - 1 - j)\ \texttt{else}\ v.bit(j)\ \texttt{end})$

40.6.4 `opr` $[\![\texttt{nat}\ m]\!][r\colon \mathrm{RangeOfStaticSize}[\![\mathrm{IndexInt}, m]\!]]$:
$\mathrm{BinaryEndianLinearEndianSequence}[\![1, \mathit{bigEndianBytes}, \mathit{bigEndianBits}, m, \mathit{bigEndianBits}]\!]$
`throws` $\{\ \mathrm{IndexOutOfBoundsException}\ \}$

40.6.5 `opr` $[\![\texttt{int}\ a, \texttt{nat}\ m, \texttt{int}\ c]\!][r\colon \mathrm{StaticRange}[\![a, m, c]]\!]$:
$\mathrm{BinaryEndianLinearEndianSequence}[\![1, \mathit{bigEndianBytes}, \mathit{bigEndianBits}, m, \mathit{bigEndianBits}]\!]$
`where` $\{\ 0 \le a < 2^b, 0 \le a + m \cdot c < 2^b\ \}$

Subscripting with a range of static size m returns the indicated subsequence of bits of this binary endian word. The numbering of the bits is dictated by $\mathit{bigEndianBits}$. The result is a binary endian linear endian sequence of bits whose sequence endianness is the same as the bit endianness of this binary endian word. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $r \subseteq 0 \# 2^b$. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time. Element k of the result sequence is the same as the bit that would be selected from this binary endian word by subscripting it with $r.\mathit{lowerBound} + k \times r.\mathit{stride}$, for all $0 \le k < m$.

40.6.6 `opr` $[j\colon \mathrm{IndexInt}] := (v\colon \mathrm{Bit})$:
$\mathrm{BinaryEndianWord}[\![b, \mathit{bigEndianBytes}, \mathit{bigEndianBits}]\!]$
`throws` $\{\ \mathrm{IndexOutOfBoundsException}\ \}$

40.6.7 `opr` $[\![\texttt{nat}\ k]\!][j\colon \mathrm{IntegerStatic}[\![k]]\!] := (v\colon \mathrm{Bit})$:
$\mathrm{BinaryEndianWord}[\![b, \mathit{bigEndianBytes}, \mathit{bigEndianBits}]\!]$
`where` $\{\ k < 2^b\ \}$

After subscripted value object assignment, the bit that would be selected from this binary endian word by subscripting it with j is the same as the given bit v, and all other bits are the same as before. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $0 \le j < 2^b$. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

40.6.8 `opr` $[\![\texttt{nat}\ m]\!][r\colon \mathrm{RangeOfStaticSize}[\![\mathrm{IndexInt}, m]\!]] := (v\colon \mathrm{BinaryLinearSequence}[\![1, m]\!])$:
$\mathrm{BinaryEndianWord}[\![b, \mathit{bigEndianBytes}, \mathit{bigEndianBits}]\!]$
`throws` $\{\ \mathrm{IndexOutOfBoundsException}\ \}$

40.6.9 `opr` $[\![\texttt{int}\ a, \texttt{nat}\ m, \texttt{int}\ c]\!][r\colon \mathrm{StaticRange}[\![a, m, c]]\!] := (v\colon \mathrm{BinaryLinearSequence}[\![1, m]\!])$:
$\mathrm{BinaryEndianWord}[\![b, \mathit{bigEndianBytes}, \mathit{bigEndianBits}]\!]$
`where` $\{\ 0 \le a < 2^b, 0 \le a + m \cdot c < 2^b\ \}$

After subscripted value object assignment, bits that would be selected from this binary endian word by subscripting it with r are the same as corresponding elements of v, and all other bits are the same as before; specifically, the bit that would be selected from this binary endian word by subscripting it with $r.\mathit{lowerBound} + k \times r.\mathit{stride}$ is the same as element k of v, for all $0 \le k < m$. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $r \subseteq 0 \# 2^b$. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time.

40.6.10 $update(j\colon \mathrm{IndexInt}, v\colon \mathrm{Bit})$:
$\mathrm{BinaryEndianWord}[\![b, bigEndianBytes, bigEndianBits]\!]$
`throws` $\{\,\mathrm{IndexOutOfBoundsException}\,\}$

40.6.11 $update[\![\texttt{nat}\ k]\!](j\colon \mathrm{IntegerStatic}[\![k]\!], v\colon \mathrm{Bit})$:
$\mathrm{BinaryEndianWord}[\![b, bigEndianBytes, bigEndianBits]\!]$
`where` $\{\,k < 2^b\,\}$

This is a functional version of subscripted value object assignment: the bit that would be selected from the result by subscripting it with j is the same as the given bit v, and all other bits are the same as before. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $0 \le j < 2^b$. If the subscript is a static expression, then its validity is checked statically, and no exception will occur at run time.

40.6.12 $update[\![\texttt{nat}\ m]\!](r\colon \mathrm{RangeOfStaticSize}[\![\mathrm{IndexInt}, m]\!], v\colon \mathrm{BinaryLinearSequence}[\![1, m]\!])$:
$\mathrm{BinaryEndianWord}[\![b, bigEndianBytes, bigEndianBits]\!]$
`throws` $\{\,\mathrm{IndexOutOfBoundsException}\,\}$

40.6.13 $update[\![\texttt{int}\ a, \texttt{nat}\ m, \texttt{int}\ c]\!](r\colon \mathrm{StaticRange}[\![a, m, c]\!], v\colon \mathrm{BinaryLinearSequence}[\![1, m]\!])$:
$\mathrm{BinaryEndianWord}[\![b, bigEndianBytes, bigEndianBits]\!]$
`where` $\{\,0 \le a < 2^b, 0 \le a + m \cdot c < 2^b\,\}$

This is a functional version of subscripted value object assignment: bits that would be selected from the result by subscripting it with r are the same as corresponding elements of v, and all other bits are the same as before; specifically, the bit that would be selected from the result by subscripting it with $r.lowerBound + k \times r.stride$ is the same as element k of v, for all $0 \le k < m$. Indexing is zero-origin; an $\mathrm{IndexOutOfBoundsException}$ is thrown unless $r \subseteq 0 \# 2^b$. If the subscript is a static range, then its validity is checked statically, and no exception will occur at run time.

40.6.14 $lowHalf\,()\colon \mathrm{BinaryEndianWord}[\![b-1, bigEndianBytes, bigEndianBits]\!]$ `where` $\{\,b > 0\,\}$
40.6.15 $highHalf\,()\colon \mathrm{BinaryEndianWord}[\![b-1, bigEndianBytes, bigEndianBits]\!]$ `where` $\{\,b > 0\,\}$

The getters *lowHalf* and *highHalf* each return a binary endian word of half the size (in bits) of this binary endian word, and with the same endian characteristics; *lowHalf* returns the less significant bits, and *highHalf* returns the more significant bits.

`property` $\forall(v) \bigwedge_{m \leftarrow 0 \# 2^{b-1}} v.lowHalf.bit(m) = v.bit(m)$
`property` $\forall(v) \bigwedge_{m \leftarrow 0 \# 2^{b-1}} v.highHalf.bit(m) = v.bit(m + 2^{b-1})$
`property` $\forall(v) \bigwedge_{m \leftarrow 0 \# 2^{b-1}} v.lowHalf_{\,m} = v[\texttt{if}\ bigEndianBits\ \texttt{then}\ m + 2^{b-1}\ \texttt{else}\ m\ \texttt{end}]$
`property` $\forall(v) \bigwedge_{m \leftarrow 0 \# 2^{b-1}} v.highHalf_{\,m} = v[\texttt{if}\ bigEndianBits\ \texttt{then}\ m\ \texttt{else}\ m + 2^{b-1}\ \texttt{end}]$

40.6.16 `opr` ‖ ⟦`nat` m, `bool` *bigEndianSequence*⟧
(`self`, *other*: BinaryEndianWord⟦b, *bigEndianBytes*, *bigEndianBits*⟧):
BinaryEndianLinearEndianSequence⟦$b + 1$, *bigEndianBytes*, *bigEndianBits*,
2, *bigEndianSequence*⟧
`where` { *bigEndianSequence* = *bigEndianBytes* }

[Description to be supplied.]

40.6.17 `opr` ‖ ⟦`nat` m, `bool` *bigEndianSequence*, `nat` *radix*, `nat` q, `nat` k, `nat` v⟧
(`self`, *other*: NaturalNumeral⟦m, *radix*, v⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*,
$k + 1$, *bigEndianSequence*⟧
`where` { *bigEndianSequence* = *bigEndianBytes*, *radix* = 2^q, $q \cdot m = k \cdot 2^b$ }

[Description to be supplied.]

40.6.18 `opr` ‖ ⟦`nat` m, `bool` *bigEndianSequence*, `nat` *radix*, `nat` q, `nat` k, `nat` v⟧
(*other*: NaturalNumeral⟦m, *radix*, v⟧, `self`):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*,
$k + 1$, *bigEndianSequence*⟧
`where` { *bigEndianSequence* = *bigEndianBytes*, *radix* = 2^q, $q \cdot m = k \cdot 2^b$ }

[Description to be supplied.]

40.6.19 *bitShuffle*(*other*: BinaryEndianWord⟦b, *bigEndianBytes*, *bigEndianBits*⟧):
BinaryEndianWord⟦$b + 1$, *bigEndianBytes*, *bigEndianBits*⟧
`where` { $b <$ *maxBinaryWordBitLog* }

[Description to be supplied.]

40.6.20 *bitUnshuffle*(): (BinaryEndianWord⟦$b - 1$, *bigEndianBytes*, *bigEndianBits*⟧,
BinaryEndianWord⟦$b - 1$, *bigEndianBytes*, *bigEndianBits*⟧)
`where` { $b > 0$ }

[Description to be supplied.]

40.6.21 *littleEndian*(): BinaryEndianWord⟦b, *false*, *false*⟧
40.6.22 *bigEndian*(): BinaryEndianWord⟦b, *true*, *true*⟧

[Description to be supplied.]

40.7 The Trait Fortress.Core.BasicBinaryOperations

`trait` BasicBinaryOperations⟦T `extends` BasicBinaryOperations⟦T⟧⟧
$wrappingAdd(other\colon T)\colon T$
$add(other\colon T, carryIn\colon \text{Bit} = 0)\colon (T, \text{Bit})$
$signedAdd(other\colon T, overflowAction\colon () \to T)\colon T$
$unsignedAdd(other\colon T, overflowAction\colon () \to T)\colon T$
$saturatingSignedAdd(other\colon T)\colon T$
$saturatingUnsignedAdd(other\colon T)\colon T$
$wrappingSubtract(other\colon T)\colon T$
$subtract(other\colon T, carryIn\colon \text{Bit} = 1)\colon (T, \text{Bit})$
$signedSubtract(other\colon T, overflowAction\colon () \to T)\colon T$
$unsignedSubtract(other\colon T, overflowAction\colon () \to T)\colon T$
$saturatingSignedSubtract(other\colon T)\colon T$
$saturatingUnsignedSubtract(other\colon T)\colon T$
$wrappingNegate()\colon T$
$negate(carryIn\colon \text{Bit} = 1)\colon (T, \text{Bit})$
$signedNegate(overflowAction\colon () \to T)\colon T$
$unsignedNegate(overflowAction\colon () \to T)\colon T$
$saturatingSignedNegate()\colon T$
$bitNot()\colon T$
$bitAnd(other\colon T)\colon T$
$bitOr(other\colon T)\colon T$
$bitXor(other\colon T)\colon T$
$bitXorNot(other\colon T)\colon T$
$bitNand(other\colon T)\colon T$
$bitNor(other\colon T)\colon T$
$bitAndNot(other\colon T)\colon T$
$bitOrNot(other\colon T)\colon T$
$signedMax(other\colon T)\colon T$
$signedMin(other\colon T)\colon T$
$unsignedMax(other\colon T)\colon T$
$unsignedMin(other\colon T)\colon T$
`opr` =(`self`, $other\colon T)\colon$ Boolean
`opr` ≠(`self`, $other\colon T)\colon$ Boolean
$signedLT(other\colon T)\colon$ Boolean
$signedLE(other\colon T)\colon$ Boolean
$signedGE(other\colon T)\colon$ Boolean
$signedGT(other\colon T)\colon$ Boolean
$unsignedLT(other\colon T)\colon$ Boolean
$unsignedLE(other\colon T)\colon$ Boolean
$unsignedGE(other\colon T)\colon$ Boolean
$unsignedGT(other\colon T)\colon$ Boolean
$signedShift(j\colon \text{IndexInt})\colon T$
$signedShift(j\colon \text{IndexInt}, overflowAction\colon () \to T)\colon T$
$saturatingSignedShift(j\colon \text{IndexInt})\colon T$
$unsignedShift(j\colon \text{IndexInt})\colon T$
$unsignedShift(j\colon \text{IndexInt}, overflowAction\colon () \to T)\colon T$
$saturatingUnsignedShift(j\colon \text{IndexInt})\colon T$
$bitRotate(j\colon \text{IndexInt})\colon T$
$countOneBits()\colon$ IndexInt

```
    countLeadingZeroBits(): IndexInt
    countTrailingZeroBits(): IndexInt
    leftmostOneBit(): T
    rightmostOneBit(): T
    bitReverse(): T
    signedIndex(): IndexInt throws { IntegerOverflowException }
    unsignedIndex(): IndexInt throws { IntegerOverflowException }
    gatherBits(mask: T): T
    spreadBits(mask: T): T
    disentangleBits(mask: T): T
    intersperseBits(mask: T): T
end
```

40.7.1 *wrappingAdd*(*other*: T): T
40.7.2 *add*(*other*: T, *carryIn*: Bit $= 0$): (T, Bit)
40.7.3 *signedAdd*(*other*: T, *overflowAction*: $() \to T$): T
40.7.4 *unsignedAdd*(*other*: T, *overflowAction*: $() \to T$): T
40.7.5 *saturatingSignedAdd*(*other*: T): T
40.7.6 *saturatingUnsignedAdd*(*other*: T): T

[Description to be supplied.]

40.7.7 *wrappingSubtract*(*other*: T): T
40.7.8 *subtract*(*other*: T, *carryIn*: Bit $= 1$): (T, Bit)
40.7.9 *signedSubtract*(*other*: T, *overflowAction*: $() \to T$): T
40.7.10 *unsignedSubtract*(*other*: T, *overflowAction*: $() \to T$): T
40.7.11 *saturatingSignedSubtract*(*other*: T): T
40.7.12 *saturatingUnsignedSubtract*(*other*: T): T

[Description to be supplied.]

40.7.13 *wrappingNegate*(): T
40.7.14 *negate*(*carryIn*: Bit $= 1$): (T, Bit)
40.7.15 *signedNegate*(*overflowAction*: $() \to T$): T
40.7.16 *unsignedNegate*(*overflowAction*: $() \to T$): T
40.7.17 *saturatingSignedNegate*(): T

[Description to be supplied.]

40.7.18 *bitNot*(): T

[Description to be supplied.]

40.7.19 $bitAnd(other\colon T)\colon T$
40.7.20 $bitOr(other\colon T)\colon T$
40.7.21 $bitXor(other\colon T)\colon T$
40.7.22 $bitXorNot(other\colon T)\colon T$
40.7.23 $bitNand(other\colon T)\colon T$
40.7.24 $bitNor(other\colon T)\colon T$
40.7.25 $bitAndNot(other\colon T)\colon T$
40.7.26 $bitOrNot(other\colon T)\colon T$

[Description to be supplied.]

40.7.27 $signedMax(other\colon T)\colon T$
40.7.28 $signedMin(other\colon T)\colon T$

[Description to be supplied.]

40.7.29 $unsignedMax(other\colon T)\colon T$
40.7.30 $unsignedMin(other\colon T)\colon T$

[Description to be supplied.]

40.7.31 `opr` $=$(`self`, $other\colon T)\colon$ Boolean
40.7.32 `opr` $\neq$(`self`, $other\colon T)\colon$ Boolean

[Description to be supplied.]

40.7.33 $signedLT(other\colon T)\colon$ Boolean
40.7.34 $signedLE(other\colon T)\colon$ Boolean
40.7.35 $signedGE(other\colon T)\colon$ Boolean
40.7.36 $signedGT(other\colon T)\colon$ Boolean

[Description to be supplied.]

40.7.37 $unsignedLT(other\colon T)\colon$ Boolean
40.7.38 $unsignedLE(other\colon T)\colon$ Boolean
40.7.39 $unsignedGE(other\colon T)\colon$ Boolean
40.7.40 $unsignedGT(other\colon T)\colon$ Boolean

[Description to be supplied.]

40.7.41 *signedShift*(*j*: IndexInt): *T*
40.7.42 *signedShift*(*j*: IndexInt, *overflowAction*: () → *T*): *T*
40.7.43 *saturatingSignedShift*(*j*: IndexInt): *T*

[Description to be supplied.]

40.7.44 *unsignedShift*(*j*: IndexInt): *T*
40.7.45 *unsignedShift*(*j*: IndexInt, *overflowAction*: () → *T*): *T*
40.7.46 *saturatingUnsignedShift*(*j*: IndexInt): *T*

[Description to be supplied.]

40.7.47 *bitRotate*(*j*: IndexInt): *T*

[Description to be supplied.]

40.7.48 *countOneBits*(): IndexInt

[Description to be supplied.]

40.7.49 *countLeadingZeroBits*(): IndexInt
40.7.50 *countTrailingZeroBits*(): IndexInt

[Description to be supplied.]

40.7.51 *leftmostOneBit*(): *T*
40.7.52 *rightmostOneBit*(): *T*

[Description to be supplied.]

40.7.53 *bitReverse*(): *T*

[Description to be supplied.]

40.7.54 *signedIndex*(): IndexInt `throws` { IntegerOverflowException }
40.7.55 *unsignedIndex*(): IndexInt `throws` { IntegerOverflowException }

[Description to be supplied.]

40.7.56 *gatherBits*(*mask*: *T*): *T*
40.7.57 *spreadBits*(*mask*: *T*): *T*

[Description to be supplied.]

40.7.58 *disentangleBits*(*mask*: *T*): *T*
40.7.59 *intersperseBits*(*mask*: *T*): *T*

[Description to be supplied.]

40.8 The Trait Fortress.Core.BasicBinaryWordOperations

```
trait BasicBinaryWordOperations⟦T extends BasicBinaryWordOperations⟦T, b⟧, nat b⟧
      extends BasicBinaryOperations⟦T⟧ where { b ≤ maxBinaryWordBitLog }
   multiplyLow(other: T): T where { b ≤ maxMultiplyBitLog }
   multiplyLow(other: T, overflowAction: () → T): T where { b ≤ maxMultiplyBitLog }
   saturatedMultiplyLow(other: T): T where { b ≤ maxMultiplyBitLog }
   multiplyHigh(other: T): T where { b ≤ maxMultiplyBitLog }
   multiplyDouble(other: T): (T, T) where { b ≤ maxMultiplyBitLog }
   signedDivide(other: T, overflowAction: () → T, zeroDivideAction: () → T): T
      where { b ≤ maxDivideBitLog }
   unsignedDivide(other: T, zeroDivideAction: () → T): T where { b ≤ maxDivideBitLog }
   signedDivRem(other: T, overflowAction: () → T, zeroDivideAction: () → T): (T, T)
      where { b ≤ maxDivideBitLog }
   unsignedDivRem(other: T, zeroDivideAction: () → T): (T, T) where { b ≤ maxDivideBitLog }
   signedRemainder(other: T, zeroDivideAction: () → T): T where { b ≤ maxDivideBitLog }
   unsignedModulo(other: T, zeroDivideAction: () → T): T where { b ≤ maxDivideBitLog }
   bitSwap(j: IndexInt): T
   getter littleEndian(): BinaryEndianWord⟦b, false, false⟧
   getter bigEndian(): BinaryEndianWord⟦b, true, true⟧
end
```

40.8.1 *multiplyLow*(*other*: *T*): *T* `where` { $b \leq$ *maxMultiplyBitLog* }
40.8.2 *multiplyLow*(*other*: *T*, *overflowAction*: () → *T*): *T* `where` { $b \leq$ *maxMultiplyBitLog* }
40.8.3 *saturatedMultiplyLow*(*other*: *T*): *T* `where` { $b \leq$ *maxMultiplyBitLog* }
40.8.4 *multiplyHigh*(*other*: *T*): *T* `where` { $b \leq$ *maxMultiplyBitLog* }
40.8.5 *multiplyDouble*(*other*: *T*): (*T*, *T*) `where` { $b \leq$ *maxMultiplyBitLog* }

[Description to be supplied.]

40.8.6 *signedDivide*(*other*: T, *overflowAction*: () $\to T$, *zeroDivideAction*: () $\to T$): T
`where` { $b \leq$ *maxDivideBitLog* }
40.8.7 *unsignedDivide*(*other*: T, *zeroDivideAction*: () $\to T$): T `where` { $b \leq$ *maxDivideBitLog* }

[Description to be supplied.]

40.8.8 *signedDivRem*(*other*: T, *overflowAction*: () $\to T$, *zeroDivideAction*: () $\to T$): (T, T)
`where` { $b \leq$ *maxDivideBitLog* }
40.8.9 *unsignedDivRem*(*other*: T, *zeroDivideAction*: () $\to T$): (T, T) `where` { $b \leq$ *maxDivideBitLog* }

[Description to be supplied.]

40.8.10 *signedRemainder*(*other*: T, *zeroDivideAction*: () $\to T$): T `where` { $b \leq$ *maxDivideBitLog* }
40.8.11 *unsignedModulo*(*other*: T, *zeroDivideAction*: () $\to T$): T `where` { $b \leq$ *maxDivideBitLog* }

[Description to be supplied.]

40.8.12 *bitSwap*(j: IndexInt): T

[Description to be supplied.]

40.8.13 `getter` *littleEndian*(): BinaryEndianWord$[\![b, \mathit{false}, \mathit{false}]\!]$
40.8.14 `getter` *bigEndian*(): BinaryEndianWord$[\![b, \mathit{true}, \mathit{true}]\!]$

[Description to be supplied.]

40.9 The Trait Fortress.Core.BinaryLinearEndianSequence

```
trait BinaryLinearEndianSequence⟦nat b, nat n, bool bigEndianSequence⟧
          extends { BasicBinaryOperations⟦BinaryLinearEndianSequence⟦b, n, bigEndianSequence⟧⟧ }
      where { b ≤ maxBinaryWordBitLog }
    coercion ⟦nat b′, bool bigEndianBytes, bool bigEndianBits⟧
        (x: BinaryEndianWord⟦b′, bigEndianBytes, bigEndianBits⟧)
      where { bigEndianBytes = bigEndianSequence, 2^b′ = n · 2^b }
    coercion ⟦nat m, nat radix, nat q, nat k, nat v⟧(x: NaturalNumeral⟦m, radix, v⟧)
      where { radix = 2^q, q · m = n · 2^bk }
    opr [j: IndexInt] : BinaryWord⟦b⟧
      throws { IndexOutOfBoundsException }
    opr ⟦nat k⟧[j: IntegerStatic⟦k⟧] : BinaryWord⟦b⟧ where { k < n }
    opr ⟦nat m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧] :
        BinaryLinearEndianSequence⟦b, m, bigEndianSequence⟧
      throws { IndexOutOfBoundsException } where { m ≤ n }
    opr ⟦int a, nat m, int c⟧[r: StaticRange⟦a, m, c⟧] :
        BinaryLinearEndianSequence⟦b, m, bigEndianSequence⟧
      where { 0 ≤ a < n, 0 ≤ a + m · c < n }
    opr [j: IndexInt] := (v: BinaryWord⟦b⟧):
        BinaryLinearEndianSequence⟦b, n, bigEndianSequence⟧
      throws { IndexOutOfBoundsException }
    opr ⟦nat k⟧[j: IntegerStatic⟦k⟧] := (v: BinaryWord⟦b⟧):
        BinaryLinearEndianSequence⟦b, n, bigEndianSequence⟧ where { k < n }
    opr ⟦nat m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧] :=
        (v: BinaryLinearEndianSequence⟦b, m, bigEndianSequence⟧):
        BinaryLinearEndianSequence⟦b, n, bigEndianSequence⟧
      throws { IndexOutOfBoundsException }
    opr ⟦int a, nat m, int c⟧[r: StaticRange⟦a, m, c⟧] :=
        (v: BinaryLinearEndianSequence⟦b, m, bigEndianSequence⟧):
        BinaryLinearEndianSequence⟦b, n, bigEndianSequence⟧
      where { 0 ≤ a < n, 0 ≤ a + m · c < n }
    update(j: IndexInt, v: BinaryWord⟦b⟧):
        BinaryLinearEndianSequence⟦b, n, bigEndianSequence⟧
      throws { IndexOutOfBoundsException }
    update⟦nat k⟧(j: IntegerStatic⟦k⟧, v: BinaryWord⟦b⟧):
        BinaryLinearEndianSequence⟦b, n, bigEndianSequence⟧ where { k < n }
    update⟦nat m⟧(r: RangeOfStaticSize⟦IndexInt, m⟧,
                  v: BinaryLinearEndianSequence⟦b, m, bigEndianSequence⟧):
        BinaryLinearEndianSequence⟦b, n, bigEndianSequence⟧
      throws { IndexOutOfBoundsException }
    update⟦int a, nat m, int c⟧(r: StaticRange⟦a, m, c⟧,
                                v: BinaryLinearEndianSequence⟦b, m, bigEndianSequence⟧):
        BinaryLinearEndianSequence⟦b, n, bigEndianSequence⟧
      where { 0 ≤ a < n, 0 ≤ a + m · c < n }
    opr ‖⟦nat m⟧(self, other: BinaryLinearEndianSequence⟦b, m, bigEndianSequence⟧):
        BinaryLinearEndianSequence⟦b, n + m, bigEndianSequence⟧
    opr ‖⟦nat m, nat radix, nat q, nat k, nat v⟧(self, other: NaturalNumeral⟦m, radix, v⟧):
        LinearSequence⟦BinaryWord⟦b⟧, n + k⟧ where { radix = 2^q, q · m = k · 2^b }
    opr ‖⟦nat m, nat radix, nat q, nat k, nat v⟧(other: NaturalNumeral⟦m, radix, v⟧, self):
        LinearSequence⟦BinaryWord⟦b⟧, n + k⟧ where { radix = 2^q, q · m = k · 2^b }
```

littleEndian(): BinaryEndianLinearEndianSequence⟦b, *false*, *false*, n, *bigEndianSequence*⟧
bigEndian(): BinaryEndianLinearEndianSequence⟦b, *true*, *true*, n, *bigEndianSequence*⟧
littleEndianBits(): BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *false*, n, *bigEndianSequence*⟧
bigEndianBits(): BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *true*, n, *bigEndianSequence*⟧
littleEndianSequence(): BinaryLinearEndianSequence⟦b, n, *false*⟧
bigEndianSequence(): BinaryLinearEndianSequence⟦b, n, *true*⟧
split⟦`nat` b'⟧(): BinaryLinearEndianSequence⟦$b', n \cdot 2^{b-b'}$, *bigEndianSequence*⟧
`where` { $b' \leq b$ }
`end`

40.9.1 `coercion` ⟦`nat` b', `bool` *bigEndianBytes*, `bool` *bigEndianBits*⟧ (x: BinaryEndianWord⟦b', *bigEndianBytes*, *bigEndianBits*⟧) `where` { *bigEndianBytes* = *bigEndianSequence*, $2^{b'} = n \cdot 2^b$ }

[Description to be supplied.]

40.9.2 `coercion` ⟦`nat` m, `nat` *radix*, `nat` q, `nat` k, `nat` v⟧(x: NaturalNumeral⟦m, *radix*, v⟧) `where` { *radix* = 2^q, $q \cdot m = n \cdot 2^{bk}$ }

[Description to be supplied.]

40.9.3 `opr` [j: IndexInt] : BinaryWord⟦b⟧ `throws` { IndexOutOfBoundsException }
40.9.4 `opr` ⟦`nat` k⟧[j: IntegerStatic⟦k⟧] : BinaryWord⟦b⟧ `where` { $k < n$ }

[Description to be supplied.]

40.9.5 `opr` ⟦`nat` m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧] : BinaryLinearEndianSequence⟦b, m, *bigEndianSequence*⟧ `throws` { IndexOutOfBoundsException } `where` { $m \leq n$ }
40.9.6 `opr` ⟦`int` a, `nat` m, `int` c⟧[r: StaticRange⟦a, m, c⟧] : BinaryLinearEndianSequence⟦b, m, *bigEndianSequence*⟧ `where` { $0 \leq a < n, 0 \leq a + m \cdot c < n$ }

[Description to be supplied.]

40.9.7 `opr` $[j\colon \text{IndexInt}] := (v\colon \text{BinaryWord}[\![b]\!])$:
BinaryLinearEndianSequence$[\![b, n, \textit{bigEndianSequence}]\!]$
`throws` { IndexOutOfBoundsException }

40.9.8 `opr` $[\![\texttt{nat}\ k]\!][j\colon \text{IntegerStatic}[\![k]\!]] := (v\colon \text{BinaryWord}[\![b]\!])$:
BinaryLinearEndianSequence$[\![b, n, \textit{bigEndianSequence}]\!]$ `where` $\{\ k < n\ \}$

[Description to be supplied.]

40.9.9 `opr` $[\![\texttt{nat}\ m]\!][r\colon \text{RangeOfStaticSize}[\![\text{IndexInt}, m]\!]] :=$
$(v\colon \text{BinaryLinearEndianSequence}[\![b, m, \textit{bigEndianSequence}]\!])$:
BinaryLinearEndianSequence$[\![b, n, \textit{bigEndianSequence}]\!]$
`throws` { IndexOutOfBoundsException }

40.9.10 `opr` $[\![\texttt{int}\ a, \texttt{nat}\ m, \texttt{int}\ c]\!][r\colon \text{StaticRange}[\![a, m, c]\!]] :=$
$(v\colon \text{BinaryLinearEndianSequence}[\![b, m, \textit{bigEndianSequence}]\!])$:
BinaryLinearEndianSequence$[\![b, n, \textit{bigEndianSequence}]\!]$
`where` $\{\ 0 \le a < n, 0 \le a + m \cdot c < n\ \}$

[Description to be supplied.]

40.9.11 $\textit{update}(j\colon \text{IndexInt}, v\colon \text{BinaryWord}[\![b]\!])$:
BinaryLinearEndianSequence$[\![b, n, \textit{bigEndianSequence}]\!]$
`throws` { IndexOutOfBoundsException }

40.9.12 $\textit{update}[\![\texttt{nat}\ k]\!](j\colon \text{IntegerStatic}[\![k]\!], v\colon \text{BinaryWord}[\![b]\!])$:
BinaryLinearEndianSequence$[\![b, n, \textit{bigEndianSequence}]\!]$ `where` $\{\ k < n\ \}$

[Description to be supplied.]

40.9.13 $\textit{update}[\![\texttt{nat}\ m]\!](r\colon \text{RangeOfStaticSize}[\![\text{IndexInt}, m]\!],$
$v\colon \text{BinaryLinearEndianSequence}[\![b, m, \textit{bigEndianSequence}]\!])$:
BinaryLinearEndianSequence$[\![b, n, \textit{bigEndianSequence}]\!]$
`throws` { IndexOutOfBoundsException }

40.9.14 $\textit{update}[\![\texttt{int}\ a, \texttt{nat}\ m, \texttt{int}\ c]\!](r\colon \text{StaticRange}[\![a, m, c]\!],$
$v\colon \text{BinaryLinearEndianSequence}[\![b, m, \textit{bigEndianSequence}]\!])$:
BinaryLinearEndianSequence$[\![b, n, \textit{bigEndianSequence}]\!]$
`where` $\{\ 0 \le a < n, 0 \le a + m \cdot c < n\ \}$

[Description to be supplied.]

40.9.15 `opr` $\|\ [\![\texttt{nat}\ m]\!](\texttt{self}, \textit{other}\colon \text{BinaryLinearEndianSequence}[\![b, m, \textit{bigEndianSequence}]\!])$:
BinaryLinearEndianSequence$[\![b, n + m, \textit{bigEndianSequence}]\!]$

[Description to be supplied.]

40.9.16 `opr` $\| [\![\texttt{nat}\ m, \texttt{nat}\ \mathit{radix}, \texttt{nat}\ q, \texttt{nat}\ k, \texttt{nat}\ v]\!](\texttt{self}, \mathit{other}\colon \mathrm{NaturalNumeral}[\![m, \mathit{radix}, v]\!])\colon$ $\mathrm{LinearSequence}[\![T, n + k]\!]$ `where` $\{\ \mathit{radix} = 2^q, q \cdot m = k \cdot 2^b\ \}$

[Description to be supplied.]

40.9.17 `opr` $\| [\![\texttt{nat}\ m, \texttt{nat}\ \mathit{radix}, \texttt{nat}\ q, \texttt{nat}\ k, \texttt{nat}\ v]\!](\mathit{other}\colon \mathrm{NaturalNumeral}[\![m, \mathit{radix}, v]\!], \texttt{self})\colon$ $\mathrm{LinearSequence}[\![T, n + k]\!]$ `where` $\{\ \mathit{radix} = 2^q, q \cdot m = k \cdot 2^b\ \}$

[Description to be supplied.]

40.9.18 $\mathit{littleEndian}()\colon \mathrm{BinaryEndianLinearEndianSequence}[\![b, \mathit{false}, \mathit{false}, n, \mathit{bigEndianSequence}]\!]$
40.9.19 $\mathit{bigEndian}()\colon \mathrm{BinaryEndianLinearEndianSequence}[\![b, \mathit{true}, \mathit{true}, n, \mathit{bigEndianSequence}]\!]$

[Description to be supplied.]

40.9.20 $\mathit{littleEndianBits}()\colon \mathrm{BinaryEndianLinearEndianSequence}[\![b, \mathit{bigEndianBytes}, \mathit{false}, n, \mathit{bigEndianSequence}]\!]$
40.9.21 $\mathit{bigEndianBits}()\colon \mathrm{BinaryEndianLinearEndianSequence}[\![b, \mathit{bigEndianBytes}, \mathit{true}, n, \mathit{bigEndianSequence}]\!]$

[Description to be supplied.]

40.9.22 $\mathit{littleEndianSequence}()\colon \mathrm{BinaryLinearEndianSequence}[\![b, n, \mathit{false}]\!]$
40.9.23 $\mathit{bigEndianSequence}()\colon \mathrm{BinaryLinearEndianSequence}[\![b, n, \mathit{true}]\!]$

[Description to be supplied.]

40.9.24 $\mathit{split}[\![\texttt{nat}\ b']\!]()\colon \mathrm{BinaryLinearEndianSequence}[\![b', n \cdot 2^{b-b'}, \mathit{bigEndianSequence}]\!]$ `where` $\{\ b' \leq b\ \}$

[Description to be supplied.]

40.10 The Trait Fortress.Core.BinaryEndianLinearEndianSequence

```
trait BinaryEndianLinearEndianSequence⟦nat b, bool bigEndianBytes, bool bigEndianBits,
                                       nat n, bool bigEndianSequence⟧
    extends { BasicBinaryOperations⟦
                  BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                                   n, bigEndianSequence⟧⟧ }
    where { b ≤ maxBinaryWordBitLog }
  coercion ⟦nat b′, bool bigEndianBytes, bool bigEndianBits⟧
      (x: BinaryEndianWord⟦b′, bigEndianBytes, bigEndianBits⟧)
    where { bigEndianBytes = bigEndianSequence, 2^b′ = n · 2^b }
  coercion ⟦nat m, nat radix, nat q, nat k, nat v⟧(x: NaturalNumeral⟦m, radix, v⟧)
    where { radix = 2^q, q · m = n · 2^bk }
  opr [j: IndexInt] : BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧
    throws { IndexOutOfBoundsException }
  opr ⟦nat k⟧[j: IntegerStatic⟦k⟧] : BinaryWord⟦b, bigEndianBytes, bigEndianBits⟧
    where { k < n }
  opr ⟦nat m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧] :
      BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                       m, bigEndianSequence⟧
    throws { IndexOutOfBoundsException } where { m ≤ n }
  opr ⟦int a, nat m, int c⟧[r: StaticRange⟦a, m, c⟧] :
      BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                       m, bigEndianSequence⟧
    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
  opr [j: IndexInt] := (v: BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧):
      BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                       n, bigEndianSequence⟧
    throws { IndexOutOfBoundsException }
  opr ⟦nat k⟧[j: IntegerStatic⟦k⟧] :=
      (v: BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧):
      BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                       n, bigEndianSequence⟧
    where { k < n }
  opr ⟦nat m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧] :=
      (v: BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                         m, bigEndianSequence⟧):
      BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                       n, bigEndianSequence⟧
    throws { IndexOutOfBoundsException }
  opr ⟦int a, nat m, int c⟧[r: StaticRange⟦a, m, c⟧] :=
      (v: BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                         k, bigEndianSequence⟧):
      BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                       n, bigEndianSequence⟧
    where { 0 ≤ a < n, 0 ≤ a + m · c < n }
  update(j: IndexInt, v: BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧):
      BinaryEndianLinearEndianSequence⟦b, bigEndianBytes, bigEndianBits,
                                       n, bigEndianSequence⟧
    throws { IndexOutOfBoundsException }
  update⟦nat k⟧(j: IntegerStatic⟦k⟧,
```

v: BinaryEndianWord⟦b, *bigEndianBytes*, *bigEndianBits*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧
`where` { $k < n$ }
update⟦`nat` m⟧(r: RangeOfStaticSize⟦IndexInt, m⟧, v: BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, m, *bigEndianSequence*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧
`throws` { IndexOutOfBoundsException }
update⟦`int` a, `nat` m, `int` c⟧ (r: StaticRange⟦a, m, c⟧, v: BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, m, *bigEndianSequence*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧
`where` { $0 \le a < n, 0 \le a + m \cdot c < n$ }
`opr` ‖⟦`nat` m⟧(`self`, *other*: BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, m, *bigEndianSequence*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, $n + m$, *bigEndianSequence*⟧
`opr` ‖⟦`nat` m, `nat` *radix*, `nat` q, `nat` k, `nat` v⟧(`self`, *other*: NaturalNumeral⟦m, *radix*, v⟧):
LinearSequence⟦$T, n + k$⟧
`where` { $radix = 2^q, q \cdot m = k \cdot 2^b$ }
`opr` ‖⟦`nat` m, `nat` *radix*, `nat` q, `nat` k, `nat` v⟧(*other*: NaturalNumeral⟦m, *radix*, v⟧, `self`):
LinearSequence⟦$T, n + k$⟧
`where` { $radix = 2^q, q \cdot m = k \cdot 2^b$ }
littleEndian(): BinaryEndianLinearEndianSequence⟦b, *false*, *false*, n, *bigEndianSequence*⟧
bigEndian(): BinaryEndianLinearEndianSequence⟦b, *true*, *true*, n, *bigEndianSequence*⟧
littleEndianBits(): BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *false*, n, *bigEndianSequence*⟧
bigEndianBits(): BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *true*, n, *bigEndianSequence*⟧
littleEndianSequence(): BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *false*⟧
bigEndianSequence(): BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *true*⟧
split⟦`nat` b'⟧() :
BinaryEndianLinearEndianSequence⟦b', *bigEndianBytes*, *bigEndianBits*, $n \cdot 2^{b-b'}$, *bigEndianSequence*⟧
`where` { $b' \le b$ }
`end`

40.10.1 `coercion` ⟦`nat` b', `bool` *bigEndianBytes*, `bool` *bigEndianBits*⟧
(x: BinaryEndianWord⟦b', *bigEndianBytes*, *bigEndianBits*⟧)
`where` { $bigEndianBytes = bigEndianSequence, 2^{b'} = n \cdot 2^b$ }

[Description to be supplied.]

40.10.2 `coercion` ⟦`nat` m, `nat` *radix*, `nat` q, `nat` k, `nat` v⟧(x: NaturalNumeral⟦m, *radix*, v⟧)
`where` { $radix = 2^q, q \cdot m = n \cdot 2^{bk}$ }

[Description to be supplied.]

40.10.3 `opr` [j: IndexInt] : BinaryEndianWord⟦b, *bigEndianBytes*, *bigEndianBits*⟧
`throws` { IndexOutOfBoundsException }

40.10.4 `opr` ⟦`nat` k⟧[j: IntegerStatic⟦k⟧] : BinaryWord⟦b, *bigEndianBytes*, *bigEndianBits*⟧
`where` { $k < n$ }

[Description to be supplied.]

40.10.5 `opr` ⟦`nat` m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧] :
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, m, *bigEndianSequence*⟧
`throws` { IndexOutOfBoundsException } `where` { $m \leq n$ }

40.10.6 `opr` ⟦`int` a, `nat` m, `int` c⟧[r: StaticRange⟦a, m, c⟧] :
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, m, *bigEndianSequence*⟧
`where` { $0 \leq a < n, 0 \leq a + m \cdot c < n$ }

[Description to be supplied.]

40.10.7 `opr` [j: IndexInt] := (v: BinaryEndianWord⟦b, *bigEndianBytes*, *bigEndianBits*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧
`throws` { IndexOutOfBoundsException }

40.10.8 `opr` ⟦`nat` k⟧[j: IntegerStatic⟦k⟧] :=
(v: BinaryEndianWord⟦b, *bigEndianBytes*, *bigEndianBits*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧
`where` { $k < n$ }

[Description to be supplied.]

40.10.9 `opr` ⟦`nat` m⟧[r: RangeOfStaticSize⟦IndexInt, m⟧] :=
(v: BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, m, *bigEndianSequence*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧
`throws` { IndexOutOfBoundsException }

40.10.10 `opr` ⟦`int` a, `nat` m, `int` c⟧[r: StaticRange⟦a, m, c⟧] :=
(v: BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, k, *bigEndianSequence*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧
`where` { $0 \le a < n, 0 \le a + m \cdot c < n$ }

[Description to be supplied.]

40.10.11 *update*(j: IndexInt, v: BinaryEndianWord⟦b, *bigEndianBytes*, *bigEndianBits*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧
`throws` { IndexOutOfBoundsException }

40.10.12 *update*⟦`nat` k⟧(j: IntegerStatic⟦k⟧,
v: BinaryEndianWord⟦b, *bigEndianBytes*, *bigEndianBits*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧
`where` { $k < n$ }

[Description to be supplied.]

40.10.13 *update*⟦`nat` m⟧(r: RangeOfStaticSize⟦IndexInt, m⟧,
v: BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, m, *bigEndianSequence*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧
`throws` { IndexOutOfBoundsException }

40.10.14 *update*⟦`int` a, `nat` m, `int` c⟧
(r: StaticRange⟦a, m, c⟧,
v: BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, m, *bigEndianSequence*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*, n, *bigEndianSequence*⟧
`where` { $0 \le a < n, 0 \le a + m \cdot c < n$ }

[Description to be supplied.]

40.10.15 `opr` ‖ ⟦`nat` m⟧(`self`, *other*: BinaryEndianLinearEndianSequence⟦
b, *bigEndianBytes*, *bigEndianBits*,
m, *bigEndianSequence*⟧):
BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*,
$n + m$, *bigEndianSequence*⟧

[Description to be supplied.]

40.10.16 `opr` ‖ ⟦`nat` m, `nat` *radix*, `nat` q, `nat` k, `nat` v⟧(`self`, *other*: NaturalNumeral⟦m, *radix*, v⟧):
LinearSequence⟦$T, n + k$⟧
`where` { $radix = 2^q, q \cdot m = k \cdot 2^b$ }

[Description to be supplied.]

40.10.17 `opr` ‖ ⟦`nat` m, `nat` *radix*, `nat` q, `nat` k, `nat` v⟧(*other*: NaturalNumeral⟦m, *radix*, v⟧, `self`):
LinearSequence⟦$T, n + k$⟧
`where` { $radix = 2^q, q \cdot m = k \cdot 2^b$ }

[Description to be supplied.]

40.10.18 *littleEndian*(): BinaryEndianLinearEndianSequence⟦b, *false*, *false*, n, *bigEndianSequence*⟧
40.10.19 *bigEndian*(): BinaryEndianLinearEndianSequence⟦b, *true*, *true*, n, *bigEndianSequence*⟧

[Description to be supplied.]

40.10.20 *littleEndianBits*(): BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *false*,
n, *bigEndianSequence*⟧
40.10.21 *bigEndianBits*(): BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *true*,
n, *bigEndianSequence*⟧

[Description to be supplied.]

40.10.22 *littleEndianSequence*(): BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*,
n, *false*⟧
40.10.23 *bigEndianSequence*(): BinaryEndianLinearEndianSequence⟦b, *bigEndianBytes*, *bigEndianBits*,
n, *true*⟧

[Description to be supplied.]

40.10.24 *split*⟦nat b'⟧() :
BinaryEndianLinearEndianSequence⟦b', *bigEndianBytes*, *bigEndianBits*, $n \cdot 2^{b-b'}$, *bigEndianSequence*⟧
where { $b' \leq b$ }

[Description to be supplied.]

40.11 The Trait Fortress.Core.BinaryHeapEndianSequence

```
trait BinaryHeapEndianSequence⟦nat b, bool bigEndianSequence⟧
      extends { BinaryHeapSequence⟦b⟧,
                BasicBinaryHeapSubsequenceOperations⟦
                   BinaryHeapEndianSequence⟦b, bigEndianSequence⟧,
                   bigEndianSequence⟧ }
end
```

40.12 The Trait Fortress.Core.BinaryEndianHeapEndianSequence

```
trait BinaryEndianHeapEndianSequence⟦nat b, bool bigEndianBytes, bool bigEndianBits,
                                     bool bigEndianSequence⟧
      extends { HeapSequence⟦BinaryEndianWord⟦b, bigEndianBytes, bigEndianBits⟧⟧,
                BasicBinaryHeapSubsequenceOperations⟦
                   BinaryHeapEndianSequence⟦b, bigEndianSequence⟧,
                   bigEndianSequence⟧ }
end
```

40.13 The Trait Fortress.Core.BasicBinaryHeapSubsequenceOperations

```
trait BasicBinaryHeapSubsequenceOperations⟦
      T extends BasicBinaryHeapSubsequenceOperations⟦T, bigEndianSequence⟧,
      bool bigEndianSequence⟧
  copy(selfStart: IndexInt, source: T, sourceStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBoundsException }
  wrappingAdd(selfStart: IndexInt, source: T, sourceStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBoundsException }
  add(selfStart: IndexInt, source: T, sourceStart: IndexInt,
      length: IndexInt, carryIn: Bit = 0): Bit
    throws { IndexOutOfBoundsException }
  signedAdd(selfStart: IndexInt, source: T, sourceStart: IndexInt,
            length: IndexInt, overflowAction: () → ()): ()
    throws { IndexOutOfBoundsException }
  unsignedAdd(selfStart: IndexInt, source: T, sourceStart: IndexInt,
              length: IndexInt, overflowAction: () → ()): ()
    throws { IndexOutOfBoundsException }
  saturatingSignedAdd(selfStart: IndexInt, source: T, sourceStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBoundsException }
  saturatingUnsignedAdd(selfStart: IndexInt, source: T, sourceStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBoundsException }
  wrappingSubtract(selfStart: IndexInt, source: T, sourceStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBoundsException }
  subtract(selfStart: IndexInt, source: T, sourceStart: IndexInt,
           length: IndexInt, carryIn: Bit = 1): Bit
    throws { IndexOutOfBoundsException }
  signedSubtract(selfStart: IndexInt, source: T, sourceStart: IndexInt,
                 length: IndexInt, overflowAction: () → ()): ()
    throws { IndexOutOfBoundsException }
  unsignedSubtract(selfStart: IndexInt, source: T, sourceStart: IndexInt,
                   length: IndexInt, overflowAction: () → ()): ()
    throws { IndexOutOfBoundsException }
  saturatingSignedSubtract(selfStart: IndexInt, source: T, sourceStart: IndexInt,
                           length: IndexInt): ()
    throws { IndexOutOfBoundsException }
  saturatingUnsignedSubtract(selfStart: IndexInt, source: T, sourceStart: IndexInt,
                             length: IndexInt): ()
    throws { IndexOutOfBoundsException }
  wrappingNegate(selfStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBoundsException }
  negate(selfStart: IndexInt, length: IndexInt, carryIn: Bit = 1): Bit
    throws { IndexOutOfBoundsException }
  signedNegate(selfStart: IndexInt, length: IndexInt, overflowAction: () → ()): ()
    throws { IndexOutOfBoundsException }
  unsignedNegate(selfStart: IndexInt, length: IndexInt, overflowAction: () → ()): ()
    throws { IndexOutOfBoundsException }
  saturatingSignedNegate(selfStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBoundsException }
  bitNot(selfStart: IndexInt, length: IndexInt): ()
    throws { IndexOutOfBoundsException }
```

bitAnd(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
bitOr(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
bitXor(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
bitXorNot(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
bitNand(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
bitNor(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
bitAndNot(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
bitOrNot(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
signedMax(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
signedMin(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
unsignedMax(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
unsignedMin(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
equal(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
unequal(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
signedLT(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
signedLE(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
signedGE(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
signedGT(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
unsignedLT(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
unsignedLE(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
unsignedGE(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
unsignedGT(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
signedShift(*selfStart*: IndexInt, *length*: IndexInt, j: IndexInt): ()
`throws` { IndexOutOfBoundsException }
signedShift(*selfStart*: IndexInt, *length*: IndexInt, j: IndexInt, *overflowAction*: () → ()): ()
`throws` { IndexOutOfBoundsException }
saturatingSignedShift(*selfStart*: IndexInt, *length*: IndexInt, j: IndexInt): ()
`throws` { IndexOutOfBoundsException }
unsignedShift(*selfStart*: IndexInt, *length*: IndexInt, j: IndexInt): ()
`throws` { IndexOutOfBoundsException }

unsignedShift(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt, *overflowAction*: () → ()): ()
`throws` { IndexOutOfBoundsException }
saturatingUnsignedShift(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
bitRotate(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
countOneBits(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
`throws` { IndexOutOfBoundsException }
countLeadingZeroBits(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
`throws` { IndexOutOfBoundsException }
countTrailingZeroBits(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
`throws` { IndexOutOfBoundsException }
leftmostOneBit(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
rightmostOneBit(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
bitReverse(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
gatherBits(*selfStart*: IndexInt, *mask*: *T*, *maskStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
spreadBits(*selfStart*: IndexInt, *mask*: *T*, *maskStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
clearAllBits(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
setAllBits(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
signedIndex(): IndexInt
`throws` { IntegerOverflowException }
signedIndex(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
`throws` { IndexOutOfBoundsException, IntegerOverflowException }
unsignedIndex(): IndexInt
`throws` { IntegerOverflowException }
unsignedIndex(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
`throws` { IndexOutOfBoundsException, IntegerOverflowException }
`end`

40.13.1 *copy*(*selfStart*: IndexInt, *source*: *T*, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.2 *wrappingAdd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.3 *add*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *carryIn*: Bit = 0): Bit
`throws` { IndexOutOfBoundsException }

40.13.4 *signedAdd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *overflowAction*: () → ()): ()
`throws` { IndexOutOfBoundsException }

40.13.5 *unsignedAdd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *overflowAction*: () → ()): ()
`throws` { IndexOutOfBoundsException }

40.13.6 *saturatingSignedAdd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.7 *saturatingUnsignedAdd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.8 *wrappingSubtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.9 *subtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *carryIn*: Bit = 1): Bit
`throws` { IndexOutOfBoundsException }

40.13.10 *signedSubtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *overflowAction*: () → ()): ()
`throws` { IndexOutOfBoundsException }

40.13.11 *unsignedSubtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt, *overflowAction*: () → ()): ()
`throws` { IndexOutOfBoundsException }

40.13.12 *saturatingSignedSubtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.13 *saturatingUnsignedSubtract*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt,
length: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.14 *wrappingNegate*(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.15 *negate*(*selfStart*: IndexInt, *length*: IndexInt, *carryIn*: Bit = 1): Bit
`throws` { IndexOutOfBoundsException }

40.13.16 *signedNegate*(*selfStart*: IndexInt, *length*: IndexInt, *overflowAction*: () → ()): ()
`throws` { IndexOutOfBoundsException }

40.13.17 *unsignedNegate*(*selfStart*: IndexInt, *length*: IndexInt, *overflowAction*: () → ()): ()
`throws` { IndexOutOfBoundsException }

40.13.18 *saturatingSignedNegate*(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.19 *bitNot*(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.20 *bitAnd*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.21 *bitOr*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.22 *bitXor*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.23 *bitXorNot*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.24 *bitNand*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.25 *bitNor*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.26 *bitAndNot*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.27 *bitOrNot*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.28 *signedMax*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

40.13.29 *signedMin*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.30 *unsignedMax*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
40.13.31 *unsignedMin*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.32 *equal*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
40.13.33 *unequal*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.34 *signedLT*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
40.13.35 *signedLE*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
40.13.36 *signedGE*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
40.13.37 *signedGT*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.38 *unsignedLT*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
40.13.39 *unsignedLE*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
40.13.40 *unsignedGE*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }
40.13.41 *unsignedGT*(*selfStart*: IndexInt, *source*: T, *sourceStart*: IndexInt, *length*: IndexInt): Boolean
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.42 *signedShift*(*selfStart*: IndexInt, *length*: IndexInt, j: IndexInt): ()
`throws` { IndexOutOfBoundsException }
40.13.43 *signedShift*(*selfStart*: IndexInt, *length*: IndexInt, j: IndexInt, *overflowAction*: () → ()): ()
`throws` { IndexOutOfBoundsException }
40.13.44 *saturatingSignedShift*(*selfStart*: IndexInt, *length*: IndexInt, j: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.45 *unsignedShift*(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
40.13.46 *unsignedShift*(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt, *overflowAction*: () → ()): ()
`throws` { IndexOutOfBoundsException }
40.13.47 *saturatingUnsignedShift*(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.48 *bitRotate*(*selfStart*: IndexInt, *length*: IndexInt, *j*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.49 *countOneBits*(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.50 *countLeadingZeroBits*(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
`throws` { IndexOutOfBoundsException }
40.13.51 *countTrailingZeroBits*(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.52 *leftmostOneBit*(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
40.13.53 *rightmostOneBit*(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.54 *bitReverse*(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.55 *gatherBits*(*selfStart*: IndexInt, *mask*: T, *maskStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
40.13.56 *spreadBits*(*selfStart*: IndexInt, *mask*: T, *maskStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.57 *clearAllBits*(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }
40.13.58 *setAllBits*(*selfStart*: IndexInt, *length*: IndexInt): ()
`throws` { IndexOutOfBoundsException }

[Description to be supplied.]

40.13.59 *signedIndex*(): IndexInt
`throws` { IntegerOverflowException }
40.13.60 *signedIndex*(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
`throws` { IndexOutOfBoundsException, IntegerOverflowException }
40.13.61 *unsignedIndex*(): IndexInt
`throws` { IntegerOverflowException }
40.13.62 *unsignedIndex*(*selfStart*: IndexInt, *length*: IndexInt): IndexInt
`throws` { IndexOutOfBoundsException, IntegerOverflowException }

[Description to be supplied.]

Part VI

Appendices

Appendix A

Fortress Calculi

A.1 Basic Core Fortress

In this section, we define a basic core calculus for Fortress. We call this calculus *Basic Core Fortress*. Following the precedent set by prior core calculi such as Featherweight Generic Java [12], we have abided by the restriction that all valid Basic Core Fortress programs are valid Fortress programs.

A.1.1 Syntax

A syntax for Basic Core Fortress is provided in Figure A.1. We use the following notational conventions:

- For brevity, we omit separators such as , and ; from Basic Core Fortress.
- $\overrightarrow{\tau}$ is a shorthand for a (possibly empty) sequence $\tau_1, \cdots, \tau_n$.
- Similarly, we abbreviate a sequence of relations α_1 `extends` $N_1, \cdots, \alpha_n$ `extends` N_n to $\overrightarrow{\alpha \texttt{ extends } N}$
- We use τ_i to denote the ith element of $\overrightarrow{\tau}$.
- For simplicity, we assume that every name (type variables, field names, and parameters) is different and every trait/object declaration declares unique name.
- We prohibit cycles in type hierarchies.

The syntax of Basic Core Fortress allows only a small subset of the Fortress language to be formalized. Basic Core Fortress includes trait and object definitions, method and field invocations, and `self` expressions. The types of Basic Core Fortress include type variables, instantiated traits, instantiated objects, and the distinguished trait Object. Note that we syntactically prohibit extending objects. Among other features, Basic Core Fortress does *not* include top-level variable and function definitions, overloading, `excludes` clauses, `bounds` clauses, `where` clauses, object expressions, and function expressions. Basic Core Fortress will be extended to formalize a larger set of Fortress programs in the future.

A.1.2 Dynamic Semantics

A dynamic semantics for Basic Core Fortress is provided in Figure A.2. This semantics has been mechanized via the PLT Redex tool [15]. It therefore follows the style of explicit evaluation contexts and redexes. The Basic Core

α, β			type variables
f			method name
x			field name
T			trait name
O			object name
τ, τ', τ''	::=	α	type
	|	σ	
σ	::=	N	type that is not a type variable
	|	$O[\![\overrightarrow{\tau}]\!]$	
N, M, L	::=	$T[\![\overrightarrow{\tau}]\!]$	type that can be a type bound
	|	`Object`	
e	::=	x	expression
	|	`self`	
	|	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
	|	$e.x$	
	|	$e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
fd	::=	$f[\![\overrightarrow{\alpha \texttt{ extends } N}]\!](\overrightarrow{x{:}\tau}){:}\tau = e$	method definition
td	::=	`trait` $T[\![\overrightarrow{\alpha \texttt{ extends } N}]\!]$ `extends` $\{\overrightarrow{N}\}$ $\overrightarrow{fd}$ `end`	trait definition
od	::=	`object` $O[\![\overrightarrow{\alpha \texttt{ extends } N}]\!](\overrightarrow{x{:}\tau})$ `extends` $\{\overrightarrow{N}\}$ $\overrightarrow{fd}$ `end`	object definition
d	::=	td	definition
	|	od	
p	::=	$\overrightarrow{d}\ e$	program

Figure A.1: Syntax of Basic Core Fortress

Fortress dynamic semantics consists of two evaluations rules: one for field access and another for method invocation. For simplicity, we use ' _ ' to denote some parts of the syntax that do not have key roles in a rule. We assume that _ does not expand across definition boundaries unless the entire definition is included in it.

A.1.3 Static Semantics

A static semantics for Basic Core Fortress is provided in Figures A.3, A.4, and A.5. The Basic Core Fortress static semantics is based on the static semantics of Featherweight Generic Java (FGJ) [12]. The major difference is the division of classes into traits and objects. Both trait and object definitions include method definitions but only object definitions include field definitions. With traits, Basic Core Fortress supports multiple inheritance. However, due to the similarity of traits and objects, many of the rules in the Basic Core Fortress dynamic and static semantics combine the two cases. Note that Basic Core Fortress allows parametric polymorphism, subtype polymorphism, and overriding in much the same way that FGJ does.

We proved the type soundness of Basic Core Fortress using the standard technique of proving a progress theorem and a subject reduction theorem.

Values, evaluation contexts, redexes, and trait and object names

v	::=	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})$	value
E	::=	$\Box$	evaluation context
	|	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e})$	
	|	$E.x$	
	|	$E.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
	|	$e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e})$	
R	::=	$v.x$	redex
	|	$v.f[\![\overrightarrow{\tau}]\!](\overrightarrow{v})$	
C	::=	T	trait name
	|	O	object name

Evaluation rules: $\boxed{p \vdash E[R] \longrightarrow E[e]}$

[R-FIELD]
$$\frac{\texttt{object}\ O\ \_\ (\overrightarrow{x\texttt{:}\ \_})\ \_\ \texttt{end} \in p}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).x_i] \longrightarrow E[v_i]}$$

[R-METHOD]
$$\frac{\texttt{object}\ O\ \_\ (\overrightarrow{x\texttt{:}\ \_})\ \_\ \texttt{end} \in p \qquad mbody_p(f[\![\overrightarrow{\tau'}]\!], O[\![\overrightarrow{\tau}]\!]) = \{(\overrightarrow{x'}) \to e\}}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).f[\![\overrightarrow{\tau'}]\!](\overrightarrow{v'})] \longrightarrow E[\overrightarrow{v}/\overrightarrow{x}][O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x'}]e]}$$

Method body lookup: $\boxed{mbody_p(f[\![\overrightarrow{\tau}]\!], \tau) = \{(\overrightarrow{x}) \to e\}}$

[MB-SELF]
$$\frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f[\![\overrightarrow{\alpha'\ \texttt{extends}\ \_}]\!](\overrightarrow{x'\texttt{:}\ \_})\ \_\ = e \in \{\overrightarrow{fd}\}}{mbody_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!]) = \{[\overrightarrow{\tau'}/\overrightarrow{\alpha'}][\overrightarrow{\tau}/\overrightarrow{\alpha}](\overrightarrow{x'}) \to e\}}$$

[MB-SUPER]
$$\frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f \notin \{\overrightarrow{Fname(fd)}\}}{mbody_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!]) = \bigcup_{N_i \in \{\overrightarrow{N}\}} mbody_p(f[\![\overrightarrow{\tau'}]\!], [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i)}$$

[MB-OBJ]
$$mbody_p(f[\![\overrightarrow{\tau}]\!], \texttt{Object}) = \emptyset$$

Function/method name lookup: $\boxed{Fname(fd) = f}$

$Fname(f[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x\texttt{:}\tau})\texttt{:}\tau = e) = f$

Figure A.2: Dynamic Semantics of Basic Core Fortress

Environments

$$\begin{array}{lcll} \Delta & ::= & \overrightarrow{\alpha <: N} & \text{bound environment} \\ \Gamma & ::= & \overrightarrow{x : \tau} & \text{type environment} \end{array}$$

Program typing: $\boxed{\vdash p : \tau}$

$$\text{[T-PROGRAM]} \quad \frac{p = \overrightarrow{d}\ e \qquad p \vdash \overrightarrow{d}\ \text{ok} \qquad p; \emptyset; \emptyset \vdash e : \tau}{\vdash p : \tau}$$

Definition typing: $\boxed{p \vdash d\ \text{ok}}$

$$\text{[T-TRAITDEF]} \quad \frac{\Delta = \overrightarrow{\alpha <: N} \qquad p; \Delta \vdash \overrightarrow{N}\ \text{ok} \qquad p; \Delta \vdash \overrightarrow{M}\ \text{ok} \qquad p; \Delta; \texttt{self} : T[\![\overrightarrow{\alpha}]\!]; T \vdash \overrightarrow{fd}\ \text{ok} \quad p \vdash \mathit{oneOwner}(T)}{p \vdash \texttt{trait}\ T[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \texttt{extends}\ \{\overrightarrow{M}\}\ \overrightarrow{fd}\ \texttt{end}\ \text{ok}}$$

$$\text{[T-OBJECTDEF]} \quad \frac{\Delta = \overrightarrow{\alpha <: N} \qquad p; \Delta \vdash \overrightarrow{N}\ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau}\ \text{ok} \qquad p; \Delta \vdash \overrightarrow{M}\ \text{ok} \quad p; \Delta; \texttt{self} : O[\![\overrightarrow{\alpha}]\!]\ \overrightarrow{x : \tau}; O \vdash \overrightarrow{fd}\ \text{ok} \qquad p \vdash \mathit{oneOwner}(O)}{p \vdash \texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x:\tau})\ \texttt{extends}\ \{\overrightarrow{M}\}\ \overrightarrow{fd}\ \texttt{end}\ \text{ok}}$$

Method typing: $\boxed{p; \Delta; \Gamma; C \vdash fd\ \text{ok}}$

$$\text{[T-METHODDEF]} \quad \frac{\_\ C\ \_\ \texttt{extends}\ \{\overrightarrow{M}\}\ \_ \in p \qquad p; \Delta \vdash \mathit{override}(f, \{\overrightarrow{M}\}, [\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \overrightarrow{\tau} \to \tau_0) \quad \Delta' = \Delta\ \overrightarrow{\alpha <: N} \qquad p; \Delta' \vdash \overrightarrow{N}\ \text{ok} \qquad p; \Delta' \vdash \overrightarrow{\tau}\ \text{ok} \qquad p; \Delta' \vdash \tau_0\ \text{ok} \quad p; \Delta'; \Gamma\ \overrightarrow{x : \tau} \vdash e : \tau' \qquad p; \Delta' \vdash \tau' <: \tau_0}{p; \Delta; \Gamma; C \vdash f[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x:\tau}):\tau_0 = e\ \text{ok}}$$

Method overriding: $\boxed{p; \Delta \vdash \mathit{override}(f, \{\overrightarrow{N}\}, [\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \overrightarrow{\tau} \to \tau)}$

$$\text{[OVERRIDE]} \quad \frac{\bigcup_{L_i \in \{\overrightarrow{L}\}} \mathit{mtype}_p(f, L_i) = \{[\![\overrightarrow{\beta\ \texttt{extends}\ M}]\!]\ \overrightarrow{\tau'} \to \tau_0'\} \quad \overrightarrow{N} = [\overrightarrow{\alpha}/\overrightarrow{\beta}]\overrightarrow{M} \qquad \overrightarrow{\tau} = [\overrightarrow{\alpha}/\overrightarrow{\beta}]\overrightarrow{\tau'} \qquad p; \overrightarrow{\alpha <: N} \vdash \tau_0 <: [\overrightarrow{\alpha}/\overrightarrow{\beta}]\tau_0'}{p; \Delta \vdash \mathit{override}(f, \{\overrightarrow{L}\}, [\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \overrightarrow{\tau} \to \tau_0)}$$

Method type lookup: $\boxed{\mathit{mtype}_p(f, \tau) = \{[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \overrightarrow{\tau} \to \tau\}}$

$$\text{[MT-SELF]} \quad \frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f[\![\overrightarrow{\beta\ \texttt{extends}\ M}]\!](\overrightarrow{\_:\tau'}):\tau_0' = e \in \{\overrightarrow{fd}\}}{\mathit{mtype}_p(f, C[\![\overrightarrow{\tau}]\!]) = \{[\overrightarrow{\tau}/\overrightarrow{\alpha}][\![\overrightarrow{\beta\ \texttt{extends}\ M}]\!]\ \overrightarrow{\tau'} \to \tau_0'\}}$$

$$\text{[MT-SUPER]} \quad \frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f \notin \{\overrightarrow{\mathit{Fname}(fd)}\}}{\mathit{mtype}_p(f, C[\![\overrightarrow{\tau}]\!]) = \bigcup_{N_i \in \{\overrightarrow{N}\}} \mathit{mtype}_p(f, [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i)}$$

$$\text{[MT-OBJ]} \quad \mathit{mtype}_p(f, \texttt{Object}) = \emptyset$$

Figure A.3: Static Semantics of Basic Core Fortress (I)

Expression typing: $\boxed{p; \Delta; \Gamma \vdash e : \tau}$

[T-VAR] $$p; \Delta; \Gamma \vdash x : \Gamma(x)$$

[T-SELF] $$p; \Delta; \Gamma \vdash \texttt{self} : \Gamma(\texttt{self})$$

[T-OBJECT] $$\frac{\texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!](\overrightarrow{\_ : \tau'})\ \_\ \texttt{end} \in p \qquad p; \Delta \vdash O[\![\overrightarrow{\tau}]\!]\ \text{ok} \qquad p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \qquad p; \Delta \vdash \overrightarrow{\tau''} <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{\tau'}}{p; \Delta; \Gamma \vdash O[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) : O[\![\overrightarrow{\tau}]\!]}$$

[T-FIELD] $$\frac{p; \Delta; \Gamma \vdash e_0 : \tau_0 \qquad bound_\Delta(\tau_0) = O[\![\overrightarrow{\tau'}]\!] \qquad \texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!](\overrightarrow{x : \tau})\ \_\ \texttt{end} \in p}{p; \Delta; \Gamma \vdash e_0\texttt{.}x_i : [\overrightarrow{\tau'}/\overrightarrow{\alpha}]\tau_i}$$

[T-METHOD] $$\frac{p; \Delta; \Gamma \vdash e_0 : \tau_0 \qquad mtype_p(f, bound_\Delta(\tau_0)) = \{[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \overrightarrow{\tau'} \to \tau_0'\} \qquad p; \Delta \vdash \overrightarrow{\tau}\ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{N} \qquad p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \qquad p; \Delta \vdash \overrightarrow{\tau''} <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{\tau'}}{p; \Delta; \Gamma \vdash e_0.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) : [\overrightarrow{\tau}/\overrightarrow{\alpha}]\tau_0'}$$

Subtyping: $\boxed{p; \Delta \vdash \tau <: \tau}$

[S-OBJ] $$p; \Delta \vdash \tau <: \texttt{Object}$$

[S-REFL] $$p; \Delta \vdash \tau <: \tau$$

[S-TRANS] $$\frac{p; \Delta \vdash \tau_1 <: \tau_2 \qquad p; \Delta \vdash \tau_2 <: \tau_3}{p; \Delta \vdash \tau_1 <: \tau_3}$$

[S-VAR] $$p; \Delta \vdash \alpha <: \Delta(\alpha)$$

[S-TAPP] $$\frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_ \in p}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!] <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i}$$

Well-formed types: $\boxed{p; \Delta \vdash \tau\ \text{ok}}$

[W-OBJ] $$p; \Delta \vdash \texttt{Object}\ \text{ok}$$

[W-VAR] $$\frac{\alpha \in \mathit{dom}(\Delta)}{p; \Delta \vdash \alpha\ \text{ok}}$$

[W-TAPP] $$\frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \_ \in p \qquad p; \Delta \vdash \overrightarrow{\tau}\ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{N}}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!]\ \text{ok}}$$

Figure A.4: Static Semantics of Basic Core Fortress (II)

Bound of type: $\boxed{bound_\Delta(\tau) = \sigma}$

$$\begin{array}{lcl} bound_\Delta(\alpha) & = & \Delta(\alpha) \\ bound_\Delta(\sigma) & = & \sigma \end{array}$$

One owner for all the visible methods: $\boxed{p \vdash oneOwner(C)}$

$$[\textsc{OneOwner}] \quad \frac{\forall f \in visible_p(C)\ .\ f \text{ only occurs once in } visible_p(C)}{p \vdash oneOwner(C)}$$

Auxiliary functions for methods: $\boxed{defined_p \text{ / } inherited_p \text{ / } visible_p(C) = \{\overrightarrow{f}\}}$

$$defined_p(C) = \{\overrightarrow{Fname(fd)}\} \qquad \text{where } \_\ C\ \_\ \overrightarrow{fd}\ \_\ \in p$$

$$inherited_p(C) = \biguplus_{N_i \in \{\overrightarrow{N}\}} \{f_i \mid f_i \in visible_p(N_i), f_i \notin defined_p(C)\} \qquad \text{where } \_\ C\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_\ \in p$$

$$visible_p(C) = defined_p(C) \uplus inherited_p(C)$$

Figure A.5: Static Semantics of Basic Core Fortress (III)

A.2 Core Fortress with Where Clauses

In this section, we define a Fortress core calculus with `where` clauses. We call this calculus *Core Fortress with Where Clauses*. Core Fortress with Where Clauses is an extension of Basic Core Fortress with `where` clauses.

A.2.1 Syntax

The syntax for Core Fortress with Where Clauses is provided in Figure A.6. For simplicity, we use the following notational conventions:

- We abbreviate a sequence of relations α_1 `extends` $\{\overrightarrow{K_1}\} \cdots \alpha_n$ `extends` $\{\overrightarrow{K_n}\}$ to $\overrightarrow{\alpha \texttt{ extends } \{\overrightarrow{K}\}}$ and $\tau_1 <: H_{11}\ \tau_1 <: H_{12} \cdots \tau_1 <: H_{1l}\ \tau_2 <: H_{21} \cdots \tau_n <: H_{nm}$ to $\overrightarrow{\overrightarrow{\tau} <: \overrightarrow{H}}$.
- Substitutions of x with v and α with τ are denoted as $[v/x]$ and $[\tau/\alpha]$, respectively. A sequence of substitutions represents the composition of those substitutions where the right-most substitution is applied first. For example, $S_n \cdots S_1\tau$ represents $S_n(\cdots S_2(S_1\tau)\cdots)$.

Most of the syntax is a straightforward extension of Basic Core Fortress in Section A.1.1. An object or trait definition may include `where` clauses. Every method invocation is annotated with three kinds of static types by type inference: the static types of the receiver, the arguments, and the result. These type annotations appear in Core Fortress with Where Clauses in a form of `as` τ. If the annotated types are not enough (to find "witnesses" for the where-clauses variables), type checking rejects the program and requires more type information from the programmer.

A.2.2 Dynamic Semantics

A dynamic semantics for Core Fortress with Where Clauses is provided in Figures A.7 and A.8.

α, β			type variables	
f			method name	
x			field name	
T			trait name	
O			object name	
C			trait or object name	
τ, τ', τ''	$::=$	α	type	
	$\mid$	σ		
σ	$::=$	N	type that is not a type variable	
	$\mid$	$O[\![\overrightarrow{\tau}]\!]$		
N, M, L	$::=$	$T[\![\overrightarrow{\tau}]\!]$	super trait	
	$\mid$	`Object`		
K, H, J	$::=$	α	bound on a type variable	
	$\mid$	N		
e	$::=$	x	expression	
	$\mid$	`self`		
	$\mid$	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$		
	$\mid$	$e.x$		
	$\mid$	e `as` $C[\![\overrightarrow{\tau}]\!].f[\![\overrightarrow{\tau}]\!](\overrightarrow{e \texttt{ as } \tau})$ `as` τ		
	$\mid$	`typecase` x $= e$ `in` $\overrightarrow{\tau \Rightarrow e}$ `else` e `end`		
fd	$::=$	$f[\![\overrightarrow{\alpha \texttt{ extends } \{K\}}]\!](\overrightarrow{x{:}\tau}){:}\tau = e$	method definition	
td	$::=$	`trait` $T[\![\overrightarrow{\alpha \texttt{ extends } \{K\}}]\!]$ `extends` $\{\overrightarrow{N}\}$ `where` $\{\overrightarrow{\alpha \texttt{ extends } \{K\}}\}$ $\overrightarrow{fd}$ `end`	trait definition	
od	$::=$	`object` $O[\![\overrightarrow{\alpha \texttt{ extends } \{K\}}]\!](\overrightarrow{x{:}\tau})$ `extends` $\{\overrightarrow{N}\}$ `where` $\{\overrightarrow{\alpha \texttt{ extends } \{K\}}\}$ $\overrightarrow{fd}$ `end`	object definition	
d	$::=$	td	definition	
	$\mid$	od		
p	$::=$	$\overrightarrow{d}$ e	program	

Figure A.6: Syntax of Core Fortress with Where Clauses

A.2.3 Static Semantics

A static semantics for Core Fortress with Where Clauses is provided in Figures A.9, A.10, A.11, and A.12.

For simplicity, we use the following conventions:

- $FTV(\tau)$ collects all free type variables in τ.
- Similarly, $FTV(e)$ collects all free type variables in all types in e.

We proved the type soundness of Core Fortress with Where Clauses using the standard technique of proving a progress theorem and a subject reduction theorem.

Values, evaluation contexts, and redexes

$$
\begin{array}{rcl}
v & ::= & O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}) \\
E & ::= & \square \\
 & | & O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}\,E\,\overrightarrow{e}) \\
 & | & E.x \\
 & | & E\ \texttt{as}\ \tau.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e\ \texttt{as}\ \tau})\texttt{as}\ \tau \\
 & | & v\ \texttt{as}\ \tau.f[\![\overrightarrow{\tau}]\!](\overrightarrow{v\ \texttt{as}\ \tau}\ E\ \texttt{as}\ \tau\ \overrightarrow{e\ \texttt{as}\ \tau})\texttt{as}\ \tau \\
 & | & \texttt{typecase}\ x\ = E\ \texttt{in}\ \overrightarrow{\tau \Rightarrow e}\ \texttt{else}\ e\ \texttt{end} \\
R & ::= & v.x \\
 & | & v\ \texttt{as}\ \tau.f[\![\overrightarrow{\tau}]\!](\overrightarrow{v\ \texttt{as}\ \tau})\texttt{as}\ \tau \\
 & | & \texttt{typecase}\ x\ = v\ \texttt{in}\ \overrightarrow{\tau \Rightarrow e}\ \texttt{else}\ e\ \texttt{end}
\end{array}
$$

Evaluation rules: $\boxed{p \vdash E[R] \longrightarrow E[e]}$

[R-FIELD]
$$\frac{\texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!](\overrightarrow{x\colon\_})\ \_\ \texttt{end} \in p}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).x_i] \longrightarrow E[v_i]}$$

[R-METHOD]
$$\frac{\begin{array}{c} mtype_p(f, C[\![\overrightarrow{\tau^c}]\!], \emptyset) = \{(\overrightarrow{[\![\alpha'\ \texttt{extends}\ \_]\!]}\ \overrightarrow{\tau''} \to \tau'_0, \_)\} \quad S([\overrightarrow{\tau'}/\overrightarrow{\alpha'}]\overrightarrow{\tau''}) = \overrightarrow{\tau^a} \quad S([\overrightarrow{\tau'}/\overrightarrow{\alpha'}]\tau'_0) = \tau^r \\ \texttt{object}\ O\ \_\ (\overrightarrow{x\colon\_})\ \_\ \texttt{end} \in p \qquad mbody_p(f[\![\overrightarrow{\tau'}]\!], O[\![\overrightarrow{\tau}]\!], C[\![\overrightarrow{\tau^c}]\!]) = \{(\overrightarrow{x'}) \to e\} \end{array}}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})\texttt{as}\ C[\![\overrightarrow{\tau^c}]\!].f[\![\overrightarrow{\tau'}]\!](\overrightarrow{v'\ \texttt{as}\ \tau^a})\texttt{as}\ \tau^r] \longrightarrow E[[\overrightarrow{v}/\overrightarrow{x}][O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x'}]Se]}$$

[R-TYPECASE]
$$\frac{type(v) = \tau^v \qquad |\overrightarrow{\tau}| = n \qquad \neg(p; \emptyset \vdash \tau^v\ <:\ \tau_i) \qquad 1 \le i < n \qquad p; \emptyset \vdash \tau^v\ <:\ \tau_n}{p \vdash E[\texttt{typecase}\ x\ = v\ \texttt{in}\ \overrightarrow{\tau \Rightarrow e}\ \overrightarrow{\tau' \Rightarrow e'}\ \texttt{else}\ e\ \texttt{end}] \longrightarrow E[[v/x]e_n]}$$

[R-TYPECASEELSE]
$$\frac{type(v) = \tau^v \qquad \neg(p; \emptyset \vdash \tau^v\ <:\ \tau_i) \qquad 1 \le i \le |\overrightarrow{\tau}|}{p \vdash E[\texttt{typecase}\ x\ = v\ \texttt{in}\ \overrightarrow{\tau \Rightarrow e}\ \texttt{else}\ e'\ \texttt{end}] \longrightarrow E[[v/x]e']}$$

Method body lookup: $\boxed{mbody_p(f[\![\overrightarrow{\tau}]\!], \tau, \tau) = \{(\overrightarrow{x}) \to e\}}$

[MB-SELF]
$$\frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f[\![\overrightarrow{\alpha'\ \texttt{extends}\ \_}]\!](\overrightarrow{x'\colon\_})\colon\_\ = e \in \{\overrightarrow{fd}\}}{mbody_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!], C[\![\overrightarrow{\tau^c}]\!]) = \{(\overrightarrow{x'}) \to [\overrightarrow{\tau'}/\overrightarrow{\alpha'}][\overrightarrow{\tau^c}/\overrightarrow{\alpha}]e\}}$$

[MB-WITNESS]
$$\frac{\begin{array}{c} \_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f[\![\overrightarrow{\alpha'\ \texttt{extends}\ \_}]\!](\overrightarrow{x'\colon\_})\colon\_\ = e \in \{\overrightarrow{fd}\} \\ \tau^o \neq C[\![\_]\!] \qquad witness_p(C[\![\overrightarrow{\tau}]\!], \tau^o) = S \end{array}}{mbody_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!], \tau^o) = \{(\overrightarrow{x'}) \to [\overrightarrow{\tau'}/\overrightarrow{\alpha'}](S([\overrightarrow{\tau}/\overrightarrow{\alpha}]e))\}}$$

[MB-SUPER]
$$\frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f \notin \{\overrightarrow{Fname(fd)}\} \qquad C \neq C'}{mbody_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!], C'[\![\overrightarrow{\tau^c}]\!]) = \bigcup_{N_i \in \{\overrightarrow{N}\}} mbody_p(f[\![\overrightarrow{\tau'}]\!], [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i, C'[\![\overrightarrow{\tau^c}]\!]))}$$

[MB-STATIC]
$$\frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f \notin \{\overrightarrow{Fname(fd)}\}}{mbody_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!], C[\![\overrightarrow{\tau^c}]\!]) = \bigcup_{N_i \in \{\overrightarrow{N}\}} mbody_p(f[\![\overrightarrow{\tau'}]\!], [\overrightarrow{\tau^c}/\overrightarrow{\alpha}]N_i, \texttt{Object})}$$

[MB-OBJ]
$$mbody_p(f[\![\overrightarrow{\tau}]\!], \texttt{Object}, \tau) = \emptyset$$

Figure A.7: Dynamic Semantics of Core Fortress with Where Clauses (I)

Function/method name lookup: $\boxed{Fname(fd) = f}$

$Fname(f\overrightarrow{[\![\alpha \texttt{ extends } \{K\}]\!]}(\overrightarrow{x{:}\tau}){:}\tau = e) = f$

Types of values: $\boxed{type(v) = \tau}$

$type(O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})) = O[\![\overrightarrow{\tau}]\!]$

Finding witnesses from the static type of the receiver: $\boxed{witness_p(\tau, \tau) = \text{S}}$

$$witness_p(\tau, \tau') = \begin{cases} [\,] & \text{if } \tau' = \texttt{Object} \\ match(C[\![\overrightarrow{\tau}]\!], C[\![\overrightarrow{\tau'}]\!]) & \text{if } \tau = C[\![\overrightarrow{\tau}]\!] \text{ and } \tau' = C[\![\overrightarrow{\tau'}]\!] \\ S_n \cdots S_1 & \text{if } \tau = C[\![\overrightarrow{\tau}]\!],\ \tau' = C'[\![\overrightarrow{\tau'}]\!],\ C \neq C', \\ & \_\ C[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_ \in p, \\ & \text{and } witness_p([\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i, C'[\![\overrightarrow{\tau'}]\!]) = S_i \text{ for } N_i \in \{\overrightarrow{N}\} \\ [\,] & \text{otherwise} \end{cases}$$

Match two types to get substitutions: $\boxed{match(\tau, \tau) = \text{S}}$

$$match(\tau, \tau') = \begin{cases} [\,] & \text{if } \tau = \tau' \\ [\tau'/\beta] & \text{if } \tau = \beta \\ S_n \cdots S_1 & \text{if } \tau = C[\![\overrightarrow{\tau}]\!],\ \tau' = C[\![\overrightarrow{\tau'}]\!], \text{ and } match(S_{i-1} \cdots S_1\tau_i, \tau_i') = S_i \text{ for } 1 \leq i \leq n \\ [\,] & \text{otherwise} \end{cases}$$

Figure A.8: Dynamic Semantics of Core Fortress with Where Clauses (II)

Environments and method types

$$\begin{array}{lcll} \Delta & ::= & \overrightarrow{\alpha \ <: \ \{\overrightarrow{K}\}} & \text{bound environment} \\ \Gamma & ::= & \overrightarrow{x : \tau} & \text{type environment} \\ \eta & ::= & \overrightarrow{[\![\alpha \ \texttt{extends} \ \{\overrightarrow{K}\}]\!]} \ \overrightarrow{\tau} \to \tau & \text{method type} \end{array}$$

Program typing: $\boxed{\vdash p : \tau}$

$$\text{[T-PROGRAM]} \quad \frac{p = \overrightarrow{d} \ e \qquad p \vdash \overrightarrow{d} \ \text{ok} \qquad p; \emptyset; \emptyset \vdash e : \tau}{\vdash p : \tau}$$

Definition typing: $\boxed{p \vdash d \ \text{ok}}$

$$\text{[T-TRAITDEF]} \quad \frac{\begin{array}{c} p \vdash \mathit{validMultipleInheritance}(T) \qquad \Delta = \overrightarrow{\alpha \ <: \ \{\overrightarrow{K}\}} \ \overrightarrow{\beta \ <: \ \{\overrightarrow{H}\}} \\ p; \Delta \vdash \overrightarrow{\overrightarrow{K}} \ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\overrightarrow{N}} \ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\overrightarrow{H}} \ \text{ok} \qquad p; \Delta; \texttt{self} : T[\![\overrightarrow{\alpha}]\!]; T \vdash \overrightarrow{fd} \ \text{ok} \end{array}}{p \vdash \texttt{trait} \ T\overrightarrow{[\![\alpha \ \texttt{extends} \ \{\overrightarrow{K}\}]\!]} \ \texttt{extends} \ \{\overrightarrow{N}\} \ \texttt{where} \ \{\overrightarrow{\beta \ \texttt{extends} \ \{\overrightarrow{H}\}}\} \ \overrightarrow{fd} \ \texttt{end} \ \text{ok}}$$

$$\text{[T-OBJECTDEF]} \quad \frac{\begin{array}{c} p \vdash \mathit{validMultipleInheritance}(O) \qquad \Delta = \overrightarrow{\alpha \ <: \ \{\overrightarrow{K}\}} \qquad \Delta' = \Delta \ \overrightarrow{\beta \ <: \ \{\overrightarrow{H}\}} \\ p; \Delta' \vdash \overrightarrow{\overrightarrow{K}} \ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau} \ \text{ok} \qquad p; \Delta' \vdash \overrightarrow{\overrightarrow{N}} \ \text{ok} \qquad p; \Delta' \vdash \overrightarrow{\overrightarrow{H}} \ \text{ok} \\ p; \Delta'; \texttt{self} : O[\![\overrightarrow{\alpha}]\!] \ \overrightarrow{x : \tau}; O \vdash \overrightarrow{fd} \ \text{ok} \end{array}}{p \vdash \texttt{object} \ O\overrightarrow{[\![\alpha \ \texttt{extends} \ \{\overrightarrow{K}\}]\!]}(\overrightarrow{x:\tau}) \ \texttt{extends} \ \{\overrightarrow{N}\} \ \texttt{where} \ \{\overrightarrow{\beta \ \texttt{extends} \ \{\overrightarrow{H}\}}\} \ \overrightarrow{fd} \ \texttt{end} \ \text{ok}}$$

Method typing: $\boxed{p; \Delta; \Gamma; C \vdash fd \ \text{ok}}$

$$\text{[T-METHODDEF]} \quad \frac{\begin{array}{c} p \vdash \mathit{override}(f, C, \overrightarrow{[\![\alpha \ \texttt{extends} \ \{\overrightarrow{K}\}]\!]} \ \overrightarrow{\tau} \to \tau_0) \qquad \Delta' = \Delta \ \overrightarrow{\alpha \ <: \ \{\overrightarrow{K}\}} \qquad p; \Delta' \vdash \overrightarrow{\overrightarrow{K}} \ \text{ok} \\ p; \Delta' \vdash \overrightarrow{\tau} \ \text{ok} \qquad p; \Delta' \vdash \tau_0 \ \text{ok} \qquad p; \Delta'; \Gamma \ \overrightarrow{x : \tau} \vdash e : \tau' \qquad p; \Delta' \vdash \tau' \ <: \ \tau_0 \end{array}}{p; \Delta; \Gamma; C \vdash f\overrightarrow{[\![\alpha \ \texttt{extends} \ \{\overrightarrow{K}\}]\!]}(\overrightarrow{x:\tau}):\tau_0 = e \ \text{ok}}$$

Method overriding: $\boxed{p \vdash \mathit{override}(f, C, \eta)}$

$$\text{[OVERRIDE]} \quad \frac{\begin{array}{c} \_ \ \overrightarrow{C[\![\alpha'' \ \texttt{extends} \ \{\overrightarrow{K''}\}]\!]} \ \_ \ \overrightarrow{\texttt{extends} \ \{\overrightarrow{N}\} \ \texttt{where} \ \{\beta \ \texttt{extends} \ \{\overrightarrow{H}\}\}} \ \_ \ \in p \\ \Delta = \alpha'' \ <: \ \{\overrightarrow{K''}\} \ \overrightarrow{\beta \ <: \ \{\overrightarrow{H}\}} \\ \bigcup_{C'\overrightarrow{[\![\tau'']\!]} \in \{\overrightarrow{N}\}} \mathit{mtype}_p(f, C'\overrightarrow{[\![\tau'']\!]}, \Delta) = \{(\overrightarrow{[\![\alpha' \ \texttt{extends} \ \{\overrightarrow{K'}\}]\!]} \ \overrightarrow{\tau'} \to \tau'_0, \Delta')\} \\ \overrightarrow{\overrightarrow{K}} = [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\overrightarrow{\overrightarrow{K'}} \qquad p; \Delta' \vdash [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\overrightarrow{\tau'} \ <: \ \overrightarrow{\tau} \qquad p; \Delta' \vdash \tau_0 \ <: \ [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_0 \end{array}}{p \vdash \mathit{override}(f, C, \overrightarrow{[\![\alpha \ \texttt{extends} \ \{\overrightarrow{K}\}]\!]} \ \overrightarrow{\tau} \to \tau_0)}$$

Valid multiple inheritance: $\boxed{p \vdash \mathit{validMultipleInheritance}(C)}$

$$\text{[VALIDMI]} \quad \frac{p \vdash \mathit{oneOwner}(C) \qquad p \vdash \mathit{validWhere}(C)}{p \vdash \mathit{validMultipleInheritance}(C)}$$

Figure A.9: Static Semantics of Core Fortress with Where Clauses (I)

One owner for all the visible methods: $\boxed{p \vdash oneOwner(C)}$

$$[\textsc{OneOwner}] \quad \frac{\forall f \in visible_p(C)\ .\ f \text{ only occurs once in } visible_p(C)}{p \vdash oneOwner(C)}$$

Valid where clauses: $\boxed{p \vdash validWhere(C)}$

$$[\textsc{ValidWhere}] \quad \frac{\begin{array}{l}\forall f \in visible_p(C)\ . \\ \text{where} \quad \_\, C[\![\overrightarrow{\alpha}\ \_]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_ \in p \\ \qquad mbody_p(f[\![\overrightarrow{\alpha^f}]\!], C[\![\overrightarrow{\alpha}]\!], C[\![\overrightarrow{\alpha}]\!]) = \{\_ \to e_f\}, \qquad mtype_p(f, C[\![\overrightarrow{\alpha}]\!], \emptyset) = \{(\eta_f, \Delta)\} \\ \qquad 1.\quad \forall \beta \in (FTV(e_f) \setminus \{\overrightarrow{\alpha}\ \overrightarrow{\alpha^f}\})\ .\ \beta \in FTV(\eta_f) \\ \qquad 2.\quad \forall \beta \in (FTV(\eta_f) \setminus \{\overrightarrow{\alpha}\ \overrightarrow{\alpha^f}\})\ .\ \forall C'[\![\overrightarrow{\tau^c}]\!] \in \bigcup_{N_i \in \{\overrightarrow{N}\}} defining_p(f, N_i)\ . \\ \qquad\qquad \beta = \tau_i^c \quad \Delta(\beta) = \overrightarrow{K'_i} \quad \text{for } 1 \le i \le |\overrightarrow{\tau^c}| \text{ where } \_\ \overrightarrow{C'[\![\alpha'\ \texttt{extends}\ \{\overrightarrow{K'}\}]\!]}\ \_ \in p\end{array}}{p \vdash validWhere(C)}$$

Valid witnesses: $\boxed{p \vdash validWitness(\Delta, \overrightarrow{\alpha\ <:\ \{\overrightarrow{\tau}\}}, \overrightarrow{\tau})}$

$$[\textsc{ValidWitness}] \quad \frac{\begin{array}{c}p; \Delta \vdash \overrightarrow{[\overrightarrow{\tau^\beta}/\overrightarrow{\beta}]\tau}\ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau^\beta}\ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau^\beta}\ <:\ \overrightarrow{[\overrightarrow{\tau^\beta}/\overrightarrow{\beta}]\tau} \\ \{\overrightarrow{\beta}\} \cap dom(\Delta) = \emptyset\end{array}}{p \vdash validWitness(\Delta, \overrightarrow{\beta\ <:\ \{\overrightarrow{\tau}\}}, \overrightarrow{\tau^\beta})}$$

Expression typing: $\boxed{p; \Delta; \Gamma \vdash e : \tau}$

$$[\text{T-}\textsc{Var}] \quad p; \Delta; \Gamma \vdash x : \Gamma(x)$$

$$[\text{T-}\textsc{Self}] \quad p; \Delta; \Gamma \vdash \texttt{self} : \Gamma(\texttt{self})$$

$$[\text{T-}\textsc{Object}] \quad \frac{\begin{array}{c}\texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ (\overrightarrow{\_ : \tau'})\ \_\ \texttt{end} \in p \qquad p; \Delta \vdash O[\![\overrightarrow{\tau}]\!]\ \text{ok} \\ p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \qquad p; \Delta \vdash \overrightarrow{\tau''}\ <:\ \overrightarrow{[\overrightarrow{\tau}/\overrightarrow{\alpha}]\tau'}\end{array}}{p; \Delta; \Gamma \vdash O[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) : O[\![\overrightarrow{\tau}]\!]}$$

$$[\text{T-}\textsc{Field}] \quad \frac{p; \Delta; \Gamma \vdash e_0 : \tau_0 \quad bound_\Delta(\tau_0) = \{O[\![\overrightarrow{\tau'}]\!]\} \quad \texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!](\overrightarrow{x:\tau})\ \_\ \texttt{end} \in p}{p; \Delta; \Gamma \vdash e_0\,.\,x_i : \overrightarrow{[\overrightarrow{\tau'}/\overrightarrow{\alpha}]\tau_i}}$$

$$[\text{T-}\textsc{Method}] \quad \frac{\begin{array}{c}p; \Delta; \Gamma \vdash e_0 : \tau_0 \quad p; \Delta \vdash \tau_0\ <:\ C[\![\overrightarrow{\tau^c}]\!] \quad mtype_p(f, C[\![\overrightarrow{\tau^c}]\!], \emptyset) = \{(\overrightarrow{[\![\alpha'\ \texttt{extends}\ \{\overrightarrow{K'}\}]\!]}\ \overrightarrow{\tau'} \to \tau'_0, \Delta')\} \\ p; \Delta \vdash \overrightarrow{\tau}\ \text{ok} \qquad p; \Delta \vdash C[\![\overrightarrow{\tau^c}]\!]\ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau^a}\ \text{ok} \qquad p; \Delta \vdash \tau^r\ \text{ok} \\ p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \quad p; \Delta \vdash \overrightarrow{\tau''}\ <:\ \overrightarrow{\tau^a} \quad dom(\Delta') = \{\overrightarrow{\beta}\} \quad S = [\overrightarrow{\tau^\beta}/\overrightarrow{\beta}] \\ p \vdash validWitness(\Delta, \Delta', \overrightarrow{\tau^\beta}) \quad p; \Delta \vdash \overrightarrow{\tau}\ <:\ S(\overrightarrow{[\overrightarrow{\tau}/\overrightarrow{\alpha'}]K'}) \quad S(\overrightarrow{[\overrightarrow{\tau}/\overrightarrow{\alpha'}]\tau'}) = \overrightarrow{\tau^a} \quad S([\overrightarrow{\tau}/\overrightarrow{\alpha'}]\tau'_0) = \tau^r\end{array}}{p; \Delta; \Gamma \vdash e_0\ \texttt{as}\ C[\![\overrightarrow{\tau^c}]\!]\,.\,f[\![\overrightarrow{\tau}]\!](\overrightarrow{e\ \texttt{as}\ \tau^a})\texttt{as}\ \tau^r : \tau^r}$$

$$[\text{T-}\textsc{Typecase}] \quad \frac{\begin{array}{c}p; \Delta; \Gamma \vdash e : \tau \qquad p; \Delta; \Gamma\ x : \tau_i \vdash e_i : \tau_i^e \qquad p; \Delta \vdash \tau_i^e\ <:\ \tau' \qquad 1 \le i \le |\overrightarrow{\tau}| \\ p; \Delta; \Gamma\ x : \tau \vdash e' : \tau^{e'} \qquad p; \Delta \vdash \tau^{e'}\ <:\ \tau'\end{array}}{p; \Delta; \Gamma \vdash \texttt{typecase}\ x\ = e\ \texttt{in}\ \overrightarrow{\tau \Rightarrow e}\ \texttt{else}\ e'\ \texttt{end} : \tau'}$$

Figure A.10: Static Semantics of Core Fortress with Where Clauses (II)

Subtyping: $\boxed{p; \Delta \vdash \tau \ <: \ \tau}$

[S-OBJ] $$p; \Delta \vdash \tau \ <: \ \texttt{Object}$$

[S-REFL] $$p; \Delta \vdash \tau \ <: \ \tau$$

[S-TRANS] $$\frac{p; \Delta \vdash \tau_1 \ <: \ \tau_2 \qquad p; \Delta \vdash \tau_2 \ <: \ \tau_3}{p; \Delta \vdash \tau_1 \ <: \ \tau_3}$$

[S-VAR] $$\frac{\tau \in \Delta(\alpha)}{p; \Delta \vdash \alpha \ <: \ \tau}$$

[S-TAPP] $$\frac{\begin{array}{c} \_ \ C[\![\overrightarrow{\alpha \ \texttt{extends} \ \{K\}}]\!] \ \texttt{extends} \ \{\overrightarrow{N}\} \ \texttt{where} \ \{\overrightarrow{\beta \ \texttt{extends} \ \{H\}}\} \ \_ \ \in p \\ p; \Delta \vdash \overrightarrow{\tau} \ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau} \ <: \ [\overrightarrow{\tau^\beta}/\overrightarrow{\beta}][\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{K} \qquad p \vdash \mathit{validWitness}(\Delta, \beta \ <: \ \{[\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{H}\}, \overrightarrow{\tau^\beta}) \end{array}}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!] \ <: \ [\overrightarrow{\tau^\beta}/\overrightarrow{\beta}][\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i}$$

Well-formed types: $\boxed{p; \Delta \vdash \tau \ \text{ok}}$

[W-OBJ] $$p; \Delta \vdash \texttt{Object} \ \text{ok}$$

[W-VAR] $$\frac{\alpha \in \mathit{dom}(\Delta)}{p; \Delta \vdash \alpha \ \text{ok}}$$

[W-TAPP] $$\frac{\begin{array}{c} \_ \ C[\![\overrightarrow{\alpha \ \texttt{extends} \ \{K\}}]\!] \ \_ \ \texttt{where} \ \{\overrightarrow{\beta \ \texttt{extends} \ \{H\}}\} \ \_ \ \in p \\ p; \Delta \vdash \overrightarrow{\tau} \ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau} \ <: \ [\overrightarrow{\tau^\beta}/\overrightarrow{\beta}][\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{K} \qquad p \vdash \mathit{validWitness}(\Delta, \beta \ <: \ \{[\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{H}\}, \overrightarrow{\tau^\beta}) \end{array}}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!] \ \text{ok}}$$

Method type lookup: $\boxed{\mathit{mtype}_p(f, \tau, \Delta) = \{(\eta, \Delta)\}}$

[MT-SELF] $$\frac{\begin{array}{c} \_ \ C[\![\overrightarrow{\alpha \ \texttt{extends} \ \_}]\!] \ \_ \ \texttt{where} \ \{\overrightarrow{\beta \ \texttt{extends} \ \{H\}}\} \ \_ \ \overrightarrow{fd} \ \_ \ \in p \qquad f[\![\overrightarrow{\alpha' \ \texttt{extends} \ \{K'\}}]\!](\overrightarrow{\_ : \tau'}) : \tau_0 \ \_ \ \in \{\overrightarrow{fd}\} \\ \Delta' = \Delta \ \overrightarrow{\beta \ <: \ \{H\}} \end{array}}{\mathit{mtype}_p(f, C[\![\overrightarrow{\tau}]\!], \Delta) = \{([\overrightarrow{\tau}/\overrightarrow{\alpha}][\![\overrightarrow{\alpha' \ \texttt{extends} \ \{K'\}}]\!] \ \overrightarrow{\tau'} \to \tau_0, [\overrightarrow{\tau}/\overrightarrow{\alpha}]\Delta')\}}$$

[MT-SUPER] $$\frac{\begin{array}{c} \_ \ C[\![\overrightarrow{\alpha \ \texttt{extends} \ \_}]\!] \ \texttt{extends} \ \{\overrightarrow{N}\} \ \texttt{where} \ \{\overrightarrow{\beta \ \texttt{extends} \ \{H\}}\} \ \_ \ \overrightarrow{fd} \ \_ \ \in p \qquad f \notin \{\overrightarrow{\mathit{Fname}(fd)}\} \\ \Delta' = \Delta \ \overrightarrow{\beta \ <: \ \{H\}} \end{array}}{\mathit{mtype}_p(f, C[\![\overrightarrow{\tau}]\!], \Delta) = \bigcup_{N_i \in \{\overrightarrow{N}\}} \mathit{mtype}_p(f, [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i, [\overrightarrow{\tau}/\overrightarrow{\alpha}]\Delta')}$$

[MT-OBJ] $$\mathit{mtype}_p(f, \texttt{Object}, \Delta) = \emptyset$$

Bound of type: $\boxed{\mathit{bound}_\Delta(\tau) = \{\overrightarrow{\sigma}\}}$

$$\begin{array}{lcl} \mathit{bound}_\Delta(\alpha) & = & \bigcup_{\tau_i \in \Delta(\alpha)} \mathit{bound}_\Delta(\tau_i) \\ \mathit{bound}_\Delta(\sigma) & = & \{\sigma\} \end{array}$$

Figure A.11: Static Semantics of Core Fortress with Where Clauses (III)

Traits defining a method: $\boxed{\mathit{defining}_p(f,\, N) = \{\overrightarrow{N}\}}$

$$\mathit{defining}_p(f, \texttt{Object}) = \emptyset$$

$$\mathit{defining}_p(f, C[\![\overrightarrow{\tau}]\!]) = \begin{cases} \bigcup_{N_i \in \{\overrightarrow{N}\}} \mathit{defining}_p(f, [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i) & \\ \qquad\qquad \text{if } \_\, C[\![\overrightarrow{\alpha \_}]\!] \,\_\, \texttt{extends}\, \{\overrightarrow{N}\} \,\_ \in p \text{ and } f \notin \mathit{defined}_p(C) & \\ \bigcup_{N_i \in \{\overrightarrow{N}\}} \mathit{defining}_p(f, [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i) \cup \{C[\![\overrightarrow{\tau}]\!]\} & \\ \qquad\qquad \text{if } \_\, C[\![\overrightarrow{\alpha \_}]\!] \,\_\, \texttt{extends}\, \{\overrightarrow{N}\} \,\_ \in p \text{ and } f \in \mathit{defined}_p(C) & \end{cases}$$

Auxiliary functions for methods: $\boxed{\mathit{defined}_p \,/\, \mathit{inherited}_p \,/\, \mathit{visible}_p(C) = \{\overrightarrow{f}\}}$

$$\mathit{defined}_p(C) = \{\overrightarrow{\mathit{Fname}(\mathit{fd})}\} \qquad \text{where } \_\, C \,\_\, \overrightarrow{fd} \,\_ \in p$$

$$\mathit{inherited}_p(C) = \biguplus_{N_i \in \{\overrightarrow{N}\}} \{f_i \mid f_i \in \mathit{visible}_p(N_i), f_i \notin \mathit{defined}_p(C)\} \qquad \text{where } \_\, C \,\_\, \texttt{extends}\, \{\overrightarrow{N}\} \,\_ \in p$$

$$\mathit{visible}_p(C) = \mathit{defined}_p(C) \uplus \mathit{inherited}_p(C)$$

Figure A.12: Static Semantics of Core Fortress with Where Clauses (IV)

A.3 Core Fortress with Overloading

In this section, we define a Fortress core calculus with overloading for dotted methods and first-order functions. We call this calculus *Core Fortress with Overloading*. Core Fortress with Overloading is an extension of Basic Core Fortress with overloading.

A.3.1 Syntax

The syntax for Core Fortress with Overloading is provided in Figure A.13.

A.3.2 Dynamic Semantics

A dynamic semantics for Core Fortress with Overloading is provided in Figure A.14.

A.3.3 Static Semantics

A static semantics for Core Fortress with Overloading is provided in Figures A.15, A.16, A.17, and A.18.

We proved the type soundness of Core Fortress with Overloading using the standard technique of proving a progress theorem and a subject reduction theorem.

α, β			type variables
f			function or method name
x			field name
T			trait name
O			object name
τ, τ', τ''	$::=$	α	type
	$\mid$	σ	
σ	$::=$	N	type that is not a type variable
	$\mid$	$O[\![\overrightarrow{\tau}]\!]$	
N, M, L	$::=$	$T[\![\overrightarrow{\tau}]\!]$	type that can be a type bound
	$\mid$	`Object`	
e	$::=$	x	expression
	$\mid$	`self`	
	$\mid$	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
	$\mid$	$e.x$	
	$\mid$	$e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
	$\mid$	$f[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
fd	$::=$	$f[\![\overrightarrow{\alpha \texttt{ extends } N}]\!](\overrightarrow{x{:}\tau}){:}\tau = e$	function or method definition
td	$::=$	$\texttt{trait } T[\![\overrightarrow{\alpha \texttt{ extends } N}]\!] \texttt{ extends } \{\overrightarrow{N}\} \ \overrightarrow{fd} \texttt{ end}$	trait definition
od	$::=$	$\texttt{object } O[\![\overrightarrow{\alpha \texttt{ extends } N}]\!](\overrightarrow{x{:}\tau}) \texttt{ extends } \{\overrightarrow{N}\} \ \overrightarrow{fd} \texttt{ end}$	object definition
d	$::=$	fd	definition
	$\mid$	td	
	$\mid$	od	
p	$::=$	$\overrightarrow{d} \ e$	program

Figure A.13: Syntax of Core Fortress with Overloading

Values, evaluation contexts, and redexes

$$
\begin{array}{lcll}
v & ::= & O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}) & \text{value} \\
E & ::= & \Box & \text{evaluation context} \\
 & | & O[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e}) & \\
 & | & E.x & \\
 & | & E.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) & \\
 & | & e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e}) & \\
 & | & f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e}) & \\
R & ::= & v.x & \text{redex} \\
 & | & v.f[\![\overrightarrow{\tau}]\!](\overrightarrow{v}) & \\
 & | & f[\![\overrightarrow{\tau}]\!](\overrightarrow{v}) &
\end{array}
$$

Evaluation rules: $\boxed{p \vdash E[R] \longrightarrow E[e]}$

[R-FIELD]
$$\frac{\texttt{object}\ O\ \_\ (\overrightarrow{x\colon \_})\ \_\ \texttt{end} \in p}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).x_i] \longrightarrow E[v_i]}$$

[R-METHOD]
$$\frac{\texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x\colon \_})\ \_\ \texttt{end} \in p \qquad \overrightarrow{type(v')} = \overrightarrow{\tau''} \qquad mostspecific_{p;\emptyset}(applicable_{p;\emptyset}(f[\![\overrightarrow{\tau'}]\!](\overrightarrow{\tau''}), visible_p(O[\![\overrightarrow{\tau}]\!])) = f[\![\overrightarrow{\alpha'\ \texttt{extends}\ N'}]\!](\overrightarrow{x'\colon \_})\colon \_ = e}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).f[\![\overrightarrow{\tau'}]\!](\overrightarrow{v'})] \longrightarrow E[[\overrightarrow{v}/\overrightarrow{x}][O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x'}]e]}$$

[R-FUNCTION]
$$\frac{\overrightarrow{type(v)} = \overrightarrow{\tau'} \qquad mostspecific_{p;\emptyset}(applicable_{p;\emptyset}(f[\![\overrightarrow{\tau}]\!](\overrightarrow{\tau'}), \{(fd, \texttt{Object}) \mid fd \in p\})) = f[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x\colon \_})\colon \_ = e}{p \vdash E[f[\![\overrightarrow{\tau}]\!](\overrightarrow{v})] \longrightarrow E[[\overrightarrow{v}/\overrightarrow{x}]e]}$$

Types of values: $\boxed{type(v) = \tau}$

$type(O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})) = O[\![\overrightarrow{\tau}]\!]$

Figure A.14: Dynamic Semantics of Core Fortress with Overloading

Environments and trait or object names

Δ	::=	$\overrightarrow{\alpha <: N}$	bound environment
Γ	::=	$\overrightarrow{x : \tau}$	type environment
C	::=	T	trait name
	|	O	object name

Program typing: $\boxed{\vdash p : \tau}$

$$\text{[T-PROGRAM]} \quad \frac{p = \overrightarrow{d}\ e \qquad p;\emptyset;\emptyset \vdash \overrightarrow{d}\ \text{ok} \qquad p \vdash \mathit{validFun}(\overrightarrow{d}) \qquad p;\emptyset;\emptyset \vdash e : \tau}{\vdash p : \tau}$$

Trait typing: $\boxed{p;\emptyset;\emptyset \vdash td\ \text{ok}}$

$$\text{[T-TRAITDEF]} \quad \frac{\begin{array}{c}\Delta = \overrightarrow{\alpha <: N} \qquad p;\Delta \vdash \overrightarrow{N}\ \text{ok} \qquad p;\Delta \vdash \overrightarrow{M}\ \text{ok} \\ p;\Delta;\texttt{self} : T[\![\overrightarrow{\alpha}]\!] \vdash \overrightarrow{fd}\ \text{ok} \qquad p \vdash \mathit{validMeth}(T)\end{array}}{p;\emptyset;\emptyset \vdash \texttt{trait}\ T[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \texttt{extends}\ \{\overrightarrow{M}\}\ \overrightarrow{fd}\ \texttt{end}\ \text{ok}}$$

Object typing: $\boxed{p;\emptyset;\emptyset \vdash od\ \text{ok}}$

$$\text{[T-OBJECTDEF]} \quad \frac{\begin{array}{c}\Delta = \overrightarrow{\alpha <: N} \qquad p;\Delta \vdash \overrightarrow{N}\ \text{ok} \qquad p;\Delta \vdash \overrightarrow{\tau}\ \text{ok} \qquad p;\Delta \vdash \overrightarrow{M}\ \text{ok} \\ p;\Delta;\texttt{self} : O[\![\overrightarrow{\alpha}]\!]\ \overrightarrow{x : \tau} \vdash \overrightarrow{fd}\ \text{ok} \qquad p \vdash \mathit{validMeth}(O)\end{array}}{p;\emptyset;\emptyset \vdash \texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x:\tau})\ \texttt{extends}\ \{\overrightarrow{M}\}\ \overrightarrow{fd}\ \texttt{end}\ \text{ok}}$$

Function and method typing: $\boxed{p;\Delta;\Gamma \vdash fd\ \text{ok}}$

$$\text{[T-FUNMETHDEF]} \quad \frac{\begin{array}{c}\Delta' = \Delta\ \overrightarrow{\alpha <: \{\overrightarrow{N}\}} \qquad p;\Delta' \vdash \overrightarrow{N}\ \text{ok} \qquad p;\Delta' \vdash \overrightarrow{\tau}\ \text{ok} \qquad p;\Delta' \vdash \tau_0\ \text{ok} \\ p;\Delta';\Gamma\ \overrightarrow{x : \tau} \vdash e : \tau' \qquad p;\Delta' \vdash \tau' <: \tau_0\end{array}}{p;\Delta;\Gamma \vdash f[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x:\tau}):\tau_0 = e\ \text{ok}}$$

Valid method declarations: $\boxed{p \vdash \mathit{validMeth}(C)}$

$$\text{[VALIDMETH]} \quad \frac{\begin{array}{l}\forall\,(fd, C[\![\overrightarrow{\tau^c}]\!]), (fd', C'[\![\overrightarrow{\tau^{c'}}]\!]) \in \mathit{visible}_p(C^o[\![\overrightarrow{\alpha^o}]\!]). \\ \quad \text{where} \quad \_\ C^o[\![\overrightarrow{\alpha^o\ \texttt{extends}\ \_}]\!]\ \_ \in p, \\ \qquad fd \neq fd' \quad \text{(not same declaration)}, \\ \qquad fd = f[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{\_:\tau}):\tau^r = \_, \quad fd' = f[\![\overrightarrow{\alpha'\ \texttt{extends}\ N'}]\!](\overrightarrow{\_:\tau'}):\tau^{r'} = \_, \\ p \vdash \mathit{valid}(\overrightarrow{[\![\alpha\ \texttt{extends}\ N]\!]}C[\![\overrightarrow{\tau^c}]\!]\overrightarrow{\tau} \to \tau^r, \overrightarrow{[\![\alpha'\ \texttt{extends}\ N']\!]}C'[\![\overrightarrow{\tau^{c'}}]\!]\overrightarrow{\tau'} \to \tau^{r'}, \mathit{visible}_p(C^o[\![\overrightarrow{\alpha^o}]\!]))\end{array}}{p \vdash \mathit{validMeth}(C^o)}$$

Figure A.15: Static Semantics of Core Fortress with Overloading (I)

Valid function declarations: $\boxed{p \vdash \mathit{validFun}(\overrightarrow{d})}$

$$[\textsc{ValidFun}] \quad \frac{\begin{array}{l}\forall\, fd, fd' \in \overrightarrow{d}\,. \\ \text{where} \quad fd \neq fd' \quad \text{(not same declaration)}, \\ \qquad fd = f\overrightarrow{[\![\alpha \texttt{ extends } N]\!]}(\overrightarrow{\_:\tau}):\tau^r = \_\,, \quad fd' = f\overrightarrow{[\![\alpha' \texttt{ extends } N']\!]}(\overrightarrow{\_:\tau'}):\tau^{r'} = \_\,, \\ p \vdash \mathit{valid}(\overrightarrow{[\![\alpha \texttt{ extends } N]\!]}\texttt{Object}\,\overrightarrow{\tau} \to \tau^r, \overrightarrow{[\![\alpha' \texttt{ extends } N']\!]}\texttt{Object}\,\overrightarrow{\tau'} \to \tau^{r'}, \{(fd, \texttt{Object}) \mid fd \in \overrightarrow{d}\})\end{array}}{p \vdash \mathit{validFun}(\overrightarrow{d})}$$

Valid declarations: $\boxed{p \vdash \mathit{valid}(\overrightarrow{[\![\alpha \texttt{ extends } N]\!]}\,\overrightarrow{\tau} \to \tau, \overrightarrow{[\![\alpha \texttt{ extends } N]\!]}\,\overrightarrow{\tau} \to \tau, \{\overrightarrow{(fd, \tau)}\})}$

$$[\textsc{Valid}] \quad \frac{\begin{array}{l}\Delta = \overrightarrow{\alpha \;<:\; \{N\}}, \quad |\overrightarrow{\tau}| = n \\ \quad 1. \quad |\overrightarrow{\tau}| \neq |\overrightarrow{\tau'}| \\ \vee \;\; 2. \quad 1)\; \overrightarrow{N} = [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\overrightarrow{N'} \\ \qquad \wedge \;\; 2)\; \forall\, 1 \leq i \leq n.\; p; \Delta \vdash \tau_i \;<:\; [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i \quad \vee \quad p; \Delta \vdash [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i \;<:\; \tau_i \\ \qquad \wedge \;\; 3)\; \exists\, 1 \leq i \leq n.\; \tau_i \neq [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i \\ \qquad \wedge \;\; 4)\; \exists\, (f\overrightarrow{[\![\alpha'' \texttt{ extends } N'']\!]}(\overrightarrow{\_:\tau''}):\tau^{r''} = \_\,, \tau''_0) \in S. \\ \qquad\quad \text{where} \\ \qquad\qquad \forall\, 0 \leq i \leq n. \quad p; \Delta \vdash \mathit{meet}(\{\tau_i, [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i\}, [\overrightarrow{\alpha}/\overrightarrow{\alpha''}]\tau''_i) \\ \qquad\quad \wedge \;\; \overrightarrow{N} = [\overrightarrow{\alpha}/\overrightarrow{\alpha''}]\overrightarrow{N''} \\ \qquad\quad \wedge \;\; p; \Delta \vdash [\overrightarrow{\alpha}/\overrightarrow{\alpha''}]\tau^{r''} \;<:\; \tau^r \\ \qquad\quad \wedge \;\; p; \Delta \vdash [\overrightarrow{\alpha'}/\overrightarrow{\alpha''}]\tau^{r''} \;<:\; \tau^{r'}\end{array}}{p \vdash \mathit{valid}(\overrightarrow{[\![\alpha \texttt{ extends } N]\!]}\,\overrightarrow{\tau} \to \tau^r, \overrightarrow{[\![\alpha' \texttt{ extends } N']\!]}\,\overrightarrow{\tau'} \to \tau^{r'}, S)}$$

Expression typing: $\boxed{p; \Delta; \Gamma \vdash e : \tau}$

$$[\textsc{T-Var}] \quad p; \Delta; \Gamma \vdash x : \Gamma(x)$$

$$[\textsc{T-Self}] \quad p; \Delta; \Gamma \vdash \texttt{self} : \Gamma(\texttt{self})$$

$$[\textsc{T-Object}] \quad \frac{\texttt{object}\; O\overrightarrow{[\![\alpha \texttt{ extends } \_]\!]}(\overrightarrow{\_:\tau'})\; \_\; \texttt{end} \in p \qquad p; \Delta \vdash O[\![\overrightarrow{\tau}]\!]\,\text{ok} \qquad p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \qquad p; \Delta \vdash \overrightarrow{\tau''} \;<:\; [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{\tau'}}{p; \Delta; \Gamma \vdash O[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) : O[\![\overrightarrow{\tau}]\!]}$$

$$[\textsc{T-Field}] \quad \frac{p; \Delta; \Gamma \vdash e_0 : \tau_0 \qquad \mathit{bound}_\Delta(\tau_0) = O[\![\overrightarrow{\tau'}]\!] \qquad \texttt{object}\; O\overrightarrow{[\![\alpha \texttt{ extends } \_]\!]}(\overrightarrow{x:\tau})\; \_\; \texttt{end} \in p}{p; \Delta; \Gamma \vdash e_0\texttt{.}x_i : [\overrightarrow{\tau'}/\overrightarrow{\alpha}]\tau_i}$$

$$[\textsc{T-Method}] \quad \frac{p; \Delta; \Gamma \vdash e_0 : \tau_0 \qquad p; \Delta \vdash \overrightarrow{\tau}\;\text{ok} \qquad p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau'} \qquad \mathit{mostspecific}_{p;\Delta}(\mathit{applicable}_{p;\Delta}(f[\![\overrightarrow{\tau}]\!](\overrightarrow{\tau'}), \mathit{visible}_p(\mathit{bound}_\Delta(\tau_0)))) = f\overrightarrow{[\![\alpha \texttt{ extends } N]\!]}(\_):\tau^r\;\_}{p; \Delta; \Gamma \vdash e_0.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) : \tau^r}$$

$$[\textsc{T-Function}] \quad \frac{p; \Delta \vdash \overrightarrow{\tau}\;\text{ok} \qquad p; \Delta; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau'} \qquad \mathit{mostspecific}_{p;\Delta}(\mathit{applicable}_{p;\Delta}(f[\![\overrightarrow{\tau}]\!](\overrightarrow{\tau'}), \{(fd, \texttt{Object}) \mid fd \in p\})) = f\overrightarrow{[\![\alpha \texttt{ extends } N]\!]}(\_):\tau^r\;\_}{p; \Delta; \Gamma \vdash f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) : \tau^r}$$

Figure A.16: Static Semantics of Core Fortress with Overloading (II)

Subtyping: $\boxed{p; \Delta \vdash \tau \ <: \ \tau}$

[S-OBJ] $$p; \Delta \vdash \tau \ <: \ \texttt{Object}$$

[S-REFL] $$p; \Delta \vdash \tau \ <: \ \tau$$

[S-TRANS] $$\frac{p; \Delta \vdash \tau_1 \ <: \ \tau_2 \qquad p; \Delta \vdash \tau_2 \ <: \ \tau_3}{p; \Delta \vdash \tau_1 \ <: \ \tau_3}$$

[S-VAR] $$p; \Delta \vdash \alpha \ <: \ \Delta(\alpha)$$

[S-TAPP] $$\frac{\_ \ C[\![\overrightarrow{\alpha \ \texttt{extends} \ \_}]\!] \ \_ \ \texttt{extends} \ \{\overrightarrow{N}\} \ \_ \in p}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!] \ <: \ [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i}$$

Well-formed types: $\boxed{p; \Delta \vdash \tau \ \text{ok}}$

[W-OBJ] $$p; \Delta \vdash \texttt{Object} \ \text{ok}$$

[W-VAR] $$\frac{\alpha \in \mathit{dom}(\Delta)}{p; \Delta \vdash \alpha \ \text{ok}}$$

[W-TAPP] $$\frac{\_ \ C[\![\overrightarrow{\alpha \ \texttt{extends} \ N}]\!] \ \_ \in p \qquad p; \Delta \vdash \overrightarrow{\tau} \ \text{ok} \qquad p; \Delta \vdash \overrightarrow{\tau} \ <: \ [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{N}}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!] \ \text{ok}}$$

Most specific definitions: $\boxed{\mathit{mostspecific}_{p;\Delta}(\{\overrightarrow{(fd, \tau)}\}) = fd}$

$$\frac{\begin{array}{c}\overrightarrow{fd} = f[\![\overrightarrow{(\alpha \ \texttt{extends} \ N)_1}]\!]((\overrightarrow{\_ : \tau^a})_1) : \tau^r_1 \ \_ \cdots f[\![\overrightarrow{(\alpha \ \texttt{extends} \ N)_n}]\!]((\overrightarrow{\_ : \tau^a})_n) : \tau^r_n \ \_ \\ 1 \leq i \leq n \qquad (\overrightarrow{\tau^a})_i = \tau^a_{i1} \cdots \tau^a_{im} \qquad \tau^a_{i0} = \tau_i \qquad \forall \, 0 \leq j \leq m \, . \, p; \Delta \vdash \mathit{meet}(\{\tau^a_{1j}, \cdots, \tau^a_{nj}\}, \tau^a_{ij})\end{array}}{\mathit{mostspecific}_{p;\Delta}(\{\overrightarrow{(fd, \tau)}\}) = fd_i}$$

Applicable definitions: $\boxed{\mathit{applicable}_{p;\Delta}(f[\![\overrightarrow{\tau}]\!](\overrightarrow{\tau}), \{\overrightarrow{(fd, \tau)}\}) = \{\overrightarrow{(fd, \tau)}\}}$

$$\mathit{applicable}_{p;\Delta}(f[\![\overrightarrow{\tau'}]\!](\overrightarrow{\tau''}), S) = \left\{ \begin{array}{ll} \{([\overrightarrow{\tau'}/\overrightarrow{\alpha}]fd, \tau) \mid & \text{where } (fd, \tau) \in S, \\ & fd = f[\![\overrightarrow{\alpha \ \texttt{extends} \ N}]\!](\overrightarrow{x : \tau'''}) : \_, \\ & p; \Delta \vdash \overrightarrow{\tau''} \ <: \ [\overrightarrow{\tau'}/\overrightarrow{\alpha}]\overrightarrow{\tau'''} \\ & p; \Delta \vdash \overrightarrow{\tau'} \ <: \ [\overrightarrow{\tau'}/\overrightarrow{\alpha}]\overrightarrow{N}\} \end{array} \right.$$

Figure A.17: Static Semantics of Core Fortress with Overloading (III)

Method definition lookup: $\boxed{visible_p \text{ / } defined_p(C[\![\overrightarrow{\tau}]\!]) = \{\overrightarrow{(fd, C[\![\overrightarrow{\tau}]\!])}\}}$

$$visible_p(C[\![\overrightarrow{\tau}]\!]) = defined_p(C[\![\overrightarrow{\tau}]\!]) \cup \bigcup_{C'[\![\overrightarrow{\tau'}]\!] \in \{\overrightarrow{N}\}} visible_p([\overrightarrow{\tau}/\overrightarrow{\alpha}]C'[\![\overrightarrow{\tau'}]\!]) \quad \text{where } \_\ C[\![\overrightarrow{\alpha}\ \_]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_ \in p$$

$$defined_p(C[\![\overrightarrow{\tau}]\!]) = \{\overrightarrow{([\overrightarrow{\tau}/\overrightarrow{\alpha}]fd, C[\![\overrightarrow{\tau}]\!])}\} \quad \text{where } \_\ C[\![\overrightarrow{\alpha}\ \_]\!]\ \_\ \overrightarrow{fd}\ \_ \in p$$

Most specific type: $\boxed{p; \Delta \vdash meet(\{\overrightarrow{\tau}\}, \tau)}$

$$[\textsc{Meet}] \quad \frac{\tau' \in \{\overrightarrow{\tau}\} \qquad \forall\, 1 \le i \le |\overrightarrow{\tau}| \,.\, p; \Delta \vdash \tau' <: \tau_i}{p; \Delta \vdash meet(\{\overrightarrow{\tau}\}, \tau')}$$

Bound of type: $\boxed{bound_\Delta(\tau) = \sigma}$

$$\begin{array}{lcl} bound_\Delta(\alpha) & = & \Delta(\alpha) \\ bound_\Delta(\sigma) & = & \sigma \end{array}$$

Figure A.18: Static Semantics of Core Fortress with Overloading (IV)

A.4 Acyclic Core Fortress with Field Definitions

In this section, we define a Fortress core calculus with acyclic type hierarchy and field definitions inside object definitions. We call this calculus *Acyclic Core Fortress with Field Definitions*. Acyclic Core Fortress with Field Definitions is an extension of Basic Core Fortress with acyclic type hierarchy and field definitions inside object definitions.

A.4.1 Syntax

The syntax for Acyclic Core Fortress with Field Definitions is provided in Figure A.19.

A.4.2 Dynamic Semantics

A dynamic semantics for Acyclic Core Fortress with Field Definitions is provided in Figure A.20.

A.4.3 Static Semantics

A static semantics for Acyclic Core Fortress with Field Definitions is provided in Figures A.21, A.22, and A.23.

We proved the type soundness of Acyclic Core Fortress with Field Definitions and the acyclic type hierarchy of a well-type program in Acyclic Core Fortress with Field Definitions using the standard technique of proving a progress theorem and a subject reduction theorem.

α, β			type variables
f			method name
x, y, z			field names
T			trait name
O			object name
τ, τ', τ''	::=	α	type
	|	σ	
σ	::=	N	type that is not a type variable
	|	$O[\![\overrightarrow{\tau}]\!]$	
N, M, L	::=	$T[\![\overrightarrow{\tau}]\!]$	type that can be a type bound
	|	`Object`	
e	::=	x	expression
	|	`self`	
	|	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
	|	$e.x$	
	|	$e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$	
fd	::=	$f[\![\overrightarrow{\alpha \texttt{ extends } N}]\!](\overrightarrow{x{:}\tau}){:}\tau = e$	method definition
vd	::=	$x{:}\tau = e$	field definition
td	::=	$\texttt{trait } T[\![\overrightarrow{\alpha \texttt{ extends } N}]\!] \texttt{ extends } \{\overrightarrow{N}\}\ \overrightarrow{fd} \texttt{ end}$	trait definition
od	::=	$\texttt{object } O[\![\overrightarrow{\alpha \texttt{ extends } N}]\!](\overrightarrow{x{:}\tau}) \texttt{ extends } \{\overrightarrow{N}\}\ \overrightarrow{fd} \texttt{ end}$	object definition
d	::=	td	definition
	|	od	
p	::=	$\overrightarrow{d}\ e$	program

Figure A.19: Syntax of Acyclic Core Fortress with Field Definitions

Values, intermediate expressions, evaluation contexts, redexes, and trait and object names

v	::=	$O[\![\overrightarrow{\tau}]\!]\{\overrightarrow{x \mapsto v};\}$	value
ϵ	::=	x	intermediate expression
	|	`self`	
	|	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{\epsilon})$	
	|	$\epsilon.x$	
	|	$\epsilon.f[\![\overrightarrow{\tau}]\!](\overrightarrow{\epsilon})$	
	|	$O[\![\overrightarrow{\tau}]\!]\{\overrightarrow{x \mapsto \epsilon}; \overrightarrow{x \mapsto \epsilon}\}$	
E	::=	$\square$	evaluation context
	|	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{\epsilon}\, E\, \overrightarrow{\epsilon})$	
	|	$E.x$	
	|	$E.f[\![\overrightarrow{\tau}]\!](\overrightarrow{\epsilon})$	
	|	$\epsilon.f[\![\overrightarrow{\tau}]\!](\overrightarrow{\epsilon}\, E\, \overrightarrow{\epsilon})$	
	|	$O[\![\overrightarrow{\tau}]\!]\{\overrightarrow{x \mapsto v};\ x \mapsto E\ \overrightarrow{x \mapsto e}\}$	
R	::=	$O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})$	redex
	|	$O[\![\overrightarrow{\tau}]\!]\{\overrightarrow{x \mapsto v};\ x \mapsto v\ \overrightarrow{x \mapsto e}\}$	
	|	$v.x$	
	|	$v.f[\![\overrightarrow{\tau}]\!](\overrightarrow{v})$	
C	::=	T	trait name
	|	O	object name

Evaluation rules: $\boxed{p \vdash E[R] \longrightarrow E[\epsilon]}$

[R-OBJECT]
$$\frac{\texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!](\overrightarrow{x{:}\_})\ \_\ \overrightarrow{x'{:}\_ = e'}\ \_\ \texttt{end} \in p}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})] \longrightarrow E[O[\![\overrightarrow{\tau}]\!]\{\overrightarrow{x \mapsto v}\ ; \overrightarrow{x' \mapsto [\overrightarrow{v}/\overrightarrow{x}][\overrightarrow{\tau}/\overrightarrow{\alpha}]e'}\}]}$$

[R-SUB]
$$p \vdash E[O[\![\overrightarrow{\tau}]\!]\{\overrightarrow{x \mapsto v};\ y \mapsto v\ \overrightarrow{z \mapsto e}\}] \longrightarrow E[O[\![\overrightarrow{\tau}]\!]\{\overrightarrow{x \mapsto v}\ y \mapsto v;\ \overrightarrow{z \mapsto [v/y]e}\}]$$

[R-FIELD]
$$p \vdash E[O[\![\overrightarrow{\tau}]\!]\{\overrightarrow{x \mapsto v};\}.x_i] \longrightarrow E[v_i]$$

[R-METHOD]
$$\frac{mbody_p(f[\![\overrightarrow{\tau'}]\!], O[\![\overrightarrow{\tau}]\!]) = \{(\overrightarrow{x'}) \to e\}}{p \vdash E[O[\![\overrightarrow{\tau}]\!]\{\overrightarrow{x \mapsto v};\}.f[\![\overrightarrow{\tau'}]\!](\overrightarrow{v'})] \longrightarrow E[[\overrightarrow{v}/\overrightarrow{x}][O[\![\overrightarrow{\tau}]\!]\{\overrightarrow{x \mapsto v};\}/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x'}]e]}$$

Method body lookup: $\boxed{mbody_p(f[\![\overrightarrow{\tau}]\!], \tau) = \{(\overrightarrow{x}) \to e\}}$

[MB-SELF]
$$\frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f[\![\overrightarrow{\alpha'\ \texttt{extends}\ \_}]\!](\overrightarrow{x'{:}\_}){:}\_\ = e \in \{\overrightarrow{fd}\}}{mbody_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!]) = \{[\overrightarrow{\tau'}/\overrightarrow{\alpha'}][\overrightarrow{\tau}/\overrightarrow{\alpha}](\overrightarrow{x'}) \to e\}}$$

[MB-SUPER]
$$\frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{K}\}\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f \notin \{\overrightarrow{Fname(fd)}\}}{mbody_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!]) = \bigcup_{N_i \in \{\overrightarrow{N}\}} mbody_p(f[\![\overrightarrow{\tau'}]\!], [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i)}$$

[MB-OBJ]
$$mbody_p(f[\![\overrightarrow{\tau}]\!], \texttt{Object}) = \emptyset$$

Function/method name lookup: $\boxed{Fname(fd) = f}$

$Fname(f[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x{:}\tau}){:}\tau = e) = f$

Figure A.20: Dynamic Semantics of Acyclic Core Fortress with Field Definitions

Environments

$\Delta \quad ::= \quad \overrightarrow{\alpha <: N}$ bound environment
$\Gamma \quad ::= \quad \overrightarrow{x : \tau}$ type environment

Program typing: $\boxed{\vdash p : \tau}$

$$[\text{T-PROGRAM}] \quad \frac{p = \overrightarrow{d}\ e \qquad p;\emptyset;\emptyset \vdash \overrightarrow{d}\ \text{ok} \qquad p;\emptyset;\emptyset \vdash e : \tau \qquad acyclic(\overrightarrow{d})}{\vdash p : \tau}$$

Acyclic type hierarchy: $\boxed{acyclic(\overrightarrow{d})}$

$$[\text{ACYCLIC}] \quad \frac{\begin{array}{c}\texttt{trait}\ T[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \texttt{extends}\ \{\overrightarrow{M}\}\ \_\ \texttt{end} \in \overrightarrow{d} \qquad 1 \le i \le |\overrightarrow{\alpha}| \\ (1)\quad M_i \neq T[\![\ \_\ ]\!] \quad \text{implies} \quad p;\ \_ \vdash [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{M}_i \not<: T[\![\overrightarrow{\tau}]\!] \\ \textit{(The type names excluding self-extensions form an acyclic hierarchy.)} \\ (2)\quad M_i = T[\![\ \_\ ]\!] \quad \text{implies} \quad M_j \neq T[\![\ \_\ ]\!] \quad 1 \le j \le |\overrightarrow{M}| \quad i \neq j \\ (3)\quad M_i = T[\![\overrightarrow{\tau}]\!] \quad \text{implies} \quad (\tau_i = \alpha_i) \vee (\tau_i = N_i) \vee (\tau_i = \alpha_j \wedge \alpha_i = N_j) \quad 1 \le j \le |\overrightarrow{\alpha}| \quad i \neq j\end{array}}{acyclic(\overrightarrow{d})}$$

Definition typing: $\boxed{p;\emptyset;\emptyset \vdash d\ \text{ok}}$

$$[\text{T-TRAITDEF}] \quad \frac{\Delta = \overrightarrow{\alpha <: N} \qquad p;\Delta \vdash \overrightarrow{N}\ \text{ok} \qquad p;\Delta \vdash \overrightarrow{M}\ \text{ok} \qquad p;\Delta;\texttt{self} : T[\![\overrightarrow{\alpha}]\!] \vdash T\ \text{ok} \overrightarrow{fd} \qquad p \vdash oneOwner(T)}{p;\emptyset;\emptyset \vdash \texttt{trait}\ T[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \texttt{extends}\ \{\overrightarrow{M}\}\ \overrightarrow{fd}\ \texttt{end}\ \text{ok}}$$

$$[\text{T-OBJECTDEF}] \quad \frac{\begin{array}{c}\Delta = \overrightarrow{\alpha <: N} \qquad p;\Delta \vdash \overrightarrow{N}\ \text{ok} \qquad p;\Delta \vdash \overrightarrow{\tau}\ \text{ok} \qquad p;\Delta \vdash \overrightarrow{M}\ \text{ok} \\ \overrightarrow{vd} = \overrightarrow{x':\tau' = e} \qquad p;\Delta;\overrightarrow{x:\tau}\ \ x'_1 : \tau'_1\ \ldots\ x'_{i-1} : \tau'_{i-1} \vdash x'_i{:}\tau'_i = e_i\ \text{ok} \quad 1 \le i \le n \\ p;\Delta;\texttt{self} : O[\![\overrightarrow{\alpha}]\!]\ \ \overrightarrow{x:\tau}\ \ \overrightarrow{x':\tau'} \vdash O\ \text{ok} \overrightarrow{fd} \qquad p \vdash oneOwner(O)\end{array}}{p;\emptyset;\emptyset \vdash \texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x:\tau})\ \texttt{extends}\ \{\overrightarrow{M}\}\ \overrightarrow{fd}\ \texttt{end}\ \text{ok}}$$

One owner for all the visible methods: $\boxed{p \vdash oneOwner(C)}$

$$[\text{ONEOWNER}] \quad \frac{\forall f \in visible_p(C)\ .\ f\ \textit{only occurs once in}\ visible_p(C)}{p \vdash oneOwner(C)}$$

Method typing: $\boxed{p;\Delta;\Gamma \vdash C\ \text{ok} fd}$

$$[\text{T-METHODDEF}] \quad \frac{\begin{array}{c}\_\ C[\![\overrightarrow{\alpha'\ \texttt{extends}\ \_}]\!]\ \_\ \ \texttt{extends}\ \{\overrightarrow{K}\}\ \_ \in p \qquad p \vdash override(f, \{\overrightarrow{M}\}, [\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \overrightarrow{\tau} \to \tau_0) \\ \Delta' = \Delta\ \overrightarrow{\alpha <: N} \qquad p;\Delta' \vdash \overrightarrow{N}\ \text{ok} \qquad p;\Delta' \vdash \overrightarrow{\tau}\ \text{ok} \qquad p;\Delta' \vdash \tau_0\ \text{ok} \\ p;\Delta';\Gamma\ \overrightarrow{x:\tau} \vdash e : \tau' \qquad p;\Delta' \vdash \tau' <: \tau_0\end{array}}{p;\Delta;\Gamma \vdash C\ \text{ok} f[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x:\tau}){:}\tau_0 = e}$$

Figure A.21: Static Semantics of Acyclic Core Fortress with Field Definitions (I)

Field typing: $\boxed{p; \Delta; \Gamma \vdash vd \text{ ok}}$

[T-FIELDDEF] $$\frac{p; \Delta \vdash \tau \text{ ok} \qquad p; \Delta; \Gamma \vdash e : \tau' \qquad p; \Delta \vdash \tau' <: \tau}{p; \Delta; \Gamma \vdash x\texttt{ : }\tau\texttt{ = }e \text{ ok}}$$

Expression typing: $\boxed{p; \Delta; \Gamma \vdash \epsilon : \tau}$

[T-VAR] $$p; \Delta; \Gamma \vdash x : \Gamma(x)$$

[T-SELF] $$p; \Delta; \Gamma \vdash \texttt{self} : \Gamma(\texttt{self})$$

[T-OBJECT] $$\frac{\texttt{object } O[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!]\ (\overrightarrow{\_ : \tau})\ \_\ \texttt{end} \in p \qquad p; \Delta \vdash O[\![\overrightarrow{\tau}]\!] \text{ ok} \qquad p; \Delta; \Gamma \vdash \overrightarrow{\epsilon} : \overrightarrow{\tau''} \qquad p; \Delta \vdash \overrightarrow{\tau''} <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{\tau'}}{p; \Delta; \Gamma \vdash O[\![\overrightarrow{\tau}]\!](\overrightarrow{\epsilon}) : O[\![\overrightarrow{\tau}]\!]}$$

[T-INT-OBJECT] $$\frac{\texttt{object } O[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!](\overrightarrow{x' : \tau'})\ \_\ \overrightarrow{x'' : \tau'' = \_}\ \_\ \texttt{end} \in p \qquad \overrightarrow{x^1}\overrightarrow{x^2} = \overrightarrow{x'}\overrightarrow{x''} \qquad \overrightarrow{\epsilon^1}\overrightarrow{\epsilon^2} = \overrightarrow{\epsilon'}\overrightarrow{\epsilon''} \qquad p; \Delta; \Gamma \vdash O[\![\overrightarrow{\tau}]\!](\overrightarrow{\epsilon'}) : O[\![\overrightarrow{\tau}]\!] \qquad p; \Delta; \Gamma\ \overrightarrow{x' : \tau'}\ \overrightarrow{x'' : \tau''} \vdash \overrightarrow{\epsilon''} : \overrightarrow{\tau'''} \qquad p; \Delta \vdash [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{\tau'''} <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{\tau''}}{p; \Delta; \Gamma \vdash O[\![\overrightarrow{\tau}]\!]\{\overrightarrow{x^1 \mapsto \epsilon^1};\ \overrightarrow{x^2 \mapsto \epsilon^2}\} : O[\![\overrightarrow{\tau}]\!]}$$

[T-FIELD] $$\frac{p; \Delta; \Gamma \vdash \epsilon : \tau_0 \qquad \mathit{bound}_\Delta(\tau_0) = O[\![\overrightarrow{\tau^o}]\!] \qquad \texttt{object } O[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!](\overrightarrow{x' : \tau'})\ \_\ \overrightarrow{x'' : \tau'' = \_}\ \_\ \texttt{end} \in p \qquad \overrightarrow{x} = \overrightarrow{x'}\overrightarrow{x''} \qquad \overrightarrow{\tau} = \overrightarrow{\tau'}\overrightarrow{\tau''}}{p; \Delta; \Gamma \vdash \epsilon\texttt{.}x_i : [\overrightarrow{\tau^o}/\overrightarrow{\alpha}]\tau_i}$$

[T-METHOD] $$\frac{p; \Delta; \Gamma \vdash \epsilon : \tau_0 \qquad \mathit{mtype}_{p;\Delta}(f, \mathit{bound}_\Delta(\tau_0)) = \{[\![\overrightarrow{\alpha \texttt{ extends } N}]\!]\ \overrightarrow{\tau'} \to \tau_0'\} \qquad p; \Delta \vdash \overrightarrow{\tau} \text{ ok} \qquad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{N} \qquad p; \Delta; \Gamma \vdash \overrightarrow{\epsilon} : \overrightarrow{\tau''} \qquad p; \Delta \vdash \overrightarrow{\tau''} <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{\tau'}}{p; \Delta; \Gamma \vdash \epsilon.f[\![\overrightarrow{\tau}]\!](\overrightarrow{\epsilon}) : [\overrightarrow{\tau}/\overrightarrow{\alpha}]\tau_0'}$$

Subtyping: $\boxed{p; \Delta \vdash \tau <: \tau}$

[S-OBJ] $$p; \Delta \vdash \tau <: \texttt{Object}$$

[S-REFL] $$p; \Delta \vdash \tau <: \tau$$

[S-TRANS] $$\frac{p; \Delta \vdash \tau_1 <: \tau_2 \qquad p; \Delta \vdash \tau_2 <: \tau_3}{p; \Delta \vdash \tau_1 <: \tau_3}$$

[S-VAR] $$p; \Delta \vdash \alpha <: \Delta(\alpha)$$

[S-TAPP] $$\frac{\_\ C[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!]\ \_\ \texttt{extends } \{\overrightarrow{N}\}\ \_ \in p}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!] <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i}$$

Figure A.22: Static Semantics of Acyclic Core Fortress with Field Definitions (II)

Well-formed types: $\boxed{p; \Delta \vdash \tau \text{ ok}}$

[W-OBJ] $$p; \Delta \vdash \texttt{Object} \text{ ok}$$

[W-VAR] $$\frac{\alpha \in \mathit{dom}(\Delta)}{p; \Delta \vdash \alpha \text{ ok}}$$

[W-TAPP] $$\frac{\_\ C[\![\overrightarrow{\alpha \texttt{ extends } N}]\!]\ \_ \in p \qquad p; \Delta \vdash \overrightarrow{\tau} \text{ ok} \qquad p; \Delta \vdash \overrightarrow{\tau} <: [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{N}}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!] \text{ ok}}$$

Method overriding: $\boxed{p \vdash \mathit{override}(f, \{\overrightarrow{N}\}, [\![\overrightarrow{\alpha \texttt{ extends } N}]\!]\ \overrightarrow{\tau} \to \tau)}$

[OVERRIDE] $$\frac{\bigcup_{L_i \in \{\overrightarrow{L}\}} \mathit{mtype}_{p;\Delta}(f, L_i) = \{[\![\overrightarrow{\beta \texttt{ extends } M}]\!]\ \overrightarrow{\tau'} \to \tau_0'\} \quad \overrightarrow{N} = [\overrightarrow{\alpha}/\overrightarrow{\beta}]\overrightarrow{M} \quad \overrightarrow{\tau} = [\overrightarrow{\alpha}/\overrightarrow{\beta}]\overrightarrow{\tau'} \quad p; \Delta\ \overrightarrow{\alpha <: \{N\}} \vdash \tau_0 <: [\overrightarrow{\alpha}/\overrightarrow{\beta}]\tau_0'}{p \vdash \mathit{override}(f, \{\overrightarrow{L}\}, [\![\overrightarrow{\alpha \texttt{ extends } N}]\!]\ \overrightarrow{\tau} \to \tau_0)}$$

Method type lookup: $\boxed{\mathit{mtype}_{p;\Delta}(f, \tau) = \{[\![\overrightarrow{\alpha \texttt{ extends } N}]\!]\ \overrightarrow{\tau} \to \tau\}}$

[MT-SELF] $$\frac{\_\ C[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!]\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f[\![\overrightarrow{\beta \texttt{ extends } M}]\!](\overrightarrow{\_:\tau}):\tau_0' = \_ \in \{\overrightarrow{fd}\}}{\mathit{mtype}_{p;\Delta}(f, C[\![\overrightarrow{\tau}]\!]) = \{[\overrightarrow{\tau}/\overrightarrow{\alpha}][\![\overrightarrow{\beta \texttt{ extends } M}]\!]\ \overrightarrow{\tau'} \to \tau_0'\}}$$

[MT-SUPER] $$\frac{\_\ C[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{K}\}\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f \notin \{\overrightarrow{\mathit{Fname}(fd)}\}}{\mathit{mtype}_{p;\Delta}(f, C[\![\overrightarrow{\tau}]\!]) = \bigcup_{N_i \in \{\overrightarrow{N}\}} \mathit{mtype}_{p;\Delta}(f, [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i)}$$

[MT-OBJ] $$\mathit{mtype}_{p;\Delta}(f, \texttt{Object}) = \emptyset$$

Auxiliary functions for methods: $\boxed{\mathit{defined}_p \text{ / } \mathit{inherited}_p \text{ / } \mathit{visible}_p(C) = \{\overrightarrow{f}\}}$

$$\mathit{defined}_p(C) = \{\overrightarrow{\mathit{Fname}(fd)}\} \quad \text{where } \_\ C\ \_\ \overrightarrow{fd}\ \_ \in p$$

$$\mathit{inherited}_p(C) = \biguplus_{N_i \in \{\overrightarrow{N}\}} \{f_i \mid f_i \in \mathit{visible}_p(N_i), f_i \notin \mathit{defined}_p(C)\} \quad \text{where } \_\ C[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_ \in p$$

$$\mathit{visible}_p(C) = \mathit{defined}_p(C) \uplus \mathit{inherited}_p(C)$$

Bound of type: $\boxed{\mathit{bound}_\Delta(\tau) = \tau}$

$$\begin{array}{lcl} \mathit{bound}_\Delta(\alpha) & = & \Delta(\alpha) \\ \mathit{bound}_\Delta(N) & = & N \\ \mathit{bound}_\Delta(O[\![\overrightarrow{\tau}]\!]) & = & O[\![\overrightarrow{\tau}]\!] \end{array}$$

Figure A.23: Static Semantics of Acyclic Core Fortress with Field Definitions (III)

Appendix B

Overloaded Functional Declarations

As mentioned in Chapter 33, this appendix proves that the restrictions discussed in Chapter 33 guarantee no undefined nor ambiguous call at run time.

B.1 Proof of Coercion Resolution for Functions

This section proves that the restrictions discussed in the previous sections guarantee the static resolution of coercion (described in Section 17.5) is well defined for functions (the case for methods is analogous).

Consider a static function call $f(A)$ at some program point Z and its corresponding dynamic function call $f(X)$. Let Σ be the set of parameter types of function declarations of f that are visible at Z and applicable to the static call $f(A)$. Let Σ' be the set of parameter types of function declarations of f that are visible at Z and applicable with coercion to the static call $f(A)$. Moreover, let σ' be the subset of Σ' for which no type in Σ' is more specific:

$$\sigma' \quad = \quad \{S \in \Sigma' \quad | \quad \neg\exists S' \in \Sigma' : S' \lhd S \quad \}.$$

We prove the following:

$$|\Sigma| = 0 \text{ and } |\Sigma'| \neq 0 \text{ imply } |\sigma'| = 1.$$

Informally, if no declaration is applicable to a static call but there is a declaration that is applicable with coercion then there exists a single most specific declaration that is applicable with coercion to the static call.

Lemma 1. *Given an acyclic, irreflexive binary relation R on a set S, and a finite nonempty subset A of S, the set $\{a \in A | \neg\exists a' \in A : (a', a) \in R\}$ is nonempty.*

Proof. Consider the relation R on S as a directed acyclic graph. Let A represent a subgraph. Then the Lemma amounts to proving that there exists a node in the graph represented by A with no edges pointing to it. This follows from the fact that A is finite and the graph is acyclic. □

Lemma 2. *If $|\Sigma'| \geq 1$ then $|\sigma'| \geq 1$.*

Proof. Follows from Lemma 1 where S is the set of all types, A is Σ', and the relation $\lhd$ is acyclic and irreflexive. □

Lemma 3. *If $|\Sigma| = 0$ then $|\sigma'| \leq 1$.*

Proof. For the purpose of contradiction suppose there are two declarations $f(P)$ and $f(Q)$ in σ'. Since both $f(P)$ and $f(Q)$ are applicable with coercion to $f(A)$ and $|\Sigma| = 0$ there must exist a coercion from some type P' to P and a coercion from some type Q' to Q such that $A \preceq P' \cap Q'$. Therefore it is not the case that $P \blacklozenge Q$. By the overloading restrictions, $P \neq Q$ and either $P \triangleleft Q$ or $Q \triangleleft P$ or for all $P' \in \mathcal{S}$ and $Q' \in \mathcal{T}$ either $P' \lozenge Q'$, or there is a declaration $f(P' \cap Q')$ visible at Z. If $P \triangleleft Q$ or $Q \triangleleft P$ then we contradict our assumption. Otherwise, if there exists a declaration $f(P' \cap Q')$ visible at Z then this declaration is applicable to $f(A)$ without coercion. This contradicts $|\Sigma| = 0$. If such a declaration does not exist then it must be the case that $P' \lozenge Q'$. Then both $f(P)$ and $f(Q)$ can not be applicable to the call $f(A)$ which is a contradiction. □

Theorem 1. *If* $|\Sigma| = 0$ *and* $|\Sigma'| \neq 0$ *then* $|\sigma'| = 1$.

Proof. Follows from Lemmas 2 and 3. □

B.2 Proof of Overloading Resolution for Functions

This section proves that the restrictions placed on overloaded function declarations are sufficient to guarantee no undefined nor ambiguous call at run time (the case for methods is analogous).

Consider a static function call $f(A)$ at some program point Z and its corresponding dynamic function call $f(X)$. Let Δ be the set of parameter types of function declarations of f that are visible at Z and applicable to the dynamic call $f(X)$. Let Σ be the set of parameter types of function declarations of f that are visible at Z and applicable to the static call $f(A)$. Moreover, let σ be the subset of Σ for which no type in Σ is more specific and let δ be the subset of Δ for which no type in Δ is more specific:

$$\begin{array}{rcl} \sigma & = & \{S \in \Sigma \quad | \quad \neg\exists S' \in \Sigma : S' \prec S \quad \} \\ \delta & = & \{D \in \Delta \quad | \quad \neg\exists D' \in \Delta : D' \prec D \quad \}. \end{array}$$

Below we prove:

$$|\Sigma| \neq 0 \text{ implies } |\sigma| = 1, and$$
$$|\Sigma| \neq 0 \text{ implies } |\delta| = 1.$$

Informally, if any declaration is applicable to a static call then there exists a single most specific declaration that is applicable to the static call and a single most specific declaration that is applicable to the corresponding dynamic call.

Lemma 4. $\Sigma \subseteq \Delta$.

Proof. Notice that $X \preceq A$ by type soundness. If $f(P)$ is applicable to the call $f(A)$ then $A \preceq P$. Notice that $X \preceq A$ implies $X \preceq P$. Therefore $f(P)$ is applicable to the call $f(X)$. □

Lemma 5. *If* $|\Delta| \geq 1$ *then* $|\delta| \geq 1$. *Also, if* $|\Sigma| \geq 1$ *then* $|\sigma| \geq 1$.

Proof. Follows from Lemma 1 where S is the set of all types, A is Δ and Σ respectively, and the relation $\prec$ is acyclic and irreflexive. □

Lemma 6. *If* $|\Sigma| \geq 1$ *then* $|\delta| \geq 1$.

Proof. Follows from Lemmas 4 and 5. □

Lemma 7. $|\sigma| \leq 1$. *Also,* $|\delta| \leq 1$.

Proof. We prove this for δ, but the case for σ is identical. For the purpose of contradiction suppose there are two declarations $f(P)$ and $f(Q)$ in δ. Since both $f(P)$ and $f(Q)$ are applicable without coercion to the call $f(X)$ we have $X \preceq P \cap Q$. Therefore it is not the case that $P \blacklozenge Q$. By the overloading restrictions, $P \neq Q$ and either $P \lozenge Q$ or there is a declaration $f(P \cap Q)$ visible at Z. Since it cannot be the case that $P \lozenge Q$ there must exist a declaration $f(P \cap Q)$ visible at Z. Since $P \cap Q \preceq P$ and $P \cap Q \preceq Q$ we know $f(P \cap Q)$ is applicable without coercion to the call $f(X)$. Since $P \neq Q$ either $P \cap Q \prec P$ or $P \cap Q \prec Q$. Either case contradicts our assumption. □

Theorem 2. *If* $|\Sigma| \neq 0$ *then* $|\sigma| = 1$. *Also, if* $|\Sigma| \neq 0$ *then* $|\delta| = 1$.

Proof. Follows from Lemmas 5, 6 and 7. □

Theorem 3. *If* $\sigma = \{S\}$ *and* $\delta = \{D\}$ *then* $D \preceq S$.

Proof. If the declaration with parameter type S and the declaration with parameter type D satisfy the Subtype Rule then the theorem is proved. Otherwise, by the definition of σ we have $\sigma \subseteq \Sigma$. Therefore $S \in \Sigma$. By Lemma 4, $S \in \Delta$. Notice that $S, D \in \Delta$ implies $X \preceq S$ and $X \preceq D$. Therefore, $S \not\lozenge D$. By the More Specific Rule for Functions, there must exist a declaration with parameter type $S \cap D$. Because $X \preceq (S \cap D)$, $(S \cap D) \in \Delta$. Notice $(S \cap D) \preceq S$ and $(S \cap D) \preceq D$. By the definition of δ, we have $\neg\exists D' \in \Delta : D' \prec D$. In particular, $(S \cap D) \not\prec D$. Therefore $(S \cap D) = D$. □

Appendix C

Components and APIs

As mentioned in Chapter 22, we formally specify key functionality of the Fortress component system, and illustrate how we can reason about the correctness of the system.

Components One important restriction on components is that no API may be both imported and exported by the same component. Formally, we introduce two functions on components, *imp* and *exp*, that return the imported and exported APIs of the component, respectively. For any component c, $\mathit{imp}(c) \cap \mathit{exp}(c) = \emptyset$. This restriction is required throughout to ground the semantics of operations on components, as discussed in Section 22.7.

APIs Other than its identity, the only relevant characteristic of an API a is the set of APIs that it uses, denoted by $\mathit{uses}(a)$. Because an API a might expose types defined in $\mathit{uses}(a)$, we require that a component that exports a also exports all APIs in $\mathit{uses}(a)$ that it does not import. Formally, the following condition holds on the exported APIs of a component c:

$$a \in \mathit{exp}(c) \wedge a' \in \mathit{uses}(a) \implies a' \in \mathit{imp}(c) \cup \mathit{exp}(c)$$

Link Given a set $C = \{c_1, \ldots, c_k\}$ of components, we define a partial function $\mathit{link}(C)$ that returns the component resulting from c_1 through c_k. If $c = \mathit{link}(C)$, then $\mathit{exp}(c) = \bigcup_{c' \in C} \mathit{exp}(c')$ and $\mathit{imp}(c) = \bigcup_{c' \in C} \mathit{imp}(c') - \mathit{exp}(c)$.

The function *link* is partial because we do not allow arbitrary sets of components to be linked. In particular, Two components cannot be linked if they export the same API.[1] This restriction is made for the sake of simplicity; it allows programmers to link a set of components without having to specify explicitly which constituent exporting an API A provides the implementation exported by the linked component, and which constituent connects to the constituents that import A: only one component exports A, so there is only one choice. Although we lose expressiveness with this design, the user interface to link is vastly simplified, and it is rare that including multiple components that export a given API in a set of linked components is even desirable. We discuss how even such rare cases can be supported in Section 22.8.

For a compound component, in addition to the exported and imported APIs, we want to know what its constituents are. So we introduce another function *cns*, which takes a component and returns the set of its constituents. That is, $\mathit{cns}(\mathit{link}(C)) = C$. It is an invariant of the system that for any compound component C (i.e., $\mathit{cns}(c) \neq \emptyset$), any API imported by any of its constituents is either imported by C or exported by one of its constituents (i.e., $\bigcup_{c' \in \mathit{cns}(c)} \mathit{imp}(c') \subseteq \mathit{imp}(c) \cup \bigcup_{c' \in \mathit{cns}(c)} \mathit{exp}(c')$). This property is crucial for executing components, as we discuss below. A simple component (i.e., one produced directly by compilation) has no constituents (i.e., $\mathit{cns}(c) = \emptyset$).

[1] There is one exception to this rule: the special API $\mathrm{Upgradable}$, which is used during upgrades discussed below.

Upgrade A predicate *upg?* takes two components and indicates whether the first can be upgraded with the second; that is, $\mathit{upg?}(c_\mathrm{t}, c_\mathrm{r})$ returns true if and only if c_t can be upgraded with c_r. This predicate captures both the constraints imposed by a component's $isValidUpgrade$ function and the conditions that guarantee the well-formedness of the result. That is,

$$\begin{aligned}\mathit{upg?}(c_\mathrm{t}, c_\mathrm{r}) \implies\ & c_\mathrm{t}.isValidUpgrade(c_\mathrm{r}) \\ & \wedge\ \mathit{imp}(c_\mathrm{r}) \subseteq \mathit{exp}(c_\mathrm{t}) \cup \mathit{imp}(c_\mathrm{t}) \\ & \wedge\ \mathit{exp}(c_\mathrm{r}) \subset \mathit{exp}(c_\mathrm{t}) \\ & \wedge\ \forall c \in \mathit{cns}(c_\mathrm{t}).(\mathit{exp}(c) \subseteq \mathit{exp}(c_\mathrm{r}) \vee \mathit{exp}(c) \cap \mathit{exp}(c_\mathrm{r}) = \emptyset \vee \mathit{upg?}(c, c_\mathrm{r}))\end{aligned}$$

Visible and Provided We introduce two new functions on components: *vis*, which returns the APIs of a component that have not been hidden; and *prov*, which returns those visible APIs that are exported by some top-level constituent of the component (or all the exported APIs of a simple component); we say these APIs are *provided* by the component. We need to distinguish provided APIs because they can be imported by the top-level constituents of a component, and thus by a replacement component in an upgrade, while other visible APIs cannot be. Thus, for a compound component c, $\mathit{prov}(c) = \mathit{vis}(c) \cap \bigcup_{c' \in \mathit{cns}(c)} \mathit{exp}(c')$. For a simple component c, $\mathit{prov}(c) = \mathit{vis}(c) = \mathit{exp}(c)$.

Constrain If c is a compound component and $A \subset \mathit{exp}(c)$ is a set of APIs such that $a \in \mathit{exp}(c) \wedge a' \in \mathit{uses}(a) \cap A \implies a \in A$, we define $c' = \mathit{constrain}(c, A)$ such that $\mathit{exp}(c') = \mathit{exp}(c) - A$ and for any component c'', $\mathit{upg?}(c', c'') \iff \mathit{upg?}(c, c'') \wedge \mathit{exp}(c') \not\subseteq \mathit{exp}(c'')$. The *imp*, *vis*, *prov* and *cns* functions all have the same values for c and c'. The extra condition on the upgrade compatibility simply captures the restriction we mentioned above, that a replacement component should not export every API exported by the target.

Hide If c is a compound component and $A \subset \mathit{vis}(c)$ is a set of APIs such that $\mathit{exp}(c) \not\subseteq A$ and $a \in \mathit{vis}(c) \wedge a' \in \mathit{uses}(a) \cap A \implies a \in A$, we define $c' = \mathit{hide}(c, A)$ such that $\mathit{vis}(c') = \mathit{vis}(c) - A$, $\mathit{prov}(c') = \mathit{prov}(c) - A$, $\mathit{exp}(c') = \mathit{exp}(c) - A$, and for any component c'', $\mathit{upg?}(c', c'') \iff \mathit{upg?}(c, c'') \wedge \mathit{exp}(c') \not\subseteq \mathit{exp}(c'') \wedge \mathit{vis}(c'') \subseteq \mathit{vis}(c')$. The additional clause in $\mathit{upg?}(c', c'')$ (compared with that of *constrain*) reflects the hiding of the APIs: we can no longer upgrade APIs that are hidden.

Upgrade The interplay between imported, exported, visible and provided APIs introduces subtleties. In particular, the last of the three conditions imposed for well-formedness of upgrades is modified to state that for any constituent that is not subsumed by a replacement component, either it can be upgraded with the replacement, or its *visible* APIs are disjoint from the APIs exported by the replacement (i.e., it is unaffected by the upgrade). To maintain the invariant that no two constituents export the same API, we need another condition, which was implied by the previous condition when no APIs were constrained or hidden: if the replacement subsumes any constituents of the target, then its exported APIs must exactly match the exported APIs of some subset of the constituents of the target. That is, if $\mathit{upg?}(c_\mathrm{t}, c_\mathrm{r}) \wedge \exists c \in \mathit{cns}(c_\mathrm{t}).\ \mathit{exp}(c) \subseteq \mathit{exp}(c_\mathrm{r})$ then $\mathit{exp}(c_\mathrm{r}) = \bigcup_{c \in C} \mathit{exp}(c)$ for some $C \subset \mathit{cns}(c_\mathrm{t})$. In practice, this restriction is rarely a problem; in most cases, a user wishes to upgrade a target with a new version of a single constituent component, where the APIs exported by the old and new versions are either an exact match, or there are new APIs introduced by the new component that have no implementation in the target.

Appendix D

Rendering of Fortress Identifiers

In order to more closely approximate mathematical notation, Fortress mandates standard rendering for various input elements, particularly for numerals and identifiers, as specified in Section 5.17.

In this Appendix, we describe the detailed rules for rendering an identifier.

If an identifier consists of letters and possibly digits, but no underscores or other connecting punctuation, prime marks, or apostrophes, then the rules are fairly simple:

(a) If the identifier consists of two ASCII capital letters that are the same, possibly followed by digits, then a single capital letter is rendered double-struck, followed by full-sized (not subscripted) digits in roman font.

`QQ`	*is rendered as*	$\mathbb{Q}$	`RR64`	*is rendered as*	$\mathbb{R}64$
`ZZ`	*is rendered as*	$\mathbb{Z}$	`ZZ512`	*is rendered as*	$\mathbb{Z}512$

(b) Otherwise, if the identifier has more than two characters and begins with a capital letter, then it is rendered in roman font. (Such names are typically used as names of types in Fortress. Note that an identifier cannot consist entirely of capital letters, because such a token is considered to be an operator.)

`Integer`	*is rendered as*	Integer	`Matrix`	*is rendered as*	Matrix
`TotalOrder`	*is rendered as*	TotalOrder	`BooleanAlgebra`	*is rendered as*	BooleanAlgebra
`Fred17`	*is rendered as*	Fred17	`R2D2`	*is rendered as*	R2D2

(c) Otherwise, if the identifier consists of one or more letters followed by one or more digits, then the letters are rendered in italic and the digits are rendered as roman subscripts.

`a3`	*is rendered as*	a_3	`foo7`	*is rendered as*	foo_7
`M1`	*is rendered as*	M_1	`z128`	*is rendered as*	z_{128}
`Ω13`	*is rendered as*	Ω_{13}	`myFavoriteThings1625`	*is rendered as*	$myFavoriteThings_{1625}$

(d) The following names are always rendered in roman type out of respect for tradition:

sin	cos	tan	cot	sec	csc
sinh	cosh	tanh	coth	sech	csch
arcsin	arccos	arctan	arccot	arcsec	arccsc
arsinh	arcosh	artanh	arcoth	arsech	arcsch
arg	deg	det	exp	inf	sup
lg	ln	log	gcd	max	min

(e) Otherwise the identifier is simply rendered entirely in italic type.

`a`	*is rendered as*	a	`foobar`	*is rendered as*	$foobar$
`length`	*is rendered as*	$length$	`isInstanceOf`	*is rendered as*	$isInstanceOf$
`foo7a`	*is rendered as*	$foo7a$	`l33tsp33k`	*is rendered as*	$l33tsp33k$

If the identifier begins or ends with an underscore, or both, but has no other underscores or other connecting punctuation, or prime marks, or apostrophes:

(f) If the identifier, ignoring its underscores, consists of two ASCII capital letters that are the same, possibly followed by one or more digits, then a single capital letter is rendered in sans-serif (for a leading underscore), script (for a trailing underscore), or italic san-serif (for both a leading and a trailing underscore), and any digits are rendered as roman subscripts.

(g) Otherwise, the identifier without its underscores is rendered in boldface (for a leading underscore), roman (for a trailing underscore), or bold italic (for both a leading and a trailing underscore); except that if the identifier, ignoring its underscores, consists of one or more letters followed by one or more digits, then the digits are rendered as roman subscripts regardless of the underscores.

`m_`	*is rendered as*	m	`s_`	*is rendered as*	s
`km_`	*is rendered as*	km	`kg_`	*is rendered as*	kg
`V_`	*is rendered as*	V	`kW_`	*is rendered as*	kW
`_v`	*is rendered as*	$\mathbf{v}$	`_foo13`	*is rendered as*	$\mathbf{foo}_{13}$

(Roman identifiers are typically used for names of SI dimensional units. See sections 6.1.1 and 6.2.1 of [24] for style questions with respect to dimensions and units.)

These last two rules are actually special cases of the following general rules that apply whenever an identifier contains at least one underscore, other connecting punctuation, prime mark, or apostrophe:

An identifier containing underscores is divided into portions by its underscores; in addition, any apostrophe, prime, or double prime character separates portions and is also itself a portion.

(h) If any portion is empty other than the first or last, then the entire identifier is rendered in italics, underscores and all.

Otherwise, the portions are rendered as follows. The idea is that there is a *principal portion* that may be preceded and/or followed by modifiers, and there may also be a *face portion*:

- If the first portion is not empty, `script`, `fraktur`, `sansserif`, or `monospace`, then the principal portion is the first portion and there is no face portion.
- If the first portion is `script`, `fraktur`, `sansserif`, or `monospace`, then the principal portion is the second portion and the face portion is the first portion.
- If the first portion is empty and the second portion is not `script`, `fraktur`, `sansserif`, or `monospace`, then the principal portion is the second portion and there is no face portion.
- Otherwise the principal portion is the third portion and the face portion is the second portion.

If there is no face portion, the principal portion will be rendered in ordinary italics. However, if the first portion is empty (that is, the identifier begins with a leading underscore), then the principal portion is to be rendered in roman boldface. If the last portion is empty (that is, the identifier ends with a trailing underscore), then the principal portion will be roman rather than italic, or bold italic rather than bold.

If there is a face portion, then that describes an alternate typeface to be used in rendering the principal portion. If there is no face portion, but the principal portion consists of two copies of the same letter, then it is rendered as a single letter in a double-struck face (also known as “blackboard bold”), sans-serif, script, or italic sans-serif font according to whether the first and last portions are (not empty, not empty), (empty, not empty), (not empty, empty), or (empty, empty), respectively. Otherwise, if the first portion is empty (that is, the identifier begins with a leading underscore), then the principal portion is to be rendered in a bold version of the selected face, and if the last portion is empty (that

is, the identifier ends with a trailing underscore), then the principal portion to be rendered in an italic (or bold italic) version of the selected face. The bold and italic modifiers may be used only in combination with certain faces; the following are the allowed combinations:

```
script
bold script
fraktur
bold fraktur
double-struck
sans-serif
bold sans-serif
italic sans-serif
bold italic sans-serif
monospace
```

If a combination can be properly rendered, then the principal portion is rendered but not any preceding portions or underscores. If a combination cannot be properly rendered, then the principal portion and all portions and underscores preceding it are rendered all in italics if possible, and otherwise all in some other default face.

If the principal portion consists of a sequence of letters followed by a sequence of digits, then the letters are rendered in the chosen face and the digits are rendered as roman subscripts. Otherwise the entire principal portion is rendered in the chosen face. The remaining portions (excepting the last, if it is empty) are then processed according to the following rules:

- If a portion is `bar`, then a bar is rendered above what has already been rendered, excluding superscripts and subscripts. For example, `x_bar` is rendered as $\bar{x}$, `x17_bar` is rendered as $\bar{x}_{17}$, `x_bar_bar` is rendered as $\bar{\bar{x}}$, and `foo_bar` is rendered as $\overline{foo}$. (Contrast this last with `foo_baz`, which is rendered as foo_baz.)

- If a portion is `vec`, then a right-pointing arrow is rendered above what has already been rendered, excluding superscripts and subscripts. For example, `v_vec` is rendered as $\vec{v}$, `v17_vec` is rendered as $\vec{v}_{17}$, and `zoom_vec` is rendered as $\overrightarrow{zoom}$.

- If a portion is `hat`, then a hat is rendered above what has already been rendered, excluding superscripts and subscripts. For example, `x17_hat` is rendered as $\hat{x}_{17}$.

- If a portion is `dot`, then a dot is rendered above what has already been rendered, excluding superscripts and subscripts; but if the preceding portion was also `dot`, then the new dot is rendered appropriately relative to the previous dot(s). Up to four dots will be rendered side-by-side rather than vertically. For example, `a_dot` is rendered as $\dot{a}$, `a_dot_dot` is rendered as $\ddot{a}$, `a_dot_dot_dot` is rendered as $\dddot{a}$, `a_dot_dot_dot_dot` is rendered as $\ddddot{a}$. Also, `a_vec_dot` is rendered as $\dot{\vec{v}}$.

- If a portion is `star`, then an asterisk `*` is rendered as a superscript. For example, `a_star` is rendered as a^*, `a_star_star` is rendered as a^{**}, `ZZ_star` is rendered as $\mathbb{Z}^*$.

- If a portion is `splat`, then a number sign `#` is rendered as a superscript. For example, `QQ_splat` is rendered as $\mathbb{Q}^\#$.

- If a portion is `prime`, then a prime mark is rendered as a superscript.

- A prime character is treated the same as `prime`, and a double prime character is treated the same as two consecutive `prime` portions. An apostrophe is treated the same as a prime character, but only if all characters following it in the identifier, if any, are also apostrophes. For example, `a'` is rendered as a', `a13'` is rendered as a'_{13}, and `a''` is rendered as a'', but `don't` is rendered as *don't*.

- If a portion is `super` and another portion follows, then that other portion is rendered as a superscript in roman type, and enclosed in parentheses if it is all digits.

- If a portion is `sub` and another portion follows, then that other portion is rendered as a subscript in roman type, and enclosed in parentheses if it is all digits, and preceded by a subscript-separating comma if this portion was immediately preceded by another portion that was rendered as a subscript.
- If a portion consists entirely of capital letters and would, if considered by itself as an identifier, be the name of a non-letter Unicode character that would be subject to replacement by preprocessing, then that Unicode character is rendered as a subscript. For example, `id_OPLUS` is rendered as $id_{\oplus}$, `ZZ_GT` is rendered as $\mathbb{Z}_{>}$, and `QQ_star_LE` is rendered as $\mathbb{Q}^*_{\leq}$.
- If the portion is the last portion, and the principal portion was a single letter (or two letters indicating a double-struck letter), and none of the preceding rules in this list applies, it is rendered as a subscript in roman type. For example, `T_min` is rendered as T_{min}. Note that `T_MAX` is rendered simply as `T_MAX`—because all its letters are capital letters, it is considered to be an operator—but `T_sub_MAX` is rendered as T_{MAX}.
- Otherwise, this portion and all succeeding portions are rendered in italics, along with any underscores that appear adjacent to any of them.

Examples:

`M`	*is rendered as*	M	`_M`	*is rendered as*	$\mathbf{M}$
`v_vec`	*is rendered as*	$\vec{v}$	`_v_vec`	*is rendered as*	$\vec{\mathbf{v}}$
`v1`	*is rendered as*	v_1	`v_x`	*is rendered as*	v_{x}
`_v1`	*is rendered as*	$\mathbf{v}_1$	`_v_x`	*is rendered as*	$\mathbf{v}_{\mathrm{x}}$
`a_dot`	*is rendered as*	$\dot{a}$	`a_dot_dot`	*is rendered as*	$\ddot{a}$
`a_dot_dot_dot`	*is rendered as*	$\dddot{a}$	`a_dot_dot_dot_dot`	*is rendered as*	$\ddddot{a}$
`a_dot_dot_dot_dot_dot`	*is rendered as*	$\dddddot{a}$	`p13'`	*is rendered as*	p'_{13}
`p'`	*is rendered as*	p'	`p_prime`	*is rendered as*	p'
`T_min`	*is rendered as*	T_{min}	`T_max`	*is rendered as*	T_{max}
`foo_bar`	*is rendered as*	$\overline{foo}$	`foo_baz`	*is rendered as*	foo_baz

In this way, through the use of underscore characters and annotation portions delimited by underscores, the programmer can exercise considerable typographical control over the rendering of variable names; but if no underscores are used, the rendering rules are quite simple.

Appendix E

Support for Unicode Input in ASCII

As mentioned in Chapter 5, to facilitate the writing of Fortress programs using legacy, ASCII-based tools, Fortress programs are subjected to two preprocessing steps. These steps are described in detail in this appendix.

E.1 Word Pasting across Line Breaks

Consider every line terminator in the program (processing them from left to right) such that the following conditions are all true:

- the last non-whitespace character before the line terminator is an ampersand (`&`);
- the first non-whitespace character after the line terminator is an ampersand (`&`);
- a word character immediately precedes the first ampersand; and
- a word character immediately follows the second ampersand.

Then all the characters from the first ampersand to the second ampersand are removed from the program, including the two ampersands. (Note that all the removed characters other than the two ampersands must be whitespace characters.) (The purpose of this is to allow very long identifier names and numeric tokens to be split across line boundaries.) For example:

```
supercalifragilisticexpiali&
   &docious = 0.1428571428571428 57&
   &142857 TIMES &
       GREEK_SMALL_LETTER_&
   &UPSILON_WITH_DIALYTICA_AND_TONOS
```

becomes

```
supercalifragilisticexpialidocious = 0.1428571428571428 57&
   &142857 TIMES &
       GREEK_SMALL_LETTER_&
   &UPSILON_WITH_DIALYTICA_AND_TONOS
```

becomes

```
supercalifragilisticexpialidocious = 0.14285714285714285714285714 TIMES &
       GREEK_SMALL_LETTER_&
   &UPSILON_WITH_DIALYTICA_AND_TONOS
```

becomes

```
supercalifragilisticexpialidocious = 0.14285714285714285714285714 TIMES &
       GREEK_SMALL_LETTER_UPSILON_WITH_DIALYTICA_AND_TONOS
```

E.2 Preprocessing of Names of Unicode Characters

After a program encoded as a sequence of ASCII characters has been processed for word pasting across line breaks as described in the previous section, this step converts restricted words into corresponding Unicode characters. It also converts some other characters, as discussed below.

First the program is analyzed to determine the boundaries of string literals and comments as follows: There are three modes of processing: outside any comment or string literal, inside a string literal and inside a comment. Within a comment, we also keep track of "nesting depth" (this is 0 when not within a comment). All processing proceeds from left to right. Outside any comment or string literal, encountering an unescaped string literal delimiter changes processing to the mode for within a string literal (however, it is a static error if the string literal delimiter is the right double quotation mark), and encountering the opening comment delimiter "`*(`" changes processing to the mode for within a comment, incrementing the nesting depth (to 1). All other characters are ignored, except to note they are outside any comment or string literal. Within a string literal, all characters, including comment delimiters, are ignored (except to note that they are within a string literal) except an unescaped string literal delimiter, which switches processing back to the mode for outside any comment or string literal. Inside a comment, all characters, including unescaped string delimiters, are ignored (except to note that they are within a comment) other than the two-character opening and closing comment delimiters "`(*`" and "`*)`". Whenever the opening comment delimiter is encountered, the nesting depth is incremented, and each time the closing comment delimiter is encountered, the nesting depth is decremented, until it becomes 0. At that point, processing is changed again to the mode for outside any comment or string literal. This step partitions the characters into those within string literals (including the string literal delimiters) and those not within string literals. Note that character literal delimiters are ignored in this step. Thus, we require string literal delimiters to be escaped within character literals.

The characters outside of string literals are partitioned into contiguous subsequences formed by the restricted words, and all the characters between the restricted words and string literals separated by whitespace. That is, no subsequence considered has any whitespace characters or ampersands (ampersands being part of whitespace). Each subsequence is considered separately.

For a restricted word, the general rule is that we try to replace the restricted word with a single Unicode character that it "names". But we never do the replacement if the character is a printable ASCII character, a control character, or a left or right double quotation mark (i.e., characters with code points below U+009F, inclusive, or with code point U+201C or U+201D). We call such characters *protected* characters. Protecting the backslash and double quotation mark characters is necessary to maintain the boundaries for string literals, and protecting the printable ASCII characters ensures that the ASCII conversion process is idempotent. Protecting the control characters makes sense because most of them are forbidden from valid Fortress programs, and those that aren't are available directly in ASCII.

There are four sources for determining whether a restricted word is a "name" for a Unicode character. Because these sources overlap in some cases, and not necessarily in compatible ways, the order in which we try these names is important.

First, Fortress explicitly provides short ASCII names for many characters, especially ones that programmers might be most commonly want. For operators, these names are given in Appendix F. For example, here are some common ones:

LE	*becomes*	$\leq$	GE	*becomes*	$\geq$	NE	*becomes*	$\neq$
BY	*becomes*	$\times$	TIMES	*becomes*	$\times$	CROSS	*becomes*	$\times$
DOT	*becomes*	$\cdot$	PRODUCT	*becomes*	$\prod$	SUM	*becomes*	$\sum$
CUP	*becomes*	$\cup$	CAP	*becomes*	$\cap$	SUBSET	*becomes*	$\subset$
EMPTYSET	*becomes*	$\emptyset$	AND	*becomes*	$\wedge$	OR	*becomes*	$\vee$

Note that some characters have more than one short name. Also, some non-operator characters also have short names, particularly, the Greek letters and the special letters:

ALPHA	*becomes*	A	alpha	*becomes*	α
BETA	*becomes*	B	beta	*becomes*	β
GAMMA	*becomes*	Γ	gamma	*becomes*	γ
DELTA	*becomes*	Δ	delta	*becomes*	δ
EPSILON	*becomes*	E	epsilon	*becomes*	ϵ
ZETA	*becomes*	Z	zeta	*becomes*	ζ
ETA	*becomes*	H	eta	*becomes*	η
THETA	*becomes*	Θ	theta	*becomes*	θ
IOTA	*becomes*	I	iota	*becomes*	ι
KAPPA	*becomes*	K	kappa	*becomes*	κ
LAMBDA	*becomes*	Λ	lambda	*becomes*	λ
MU	*becomes*	M	mu	*becomes*	μ
NU	*becomes*	N	nu	*becomes*	ν
XI	*becomes*	Ξ	xi	*becomes*	ξ
OMICRON	*becomes*	O	omicron	*becomes*	o
PI	*becomes*	Π	pi	*becomes*	π
RHO	*becomes*	P	rho	*becomes*	ρ
SIGMA	*becomes*	Σ	sigma	*becomes*	σ
TAU	*becomes*	T	tau	*becomes*	τ
UPSILON	*becomes*	Υ	upsilon	*becomes*	υ
PHI	*becomes*	Φ	phi	*becomes*	ϕ
CHI	*becomes*	X	chi	*becomes*	χ
PSI	*becomes*	Ψ	psi	*becomes*	ψ
OMEGA	*becomes*	Ω	omega	*becomes*	ω
BOTTOM	*becomes*	$\perp$	TOP	*becomes*	$\top$
INF	*becomes*	∞			

A careful reader will note that Appendix F also gives the following short names for printable ASCII characters:

LT	*becomes*	$<$	GT	*becomes*	$>$	EQ	*becomes*	$=$

These names provide a certain level of compatibility with Fortran. However, they are only replaced by the corresponding character only when they are delimited by whitespace characters (note, not ampersands) or the beginning or end of the program. Thus, they cannot participate in further conversion.

The second source is the official Unicode 5.0 names, as specified by the Unicode Standard. However, recall that restricted words consist of letters, digits and underscores only, while Unicode names may include hyphens and spaces. Thus, we replace a restricted word if it is the Unicode 5.0 name of a character with hyphens and spaces replaced by underscores. For any Unicode character other than the control characters, there is a unique official Unicode 5.0 name not shared by any other Unicode character. Since control characters are protected characters, they do not present a problem in this regard. The third source is alternative names for characters specified by the Unicode Standard, again we use the names with underscores in place of hyphens and spaces. With this source, however, some names designate more than one character. In this case, we replace the restricted word with the character with a smallest code point, unless that character is a protected character (in which case we replace the restricted word with the appropriate unprotected character with the smallest code point, if any). Fourth, we consider the official Unicode 5.0 names and

any alternative names, with underscores in place of hyphens and spaces, where any of the following substrings may be omitted:

```
"LETTER_"
"DIGIT_"
"RADICAL_"
"NUMERAL_"
```

If there are multiple such substrings in a given name, any combination of them may be omitted. Again, if this process yields multiple characters as possible replacement, the unprotected character with the smallest code point is used.

If none of the above replaces the restricted word with a single Unicode character, then the following step is applied, which transforms certain restricted words by replacing just parts of them with Unicode characters. If the restricted word begins with the short name (i.e., the name in the table above) of a Greek letter followed by an underscore or a digit, or ends with the short name of a Greek letter that is preceded by an underscore, or contains the short name of a Greek letter with an underscore on each side of it, then the short name of the Greek letter is replaced by the Greek letter itself. In the same manner, the word "micro" is replaced with the Unicode character `MICRO SIGN` μ (U+00B5, which looks just like the Greek lowercase mu μ but is different). A special ad-hoc rule is that if a word-part being thus replaced has an underscore to each side, and the underscore on the right is the last character of the restricted word, then the underscore on the left is removed as the name is replaced; this is done for the sake of the abbreviations of certain dimensional units, so that, for example, `micro_OMEGA_` will be transformed into $\mu\Omega\_$, signifying micro-ohms, and `G_OMEGA_` will be transformed into $\mathrm{G}\Omega\_$, signifying gigaohms.

Here are some other examples:

`alpha`	*becomes*	α	`OMEGA13`	*becomes*	$\Omega 13$
`alpha_hat`	*becomes*	α_hat	`theta_elephant`	*becomes*	$\theta\_elephant$
`OMEGA_`	*becomes*	$\Omega\_$	`_XI`	*becomes*	$\_\Xi$

For the sequences of characters other than restricted words, each is converted from left to right, with the longest possible substring being converted at once, with one exception: The sequence "`(<`" is not converted if it is immediately followed by any of the following characters: '`<`', '`|`', '`/`', '`\`', '`*`', or '`.`'. That is, with this one exception, the longest shorthand begining from the first character, if any, is converted first. Then, the longest shorthand beginning from the second character of the string after replacement is converted, and so on. Here are the ASCII shorthands for some of the characters we expect to be most frequently used:

`[\`	*becomes*	$[\![$	`\]`	*becomes*	$]\!]$
`->`	*becomes*	$\rightarrow$	`=>`	*becomes*	$\Rightarrow$
`~>`	*becomes*	$\rightsquigarrow$	`|->`	*becomes*	$\mapsto$
`>=`	*becomes*	$\geq$	`<=`	*becomes*	$\leq$
`=/=`	*becomes*	$\neq$			

Although the characters with string literals are generally not subject to this step of ASCII conversion, They are if they are part of a restricted-word escape sequence or a quoted-character escape sequence. See Section 5.10 for details.

Finally, if an ampersand is adjacent to a sequence of characters that is changed by this step of ASCII conversion (even if the sequence was only partly changed, as long as the character adjacent to the ampersand is changed), or to two such names, one on either side, the ampersand is removed after the transformation, *unless* the ampersand is the first or last non-whitespace character on the line.

The process is not iterative. It behaves as if all names are located in the program text, *then* all the names are replaced or transformed as described above, *then* ampersands that had been adjacent to replaced or transformed names are removed. However, because we never replace sequences this step is idempotent, so applying the process again won't change the string again.

Here is a simple example. The expression:

```
(GREEK_SMALL_LETTER_PHI GREEK_SMALL_LETTER_PSI +
GREEK_SMALL_LETTER_OMEGA GREEK_SMALL_LETTER_LAMBDA)
```

is converted to:

$(\phi\ \psi\ +\ \omega\ \lambda)$

where there are four identifiers in all. To get two identifiers, each consisting of two Greek letters, one may write

```
(GREEK_SMALL_LETTER_PHI&GREEK_SMALL_LETTER_PSI +
GREEK_SMALL_LETTER_OMEGA&GREEK_SMALL_LETTER_LAMBDA)
```

which is converted to:

$(\phi\psi\ +\ \omega\lambda)$

A comprehensive list of recognized Unicode operators with their names and abbreviations appears in Appendix F.

Appendix F

Operator Precedence, Chaining, and Enclosure

This appendix contains the detailed rules for which Unicode 5.0 characters may be used as operators, which operators form enclosing pairs, which operators may be chained, and what precedence relationships exist among the various operators. (If no precedence relationship is stated explicitly for any given pair of operators, then there is no precedence relationship between those two operators. Remember that precedence is not transitive in Fortress.)

In each of the character lists below, each line gives the Unicode codepoint, the full Unicode 5.0 name, an indication of what the character looks like in TEX (if possible), then any alternate names or ASCII shorthand for the character.

F.1 Bracket Pairs for Enclosing Operators

Here are the bracket pairs that may be used as enclosing operators. Note that there is one group of four brackets; within that group, either left bracket may be paired with either right bracket to form an enclosing operator.

U+005B	LEFT SQUARE BRACKET	[	[
U+005D	RIGHT SQUARE BRACKET	]	]
U+007B	LEFT CURLY BRACKET	{	{
U+007D	RIGHT CURLY BRACKET	}	}
U+2045	LEFT SQUARE BRACKET WITH QUILL		[./
U+2046	RIGHT SQUARE BRACKET WITH QUILL		/.]
U+2308	LEFT CEILING	⌈	|/
U+2309	RIGHT CEILING	⌉	|
U+230A	LEFT FLOOR	⌊	|\
U+230B	RIGHT FLOOR	⌋	/|
U+27C5	LEFT S-SHAPED BAG DELIMITER		|.\
U+27C6	RIGHT S-SHAPED BAG DELIMITER		/.|
U+27E8	MATHEMATICAL LEFT ANGLE BRACKET	⟨	<|
U+27E9	MATHEMATICAL RIGHT ANGLE BRACKET	⟩	|>
U+27EA	MATHEMATICAL LEFT DOUBLE ANGLE BRACKET	⟪	<<|
U+27EB	MATHEMATICAL RIGHT DOUBLE ANGLE BRACKET	⟫	|>>

U+2983	LEFT WHITE CURLY BRACKET	⦃	{\
U+2984	RIGHT WHITE CURLY BRACKET	⦄	\}
U+2985	LEFT WHITE PARENTHESIS		(\
U+2986	RIGHT WHITE PARENTHESIS		\)
U+2987	Z NOTATION LEFT IMAGE BRACKET		(/
U+2988	Z NOTATION RIGHT IMAGE BRACKET		/)
U+2989	Z NOTATION LEFT BINDING BRACKET		<||
U+298A	Z NOTATION RIGHT BINDING BRACKET		||>
U+298B	LEFT SQUARE BRACKET WITH UNDERBAR		[.\
U+298C	RIGHT SQUARE BRACKET WITH UNDERBAR		\.]
U+298D	LEFT SQUARE BRACKET WITH TICK IN TOP CORNER		[.//
U+298E	RIGHT SQUARE BRACKET WITH TICK IN BOTTOM CORNER		//.]
U+298F	LEFT SQUARE BRACKET WITH TICK IN BOTTOM CORNER		[.\\
U+2990	RIGHT SQUARE BRACKET WITH TICK IN TOP CORNER		\\.]
U+2991	LEFT ANGLE BRACKET WITH DOT		<.|
U+2992	RIGHT ANGLE BRACKET WITH DOT		|.>
U+2993	LEFT ARC LESS-THAN BRACKET		(<
U+2994	RIGHT ARC GREATER-THAN BRACKET		>)
U+2995	DOUBLE LEFT ARC GREATER-THAN BRACKET		((>
U+2996	DOUBLE RIGHT ARC LESS-THAN BRACKET		<))
U+2997	LEFT BLACK TORTOISE SHELL BRACKET		[*/
U+2998	RIGHT BLACK TORTOISE SHELL BRACKET		/*]
U+29D8	LEFT WIGGLY FENCE		[/\/
U+29D9	RIGHT WIGGLY FENCE		/\/]
U+29DA	LEFT DOUBLE WIGGLY FENCE		[/\/\/
U+29DB	RIGHT DOUBLE WIGGLY FENCE		/\/\/]
U+29FC	LEFT-POINTING CURVED ANGLE BRACKET		<|||
U+29FD	RIGHT-POINTING CURVED ANGLE BRACKET		|||>
U+300C	LEFT CORNER BRACKET	⌜	</
U+300D	RIGHT CORNER BRACKET	⌝	\>
U+300E	LEFT WHITE CORNER BRACKET		<</
U+300F	RIGHT WHITE CORNER BRACKET		\>>
U+3010	LEFT BLACK LENTICULAR BRACKET		{*/
U+3011	RIGHT BLACK LENTICULAR BRACKET		/*}
U+3018	LEFT WHITE TORTOISE SHELL BRACKET		[//
U+3014	LEFT TORTOISE SHELL BRACKET		[/
U+3015	RIGHT TORTOISE SHELL BRACKET		/]
U+3019	RIGHT WHITE TORTOISE SHELL BRACKET		//]
U+3016	LEFT WHITE LENTICULAR BRACKET		{/
U+3017	RIGHT WHITE LENTICULAR BRACKET		/}

F.2 Vertical-Line Operators

The following are vertical-line operators:

U+007C	VERTICAL LINE	|	|
U+2016	DOUBLE VERTICAL LINE	‖	||
U+2AF4	TRIPLE VERTICAL BAR BINARY RELATION		|||

F.3 Arithmetic Operators

F.3.1 Multiplication and Division

The following are multiplication operators. Note that ASTERISK OPERATOR is always a multiplication operator; ASTERISK is treated as a synonym for ASTERISK OPERATOR where appropriate, but ASTERISK also has other uses, for example in the ASCII bracket encodings [*/ and /*] and {*/ and /*}.

U+002A	ASTERISK	*	*
U+00B7	MIDDLE DOT	·	DOT
U+00D7	MULTIPLICATION SIGN	×	TIMES BY
U+2217	ASTERISK OPERATOR	∗	
U+228D	MULTISET MULTIPLICATION		
U+2297	CIRCLED TIMES	⊗	OTIMES
U+2299	CIRCLED DOT OPERATOR	⊙	ODOT
U+229B	CIRCLED ASTERISK OPERATOR	⊛	CIRCLEDAST
U+22A0	SQUARED TIMES	⊠	BOXTIMES
U+22A1	SQUARED DOT OPERATOR	⊡	BOXDOT
U+22C5	DOT OPERATOR	⋅	
U+29C6	SQUARED ASTERISK		BOXAST
U+29D4	TIMES WITH LEFT HALF BLACK		
U+29D5	TIMES WITH RIGHT HALF BLACK		
U+2A2F	VECTOR OR CROSS PRODUCT	×	CROSS
U+2A30	MULTIPLICATION SIGN WITH DOT ABOVE		DOTTIMES
U+2A31	MULTIPLICATION SIGN WITH UNDERBAR		
U+2A34	MULTIPLICATION SIGN IN LEFT HALF CIRCLE		
U+2A35	MULTIPLICATION SIGN IN RIGHT HALF CIRCLE		
U+2A36	CIRCLED MULTIPLICATION SIGN WITH CIRCUMFLEX ACCENT		
U+2A37	MULTIPLICATION SIGN IN DOUBLE CIRCLE		
U+2A3B	MULTIPLICATION SIGN IN TRIANGLE		TRITIMES

The following are division operators. Note that DIVISION SLASH is always a division operator; SOLIDUS is treated as a synonym for DIVISION SLASH where appropriate, but SOLIDUS also has other uses, for example in the ASCII bracket encodings (/ and /) and [/ and /] and {/ and /}.

U+002F	SOLIDUS	/	/
U+00F7	DIVISION SIGN	÷	DIV
U+2215	DIVISION SLASH	∕	
U+2298	CIRCLED DIVISION SLASH	⊘	OSLASH
U+29B8	CIRCLED REVERSE SOLIDUS		
U+29BC	CIRCLED ANTICLOCKWISE-ROTATED DIVISION SIGN		
U+29C4	SQUARED RISING DIAGONAL SLASH		BOXSLASH
U+29F5	REVERSE SOLIDUS OPERATOR	⧵	
U+29F8	BIG SOLIDUS	⟋	
U+29F9	BIG REVERSE SOLIDUS	⟍	
U+2A38	CIRCLED DIVISION SIGN		ODIV

U+2AFD DOUBLE SOLIDUS OPERATOR ⫽ //
U+2AFB TRIPLE SOLIDUS BINARY RELATION ///

Note also that `per` is treated as a division operator.

F.3.2 Addition and Subtraction

The following three operators have the same precedence and may be mixed.

U+002B PLUS SIGN + +
U+002D HYPHEN-MINUS − -
U+2212 MINUS SIGN −

They each have lower precedence than any of the following multiplication and division operators:

U+002A ASTERISK * *
U+002F SOLIDUS / /
U+00B7 MIDDLE DOT · DOT
U+00D7 MULTIPLICATION SIGN × TIMES
U+00F7 DIVISION SIGN ÷ DIV
U+2215 DIVISION SLASH ∕
U+2217 ASTERISK OPERATOR ∗
U+22C5 DOT OPERATOR ⋅
U+2A2F VECTOR OR CROSS PRODUCT × CROSS

The following two operators have the same precedence and may be mixed.

U+2214 DOT PLUS ∔ DOTPLUS
U+2238 DOT MINUS ∸ DOTMINUS

They each have lower precedence than this multiplication operator:

U+2A30 MULTIPLICATION SIGN WITH DOT ABOVE DOTTIMES

The following two operators have the same precedence and may be mixed.

U+2A25 PLUS SIGN WITH DOT BELOW
U+2A2A MINUS SIGN WITH DOT BELOW

The following two operators have the same precedence and may be mixed.

U+2A39 PLUS SIGN IN TRIANGLE TRIPLUS
U+2A3A MINUS SIGN IN TRIANGLE TRIMINUS

They each have lower precedence than this multiplication operator:

U+2A3B MULTIPLICATION SIGN IN TRIANGLE TRITIMES

The following two operators have the same precedence and may be mixed.

U+2295 CIRCLED PLUS ⊕ OPLUS
U+2296 CIRCLED MINUS ⊖ OMINUS

They each have lower precedence than any of the following multiplication and division operators:

U+2297 CIRCLED TIMES ⊗ OTIMES
U+2298 CIRCLED DIVISION SLASH ⊘ OSLASH
U+2299 CIRCLED DOT OPERATOR ⊙ ODOT
U+229B CIRCLED ASTERISK OPERATOR ⊛ CIRCLEDAST
U+2A38 CIRCLED DIVISION SIGN ODIV

The following two operators have the same precedence and may be mixed.

U+229E SQUARED PLUS	⊞	BOXPLUS
U+229F SQUARED MINUS	⊟	BOXMINUS

They each have lower precedence than any of these multiplication or division operators:

U+22A0 SQUARED TIMES	⊠	BOXTIMES
U+22A1 SQUARED DOT OPERATOR	⊡	BOXDOT
U+29C4 SQUARED RISING DIAGONAL SLASH		BOXSLASH
U+29C6 SQUARED ASTERISK		BOXAST

These are other miscellaneous addition and subtraction operators:

U+00B1 PLUS-MINUS SIGN	±
U+2213 MINUS-OR-PLUS SIGN	∓
U+2242 MINUS TILDE	
U+2A22 PLUS SIGN WITH SMALL CIRCLE ABOVE	$\overset{\circ}{+}$
U+2A23 PLUS SIGN WITH CIRCUMFLEX ACCENT ABOVE	$\overset{\wedge}{+}$
U+2A24 PLUS SIGN WITH TILDE ABOVE	$\overset{\sim}{+}$
U+2A26 PLUS SIGN WITH TILDE BELOW	$+\!\!\underset{\sim}{}$
U+2A27 PLUS SIGN WITH SUBSCRIPT TWO	$+_2$
U+2A28 PLUS SIGN WITH BLACK TRIANGLE	
U+2A29 MINUS SIGN WITH COMMA ABOVE	$\overset{,}{-}$
U+2A2B MINUS SIGN WITH FALLING DOTS	
U+2A2C MINUS SIGN WITH RISING DOTS	
U+2A2D PLUS SIGN IN LEFT HALF CIRCLE	
U+2A2E PLUS SIGN IN RIGHT HALF CIRCLE	

F.3.3 Miscellaneous Arithmetic Operators

The operators MAX, MIN, REM, MOD, GCD, LCM, CHOOSE, and per, none of which corresponds to a single Unicode character, are considered to be arithmetic operators, having higher precedence than certain relational operators, as described in a later section.

F.3.4 Set Intersection, Union, and Difference

The following are the set intersection operators:

U+2229 INTERSECTION	∩	CAP INTERSECT
U+22D2 DOUBLE INTERSECTION	⋒	CAPCAP
U+2A40 INTERSECTION WITH DOT		
U+2A43 INTERSECTION WITH OVERBAR	⩃	
U+2A44 INTERSECTION WITH LOGICAL AND		
U+2A4B INTERSECTION BESIDE AND JOINED WITH INTERSECTION		
U+2A4D CLOSED INTERSECTION WITH SERIFS		
U+2ADB TRANSVERSAL INTERSECTION		

The following are the set union operators:

U+222A	UNION	∪	CUP UNION
U+228E	MULTISET UNION	⊎	UPLUS
U+22D3	DOUBLE UNION	⋓	CUPCUP
U+2A41	UNION WITH MINUS SIGN		
U+2A42	UNION WITH OVERBAR	⩂	
U+2A45	UNION WITH LOGICAL OR		
U+2A4A	UNION BESIDE AND JOINED WITH UNION		
U+2A4C	CLOSED UNION WITH SERIFS		
U+2A50	CLOSED UNION WITH SERIFS AND SMASH PRODUCT		

They each have lower precedence than any of the set intersection operators.

This is a miscellaneous set operator:

U+2216	SET MINUS	∖	SETMINUS

F.3.5 Square Arithmetic Operators

The following are the square intersection operators:

U+2293	SQUARE CAP	⊓	SQCAP
U+2A4E	DOUBLE SQUARE INTERSECTION		SQCAPCAP

The following are the square union operators:

U+2294	SQUARE CUP	⊔	SQCUP
U+2A4F	DOUBLE SQUARE UNION		SQCUPCUP

They each have lower precedence than either of the square intersection operators.

F.3.6 Curly Arithmetic Operators

The following is the curly intersection operator:

U+22CF	CURLY LOGICAL AND	⋏	CURLYAND

The following is the curly union operator:

U+22CE	CURLY LOGICAL OR	⋎	CURLYOR

It has lower precedence than the curly intersection operator.

F.4 Relational Operators

F.4.1 Equivalence and Inequivalence Operators

Every operator listed in this section has lower precedence than any operator listed in Section F.3.

The following are equivalence operators. They may be chained. Moreover, they may be chained with any other single group of chainable relational operators, as described in later sections.

U+003D EQUALS SIGN = = EQ
U+2243 ASYMPTOTICALLY EQUAL TO ≃ SIMEQ
U+2245 APPROXIMATELY EQUAL TO ≅
U+2246 APPROXIMATELY BUT NOT ACTUALLY EQUAL TO
U+2247 NEITHER APPROXIMATELY NOR ACTUALLY EQUAL TO ≇
U+2248 ALMOST EQUAL TO ≈ APPROX
U+224A ALMOST EQUAL OR EQUAL TO ≊ APPROXEQ
U+224C ALL EQUAL TO
U+224D EQUIVALENT TO ≍
U+224E GEOMETRICALLY EQUIVALENT TO ≎ BUMPEQV
U+2251 GEOMETRICALLY EQUAL TO ≑ DOTEQDOT
U+2252 APPROXIMATELY EQUAL TO OR THE IMAGE OF ≒
U+2253 IMAGE OF OR APPROXIMATELY EQUAL TO ≓
U+2256 RING IN EQUAL TO ≖ EQRING
U+2257 RING EQUAL TO ≗ RINGEQ
U+225B STAR EQUALS
U+225C DELTA EQUAL TO ≜ EQDEL
U+225D EQUAL TO BY DEFINITION EQDEF
U+225F QUESTIONED EQUAL TO
U+2261 IDENTICAL TO ≡ EQV EQUIV
U+2263 STRICTLY EQUIVALENT TO SEQV ===
U+229C CIRCLED EQUALS
U+22CD REVERSED TILDE EQUALS ⋍
U+22D5 EQUAL AND PARALLEL TO
U+29E3 EQUALS SIGN AND SLANTED PARALLEL
U+29E4 EQUALS SIGN AND SLANTED PARALLEL WITH TILDE ABOVE
U+29E5 IDENTICAL TO AND SLANTED PARALLEL
U+2A66 EQUALS SIGN WITH DOT BELOW
U+2A67 IDENTICAL WITH DOT ABOVE
U+2A6C SIMILAR MINUS SIMILAR
U+2A6E EQUALS WITH ASTERISK
U+2A6F ALMOST EQUAL TO WITH CIRCUMFLEX ACCENT
U+2A70 APPROXIMATELY EQUAL OR EQUAL TO
U+2A71 EQUALS SIGN ABOVE PLUS SIGN
U+2A72 PLUS SIGN ABOVE EQUALS SIGN
U+2A73 EQUALS SIGN ABOVE TILDE OPERATOR
U+2A75 TWO CONSECUTIVE EQUALS SIGNS
U+2A76 THREE CONSECUTIVE EQUALS SIGNS
U+2A77 EQUALS SIGN WITH TWO DOTS ABOVE AND TWO DOTS BELOW
U+2A78 EQUIVALENT WITH FOUR DOTS ABOVE
U+2AAE EQUALS SIGN WITH BUMPY ABOVE
U+FE66 SMALL EQUALS SIGN
U+FF1D FULLWIDTH EQUALS SIGN

The following are inequivalence operators. They may not be chained.

U+2244 NOT ASYMPTOTICALLY EQUAL TO ≄ NSIMEQ
U+2249 NOT ALMOST EQUAL TO ≉ NAPPROX
U+2260 NOT EQUAL TO ≠ =/= NE
U+2262 NOT IDENTICAL TO ≢ NEQV
U+226D NOT EQUIVALENT TO ≭

F.4.2 Plain Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Sections F.3.1, F.3.2, and F.3.3.

The following are less-than operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+003C	LESS-THAN SIGN	<	< LT
U+2264	LESS-THAN OR EQUAL TO	≤	<= LE
U+2266	LESS-THAN OVER EQUAL TO	≦	
U+2268	LESS-THAN BUT NOT EQUAL TO	≨	
U+226A	MUCH LESS-THAN	≪	<<
U+2272	LESS-THAN OR EQUIVALENT TO	≲	
U+22D6	LESS-THAN WITH DOT	⋖	DOTLT
U+22D8	VERY MUCH LESS-THAN	⋘	<<<
U+22DC	EQUAL TO OR LESS-THAN		
U+22E6	LESS-THAN BUT NOT EQUIVALENT TO	⋦	
U+29C0	CIRCLED LESS-THAN		
U+2A79	LESS-THAN WITH CIRCLE INSIDE		
U+2A7B	LESS-THAN WITH QUESTION MARK ABOVE		
U+2A7D	LESS-THAN OR SLANTED EQUAL TO		
U+2A7F	LESS-THAN OR SLANTED EQUAL TO WITH DOT INSIDE		
U+2A81	LESS-THAN OR SLANTED EQUAL TO WITH DOT ABOVE		
U+2A83	LESS-THAN OR SLANTED EQUAL TO WITH DOT ABOVE RIGHT		
U+2A85	LESS-THAN OR APPROXIMATE		
U+2A87	LESS-THAN AND SINGLE-LINE NOT EQUAL TO		
U+2A89	LESS-THAN AND NOT APPROXIMATE		
U+2A8D	LESS-THAN ABOVE SIMILAR OR EQUAL		
U+2A95	SLANTED EQUAL TO OR LESS-THAN		
U+2A97	SLANTED EQUAL TO OR LESS-THAN WITH DOT INSIDE		
U+2A99	DOUBLE-LINE EQUAL TO OR LESS-THAN		
U+2A9B	DOUBLE-LINE SLANTED EQUAL TO OR LESS-THAN		
U+2A9D	SIMILAR OR LESS-THAN		
U+2A9F	SIMILAR ABOVE LESS-THAN ABOVE EQUALS SIGN		
U+2AA1	DOUBLE NESTED LESS-THAN		
U+2AA3	DOUBLE NESTED LESS-THAN WITH UNDERBAR		
U+2AA6	LESS-THAN CLOSED BY CURVE		
U+2AA8	LESS-THAN CLOSED BY CURVE ABOVE SLANTED EQUAL		
U+2AF7	TRIPLE NESTED LESS-THAN		
U+2AF9	DOUBLE-LINE SLANTED LESS-THAN OR EQUAL TO		
U+FE64	SMALL LESS-THAN SIGN		
U+FF1C	FULLWIDTH LESS-THAN SIGN		

The following are greater-than operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+003E	GREATER-THAN SIGN	>	> GT
U+2265	GREATER-THAN OR EQUAL TO	≥	>= GE
U+2267	GREATER-THAN OVER EQUAL TO	≧	
U+2269	GREATER-THAN BUT NOT EQUAL TO	≩	
U+226B	MUCH GREATER-THAN	≫	>>
U+2273	GREATER-THAN OR EQUIVALENT TO	≳	
U+22D7	GREATER-THAN WITH DOT	⋗	DOTGT
U+22D9	VERY MUCH GREATER-THAN	⋙	>>>

U+22DD EQUAL TO OR GREATER-THAN
U+22E7 GREATER-THAN BUT NOT EQUIVALENT TO ⋧
U+29C1 CIRCLED GREATER-THAN
U+2A7A GREATER-THAN WITH CIRCLE INSIDE
U+2A7C GREATER-THAN WITH QUESTION MARK ABOVE
U+2A7E GREATER-THAN OR SLANTED EQUAL TO
U+2A80 GREATER-THAN OR SLANTED EQUAL TO WITH DOT INSIDE
U+2A82 GREATER-THAN OR SLANTED EQUAL TO WITH DOT ABOVE
U+2A84 GREATER-THAN OR SLANTED EQUAL TO WITH DOT ABOVE LEFT
U+2A86 GREATER-THAN OR APPROXIMATE
U+2A88 GREATER-THAN AND SINGLE-LINE NOT EQUAL TO
U+2A8A GREATER-THAN AND NOT APPROXIMATE
U+2A8E GREATER-THAN ABOVE SIMILAR OR EQUAL
U+2A96 SLANTED EQUAL TO OR GREATER-THAN
U+2A98 SLANTED EQUAL TO OR GREATER-THAN WITH DOT INSIDE
U+2A9A DOUBLE-LINE EQUAL TO OR GREATER-THAN
U+2A9C DOUBLE-LINE SLANTED EQUAL TO OR GREATER-THAN
U+2A9E SIMILAR OR GREATER-THAN
U+2AA0 SIMILAR ABOVE GREATER-THAN ABOVE EQUALS SIGN
U+2AA2 DOUBLE NESTED GREATER-THAN
U+2AA7 GREATER-THAN CLOSED BY CURVE
U+2AA9 GREATER-THAN CLOSED BY CURVE ABOVE SLANTED EQUAL
U+2AF8 TRIPLE NESTED GREATER-THAN
U+2AFA DOUBLE-LINE SLANTED GREATER-THAN OR EQUAL TO
U+FE65 SMALL GREATER-THAN SIGN
U+FF1E FULLWIDTH GREATER-THAN SIGN

The following are miscellaneous plain comparison operators. They may not be mixed or chained.

U+226E NOT LESS-THAN ≮ NLT
U+226F NOT GREATER-THAN ≯ NGT
U+2270 NEITHER LESS-THAN NOR EQUAL TO ≰ NLE
U+2271 NEITHER GREATER-THAN NOR EQUAL TO ≱ NGE
U+2274 NEITHER LESS-THAN NOR EQUIVALENT TO
U+2275 NEITHER GREATER-THAN NOR EQUIVALENT TO
U+2276 LESS-THAN OR GREATER-THAN ≶
U+2277 GREATER-THAN OR LESS-THAN ≷
U+2278 NEITHER LESS-THAN NOR GREATER-THAN
U+2279 NEITHER GREATER-THAN NOR LESS-THAN
U+22DA LESS-THAN EQUAL TO OR GREATER-THAN ⋚
U+22DB GREATER-THAN EQUAL TO OR LESS-THAN ⋛
U+2A8B LESS-THAN ABOVE DOUBLE-LINE EQUAL ABOVE GREATER-THAN
U+2A8C GREATER-THAN ABOVE DOUBLE-LINE EQUAL ABOVE LESS-THAN
U+2A8F LESS-THAN ABOVE SIMILAR ABOVE GREATER-THAN
U+2A90 GREATER-THAN ABOVE SIMILAR ABOVE LESS-THAN
U+2A91 LESS-THAN ABOVE GREATER-THAN ABOVE DOUBLE-LINE EQUAL
U+2A92 GREATER-THAN ABOVE LESS-THAN ABOVE DOUBLE-LINE EQUAL
U+2A93 LESS-THAN ABOVE SLANTED EQUAL ABOVE GREATER-THAN ABOVE SLANTED EQUAL
U+2A94 GREATER-THAN ABOVE SLANTED EQUAL ABOVE LESS-THAN ABOVE SLANTED EQUAL
U+2AA4 GREATER-THAN OVERLAPPING LESS-THAN
U+2AA5 GREATER-THAN BESIDE LESS-THAN

The following are not really comparison operators, but it is convenient to list them here because they also have lower

precedence than any operator listed in Sections F.3.1, F.3.2, and F.3.3:

U+0023	NUMBER SIGN	#	#
U+003A	COLON	:	:

F.4.3 Set Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section F.3.4.

The following are subset comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+2282	SUBSET OF	⊂	SUBSET
U+2286	SUBSET OF OR EQUAL TO	⊆	SUBSETEQ
U+228A	SUBSET OF WITH NOT EQUAL TO	⊊	SUBSETNEQ
U+22D0	DOUBLE SUBSET	⋐	SUBSUB
U+27C3	OPEN SUBSET		
U+2ABD	SUBSET WITH DOT		
U+2ABF	SUBSET WITH PLUS SIGN BELOW		
U+2AC1	SUBSET WITH MULTIPLICATION SIGN BELOW		
U+2AC3	SUBSET OF OR EQUAL TO WITH DOT ABOVE		
U+2AC5	SUBSET OF ABOVE EQUALS SIGN		
U+2AC7	SUBSET OF ABOVE TILDE OPERATOR		
U+2AC9	SUBSET OF ABOVE ALMOST EQUAL TO		
U+2ACB	SUBSET OF ABOVE NOT EQUAL TO		
U+2ACF	CLOSED SUBSET		
U+2AD1	CLOSED SUBSET OR EQUAL TO		
U+2AD5	SUBSET ABOVE SUBSET		

The following are superset comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+2283	SUPERSET OF	⊃	SUPSET
U+2287	SUPERSET OF OR EQUAL TO	⊇	SUPSETEQ
U+228B	SUPERSET OF WITH NOT EQUAL TO	⊋	SUPSETNEQ
U+22D1	DOUBLE SUPERSET	⋑	SUPSUP
U+27C4	OPEN SUPERSET		
U+2ABE	SUPERSET WITH DOT		
U+2AC0	SUPERSET WITH PLUS SIGN BELOW		
U+2AC2	SUPERSET WITH MULTIPLICATION SIGN BELOW		
U+2AC4	SUPERSET OF OR EQUAL TO WITH DOT ABOVE		
U+2AC6	SUPERSET OF ABOVE EQUALS SIGN		
U+2AC8	SUPERSET OF ABOVE TILDE OPERATOR		
U+2ACA	SUPERSET OF ABOVE ALMOST EQUAL TO		
U+2ACC	SUPERSET OF ABOVE NOT EQUAL TO		
U+2AD0	CLOSED SUPERSET		
U+2AD2	CLOSED SUPERSET OR EQUAL TO		
U+2AD6	SUPERSET ABOVE SUPERSET		

The following are miscellaneous set comparison operators. They may not be mixed or chained.

U+2284	NOT A SUBSET OF	⊄	NSUBSET
U+2285	NOT A SUPERSET OF	⊅	NSUPSET
U+2288	NEITHER A SUBSET OF NOR EQUAL TO	⊈	NSUBSETEQ

U+2289 NEITHER A SUPERSET OF NOR EQUAL TO ⊉ NSUPSETEQ
U+2AD3 SUBSET ABOVE SUPERSET
U+2AD4 SUPERSET ABOVE SUBSET
U+2AD7 SUPERSET BESIDE SUBSET
U+2AD8 SUPERSET BESIDE AND JOINED BY DASH WITH SUBSET

F.4.4 Square Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section F.3.5.

The following are square "image of" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+228F SQUARE IMAGE OF ⊏ SQSUBSET
U+2291 SQUARE IMAGE OF OR EQUAL TO ⊑ SQSUBSETEQ
U+22E4 SQUARE IMAGE OF OR NOT EQUAL TO

The following are square "original of" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+2290 SQUARE ORIGINAL OF ⊐ SQSUPSET
U+2292 SQUARE ORIGINAL OF OR EQUAL TO ⊒ SQSUPSETEQ
U+22E5 SQUARE ORIGINAL OF OR NOT EQUAL TO

The following are miscellaneous square comparison operators. They may not be mixed or chained.

U+22E2 NOT SQUARE IMAGE OF OR EQUAL TO ⋢
U+22E3 NOT SQUARE ORIGINAL OF OR EQUAL TO ⋣

F.4.5 Curly Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section F.3.6.

The following are curly "precedes" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+227A PRECEDES ≺ PREC
U+227C PRECEDES OR EQUAL TO ≼ PRECEQ
U+227E PRECEDES OR EQUIVALENT TO ≾ PRECSIM
U+22B0 PRECEDES UNDER RELATION
U+22DE EQUAL TO OR PRECEDES ⋞ EQPREC
U+22E8 PRECEDES BUT NOT EQUIVALENT TO ⋨ PRECNSIM
U+2AAF PRECEDES ABOVE SINGLE-LINE EQUALS SIGN
U+2AB1 PRECEDES ABOVE SINGLE-LINE NOT EQUAL TO
U+2AB3 PRECEDES ABOVE EQUALS SIGN
U+2AB5 PRECEDES ABOVE NOT EQUAL TO
U+2AB7 PRECEDES ABOVE ALMOST EQUAL TO
U+2AB9 PRECEDES ABOVE NOT ALMOST EQUAL TO
U+2ABB DOUBLE PRECEDES

The following are curly "succeeds" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+227B SUCCEEDS ≻ SUCC
U+227D SUCCEEDS OR EQUAL TO ≽ SUCCEQ
U+227F SUCCEEDS OR EQUIVALENT TO ≿ SUCCSIM
U+22B1 SUCCEEDS UNDER RELATION
U+22DF EQUAL TO OR SUCCEEDS ⋟ EQSUCC
U+22E9 SUCCEEDS BUT NOT EQUIVALENT TO ⋩ SUCCNSIM
U+2AB0 SUCCEEDS ABOVE SINGLE-LINE EQUALS SIGN
U+2AB2 SUCCEEDS ABOVE SINGLE-LINE NOT EQUAL TO
U+2AB4 SUCCEEDS ABOVE EQUALS SIGN
U+2AB6 SUCCEEDS ABOVE NOT EQUAL TO
U+2AB8 SUCCEEDS ABOVE ALMOST EQUAL TO
U+2ABA SUCCEEDS ABOVE NOT ALMOST EQUAL TO
U+2ABC DOUBLE SUCCEEDS

The following are miscellaneous curly comparison operators. They may not be mixed or chained.

U+2280 DOES NOT PRECEDE ⊀ NPREC
U+2281 DOES NOT SUCCEED ⊁ NSUCC
U+22E0 DOES NOT PRECEDE OR EQUAL ⋠
U+22E1 DOES NOT SUCCEED OR EQUAL ⋡

F.4.6 Triangular Comparison Operators

The following are triangular "subgroup" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+22B2 NORMAL SUBGROUP OF ◁
U+22B4 NORMAL SUBGROUP OF OR EQUAL TO ⊴

The following are triangular "contains as subgroup" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+22B3 CONTAINS AS NORMAL SUBGROUP ▷
U+22B5 CONTAINS AS NORMAL SUBGROUP OR EQUAL TO ⊵

The following are miscellaneous triangular comparison operators. They may not be mixed or chained.

U+22EA NOT NORMAL SUBGROUP OF ⋪
U+22EB DOES NOT CONTAIN AS NORMAL SUBGROUP ⋫
U+22EC NOT NORMAL SUBGROUP OF OR EQUAL TO ⋬
U+22ED DOES NOT CONTAIN AS NORMAL SUBGROUP OR EQUAL ⋭

F.4.7 Chickenfoot Comparison Operators

The following are chickenfoot "smaller than" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+2AAA SMALLER THAN ⩹ SMALLER
U+2AAC SMALLER THAN OR EQUAL TO ⩽ SMALLEREQ

The following are chickenfoot "larger than" comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section F.4.1).

U+2AAB LARGER THAN ⩺ LARGER
U+2AAD LARGER THAN OR EQUAL TO ⩾ LARGEREQ

F.4.8 Miscellaneous Relational Operators

The following operators are considered to be relational operators, having higher precedence than certain boolean operators, as described in a later section.

U+2208	ELEMENT OF	∈	IN
U+2209	NOT AN ELEMENT OF	∉	NOTIN
U+220A	SMALL ELEMENT OF	∊	
U+220B	CONTAINS AS MEMBER	∋	CONTAINS
U+220C	DOES NOT CONTAIN AS MEMBER	∌	
U+220D	SMALL CONTAINS AS MEMBER	∍	
U+22F2	ELEMENT OF WITH LONG HORIZONTAL STROKE		
U+22F3	ELEMENT OF WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22F4	SMALL ELEMENT OF WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22F5	ELEMENT OF WITH DOT ABOVE	∈̇	
U+22F6	ELEMENT OF WITH OVERBAR	∈̅	
U+22F7	SMALL ELEMENT OF WITH OVERBAR	∊̅	
U+22F8	ELEMENT OF WITH UNDERBAR	∈̲	
U+22F9	ELEMENT OF WITH TWO HORIZONTAL STROKES		
U+22FA	CONTAINS WITH LONG HORIZONTAL STROKE		
U+22FB	CONTAINS WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22FC	SMALL CONTAINS WITH VERTICAL BAR AT END OF HORIZONTAL STROKE		
U+22FD	CONTAINS WITH OVERBAR	∋̅	
U+22FE	SMALL CONTAINS WITH OVERBAR	∍̅	
U+22FF	Z NOTATION BAG MEMBERSHIP		

F.5 Boolean Operators

Every operator listed in this section has lower precedence than any operator listed in Section F.4.

The following are the Boolean conjunction operators:

U+2227	LOGICAL AND	∧	AND
U+27D1	AND WITH DOT		
U+2A51	LOGICAL AND WITH DOT ABOVE	⩑	
U+2A53	DOUBLE LOGICAL AND		
U+2A55	TWO INTERSECTING LOGICAL AND	⩕	
U+2A5A	LOGICAL AND WITH MIDDLE STEM		
U+2A5C	LOGICAL AND WITH HORIZONTAL DASH		
U+2A5E	LOGICAL AND WITH DOUBLE OVERBAR		
U+2A60	LOGICAL AND WITH DOUBLE UNDERBAR		

The following are the Boolean disjunction operators:

U+2228	LOGICAL OR	∨	OR
U+2A52	LOGICAL OR WITH DOT ABOVE	⩒	
U+2A54	DOUBLE LOGICAL OR		
U+2A56	TWO INTERSECTING LOGICAL OR	⩖	
U+2A5B	LOGICAL OR WITH MIDDLE STEM		
U+2A5D	LOGICAL OR WITH HORIZONTAL DASH		
U+2A62	LOGICAL OR WITH DOUBLE OVERBAR		
U+2A63	LOGICAL OR WITH DOUBLE UNDERBAR		

They each have lower precedence than any of the Boolean conjunction operators.

The following are miscellaneous Boolean operators:

U+2192 RIGHTWARDS ARROW	→	-> IMPLIES
U+2194 LEFT RIGHT ARROW	↔	<-> IFF
U+22BB XOR	⊻	
U+22BC NAND	⊼	
U+22BD NOR	⊽	

F.6 Other Operators

Each of the following operators has no defined precedence relationships to any of the other operators listed in this appendix.

U+0021 EXCLAMATION MARK	!	!
U+0024 DOLLAR SIGN	$	$
U+0025 PERCENT SIGN	%	%
U+003F QUESTION MARK	?	?
U+0040 COMMERCIAL AT	@	@
U+005E CIRCUMFLEX ACCENT	^	^
U+007E TILDE	~	~
U+00A1 INVERTED EXCLAMATION MARK	¡	
U+00A2 CENT SIGN		CENTS
U+00A3 POUND SIGN		
U+00A4 CURRENCY SIGN		
U+00A5 YEN SIGN		
U+00A6 BROKEN BAR		
U+00AC NOT SIGN	¬	NOT
U+00B0 DEGREE SIGN	°	DEGREES
U+00BF INVERTED QUESTION MARK	¿	
U+203C DOUBLE EXCLAMATION MARK	!!	!!
U+2190 LEFTWARDS ARROW	←	<-
U+2191 UPWARDS ARROW	↑	UPARROW
U+2193 DOWNWARDS ARROW	↓	DOWNARROW
U+2195 UP DOWN ARROW	↕	UPDOWNARROW
U+2196 NORTH WEST ARROW	↖	NWARROW
U+2197 NORTH EAST ARROW	↗	NEARROW
U+2198 SOUTH EAST ARROW	↘	SEARROW
U+2199 SOUTH WEST ARROW	↙	SWARROW
U+219A LEFTWARDS ARROW WITH STROKE	↚	<-/-
U+219B RIGHTWARDS ARROW WITH STROKE	↛	-/->
U+219C LEFTWARDS WAVE ARROW		
U+219D RIGHTWARDS WAVE ARROW	⇝	LEADSTO
U+219E LEFTWARDS TWO HEADED ARROW		
U+219F UPWARDS TWO HEADED ARROW		
U+21A0 RIGHTWARDS TWO HEADED ARROW		
U+21A1 DOWNWARDS TWO HEADED ARROW		
U+21A2 LEFTWARDS ARROW WITH TAIL		
U+21A3 RIGHTWARDS ARROW WITH TAIL		
U+21A4 LEFTWARDS ARROW FROM BAR		

U+21A5 UPWARDS ARROW FROM BAR
U+21A7 DOWNWARDS ARROW FROM BAR
U+21A8 UP DOWN ARROW WITH BASE
U+21A9 LEFTWARDS ARROW WITH HOOK
U+21AA RIGHTWARDS ARROW WITH HOOK
U+21AB LEFTWARDS ARROW WITH LOOP
U+21AC RIGHTWARDS ARROW WITH LOOP
U+21AD LEFT RIGHT WAVE ARROW
U+21AE LEFT RIGHT ARROW WITH STROKE
U+21AF DOWNWARDS ZIGZAG ARROW
U+21B0 UPWARDS ARROW WITH TIP LEFTWARDS
U+21B1 UPWARDS ARROW WITH TIP RIGHTWARDS
U+21B2 DOWNWARDS ARROW WITH TIP LEFTWARDS
U+21B3 DOWNWARDS ARROW WITH TIP RIGHTWARDS
U+21B4 RIGHTWARDS ARROW WITH CORNER DOWNWARDS
U+21B5 DOWNWARDS ARROW WITH CORNER LEFTWARDS
U+21B6 ANTICLOCKWISE TOP SEMICIRCLE ARROW
U+21B7 CLOCKWISE TOP SEMICIRCLE ARROW
U+21B8 NORTH WEST ARROW TO LONG BAR
U+21B9 LEFTWARDS ARROW TO BAR OVER RIGHTWARDS ARROW TO BAR
U+21BA ANTICLOCKWISE OPEN CIRCLE ARROW
U+21BB CLOCKWISE OPEN CIRCLE ARROW
U+21BC LEFTWARDS HARPOON WITH BARB UPWARDS ↼ LEFTHARPOONUP
U+21BD LEFTWARDS HARPOON WITH BARB DOWNWARDS ↽ LEFTHARPOONDOWN
U+21BE UPWARDS HARPOON WITH BARB RIGHTWARDS ↾ UPHARPOONRIGHT
U+21BF UPWARDS HARPOON WITH BARB LEFTWARDS ↿ UPHARPOONLEFT
U+21C0 RIGHTWARDS HARPOON WITH BARB UPWARDS ⇀ RIGHTHARPOONUP
U+21C1 RIGHTWARDS HARPOON WITH BARB DOWNWARDS ⇁ RIGHTHARPOONDOWN
U+21C2 DOWNWARDS HARPOON WITH BARB RIGHTWARDS ⇂ DOWNHARPOONRIGHT
U+21C3 DOWNWARDS HARPOON WITH BARB LEFTWARDS ⇃ DOWNHARPOONLEFT
U+21C4 RIGHTWARDS ARROW OVER LEFTWARDS ARROW ⇄ RIGHTLEFTARROWS
U+21C5 UPWARDS ARROW LEFTWARDS OF DOWNWARDS ARROW
U+21C6 LEFTWARDS ARROW OVER RIGHTWARDS ARROW ⇆ LEFTRIGHTARROWS
U+21C7 LEFTWARDS PAIRED ARROWS ⇇ LEFTLEFTARROWS
U+21C8 UPWARDS PAIRED ARROWS ⇈ UPUPARROWS
U+21C9 RIGHTWARDS PAIRED ARROWS ⇉ RIGHTRIGHTARROWS
U+21CA DOWNWARDS PAIRED ARROWS ⇊ DOWNDOWNARROWS
U+21CB LEFTWARDS HARPOON OVER RIGHTWARDS HARPOON
U+21CC RIGHTWARDS HARPOON OVER LEFTWARDS HARPOON ⇌ RIGHTLEFTHARPOONS
U+21CD LEFTWARDS DOUBLE ARROW WITH STROKE ⇍
U+21CE LEFT RIGHT DOUBLE ARROW WITH STROKE ⇎
U+21CF RIGHTWARDS DOUBLE ARROW WITH STROKE ⇏
U+21D0 LEFTWARDS DOUBLE ARROW ⇐
U+21D1 UPWARDS DOUBLE ARROW ⇑
U+21D2 RIGHTWARDS DOUBLE ARROW ⇒ =>
U+21D3 DOWNWARDS DOUBLE ARROW ⇓
U+21D4 LEFT RIGHT DOUBLE ARROW ⇔ <=>
U+21D5 UP DOWN DOUBLE ARROW ⇕
U+21D6 NORTH WEST DOUBLE ARROW
U+21D7 NORTH EAST DOUBLE ARROW
U+21D8 SOUTH EAST DOUBLE ARROW
U+21D9 SOUTH WEST DOUBLE ARROW

U+21DA LEFTWARDS TRIPLE ARROW ⇚
U+21DB RIGHTWARDS TRIPLE ARROW ⇛
U+21DC LEFTWARDS SQUIGGLE ARROW
U+21DD RIGHTWARDS SQUIGGLE ARROW ⇝
U+21DE UPWARDS ARROW WITH DOUBLE STROKE
U+21DF DOWNWARDS ARROW WITH DOUBLE STROKE
U+21E0 LEFTWARDS DASHED ARROW ⇠
U+21E1 UPWARDS DASHED ARROW
U+21E2 RIGHTWARDS DASHED ARROW ⇢
U+21E3 DOWNWARDS DASHED ARROW
U+21E4 LEFTWARDS ARROW TO BAR
U+21E5 RIGHTWARDS ARROW TO BAR
U+21E6 LEFTWARDS WHITE ARROW
U+21E7 UPWARDS WHITE ARROW
U+21E8 RIGHTWARDS WHITE ARROW
U+21E9 DOWNWARDS WHITE ARROW
U+21EA UPWARDS WHITE ARROW FROM BAR
U+21EB UPWARDS WHITE ARROW ON PEDESTAL
U+21EC UPWARDS WHITE ARROW ON PEDESTAL WITH HORIZONTAL BAR
U+21ED UPWARDS WHITE ARROW ON PEDESTAL WITH VERTICAL BAR
U+21EE UPWARDS WHITE DOUBLE ARROW
U+21EF UPWARDS WHITE DOUBLE ARROW ON PEDESTAL
U+21F0 RIGHTWARDS WHITE ARROW FROM WALL
U+21F1 NORTH WEST ARROW TO CORNER
U+21F2 SOUTH EAST ARROW TO CORNER
U+21F3 UP DOWN WHITE ARROW
U+21F4 RIGHT ARROW WITH SMALL CIRCLE
U+21F5 DOWNWARDS ARROW LEFTWARDS OF UPWARDS ARROW
U+21F6 THREE RIGHTWARDS ARROWS
U+21F7 LEFTWARDS ARROW WITH VERTICAL STROKE
U+21F8 RIGHTWARDS ARROW WITH VERTICAL STROKE
U+21F9 LEFT RIGHT ARROW WITH VERTICAL STROKE
U+21FA LEFTWARDS ARROW WITH DOUBLE VERTICAL STROKE
U+21FB RIGHTWARDS ARROW WITH DOUBLE VERTICAL STROKE
U+21FC LEFT RIGHT ARROW WITH DOUBLE VERTICAL STROKE
U+21FD LEFTWARDS OPEN-HEADED ARROW
U+21FE RIGHTWARDS OPEN-HEADED ARROW
U+21FF LEFT RIGHT OPEN-HEADED ARROW
U+2201 COMPLEMENT ∁
U+2202 PARTIAL DIFFERENTIAL ∂ DEL
U+2204 THERE DOES NOT EXIST ∄
U+2206 INCREMENT ∆
U+220F N-ARY PRODUCT ∏ PRODUCT
U+2210 N-ARY COPRODUCT ∐ COPRODUCT
U+2211 N-ARY SUMMATION ∑ SUM
U+2218 RING OPERATOR ∘ CIRC RING COMPOSE
U+2219 BULLET OPERATOR ∙ BULLET
U+221A SQUARE ROOT √ SQRT
U+221B CUBE ROOT CBRT
U+221C FOURTH ROOT FOURTHROOT
U+221D PROPORTIONAL TO ∝ PROPTO
U+2223 DIVIDES ∣ DIVIDES

U+2224 DOES NOT DIVIDE ∤
U+2225 PARALLEL TO ∥ PARALLEL
U+2226 NOT PARALLEL TO ∦ NPARALLEL
U+222B INTEGRAL ∫
U+222C DOUBLE INTEGRAL
U+222D TRIPLE INTEGRAL
U+222E CONTOUR INTEGRAL ∮
U+222F SURFACE INTEGRAL
U+2230 VOLUME INTEGRAL
U+2231 CLOCKWISE INTEGRAL
U+2232 CLOCKWISE CONTOUR INTEGRAL
U+2233 ANTICLOCKWISE CONTOUR INTEGRAL
U+2234 THEREFORE ∴
U+2235 BECAUSE ∵
U+2236 RATIO
U+2237 PROPORTION
U+2239 EXCESS
U+223A GEOMETRIC PROPORTION
U+223B HOMOTHETIC
U+223C TILDE OPERATOR ∼
U+223D REVERSED TILDE ∽
U+223E INVERTED LAZY S
U+223F SINE WAVE
U+2240 WREATH PRODUCT ≀ WREATH
U+2241 NOT TILDE ≁
U+224B TRIPLE TILDE
U+224F DIFFERENCE BETWEEN ≏ BUMPEQ
U+2250 APPROACHES THE LIMIT ≐ DOTEQ
U+2258 CORRESPONDS TO
U+2259 ESTIMATES
U+225A EQUIANGULAR TO
U+225E MEASURED BY
U+226C BETWEEN ≬
U+228C MULTISET
U+229A CIRCLED RING OPERATOR ⊚ CIRCLEDRING
U+229D CIRCLED DASH ⊝
U+22A2 RIGHT TACK ⊢ VDASH TURNSTILE
U+22A3 LEFT TACK ⊣ DASHV
U+22A6 ASSERTION ⊦
U+22A7 MODELS ⊧
U+22A8 TRUE ⊨
U+22A9 FORCES ⊩
U+22AA TRIPLE VERTICAL BAR RIGHT TURNSTILE ⊪
U+22AB DOUBLE VERTICAL BAR DOUBLE RIGHT TURNSTILE
U+22AC DOES NOT PROVE ⊬
U+22AD NOT TRUE
U+22AE DOES NOT FORCE ⊮
U+22AF NEGATED DOUBLE VERTICAL BAR DOUBLE RIGHT TURNSTILE ⊯
U+22B6 ORIGINAL OF
U+22B7 IMAGE OF
U+22B8 MULTIMAP ⊸
U+22B9 HERMITIAN CONJUGATE MATRIX

U+22BA INTERCALATE ⊺
U+22BE RIGHT ANGLE WITH ARC
U+22BF RIGHT TRIANGLE
U+22C0 N-ARY LOGICAL AND ⋀ BIGAND ALL
U+22C1 N-ARY LOGICAL OR ⋁ BIGOR ANY
U+22C2 N-ARY INTERSECTION ⋂ BIGCAP BIGINTERSECT
U+22C3 N-ARY UNION ⋃ BIGCUP BIGUNION
U+22C4 DIAMOND OPERATOR ⋄ DIAMOND
U+22C6 STAR OPERATOR ⋆ STAR
U+22C7 DIVISION TIMES ⋇
U+22C8 BOWTIE ⋈
U+22C9 LEFT NORMAL FACTOR SEMIDIRECT PRODUCT ⋉
U+22CA RIGHT NORMAL FACTOR SEMIDIRECT PRODUCT ⋊
U+22CB LEFT SEMIDIRECT PRODUCT ⋋
U+22CC RIGHT SEMIDIRECT PRODUCT ⋌
U+22D4 PITCHFORK ⋔
U+22EE VERTICAL ELLIPSIS
U+22EF MIDLINE HORIZONTAL ELLIPSIS
U+22F0 UP RIGHT DIAGONAL ELLIPSIS
U+22F1 DOWN RIGHT DIAGONAL ELLIPSIS
U+27C0 THREE DIMENSIONAL ANGLE
U+27C1 WHITE TRIANGLE CONTAINING SMALL WHITE TRIANGLE
U+27C2 PERPENDICULAR PERP
U+27D0 WHITE DIAMOND WITH CENTRED DOT
U+27D2 ELEMENT OF OPENING UPWARDS
U+27D3 LOWER RIGHT CORNER WITH DOT
U+27D4 UPPER LEFT CORNER WITH DOT
U+27D5 LEFT OUTER JOIN
U+27D6 RIGHT OUTER JOIN
U+27D7 FULL OUTER JOIN
U+27D8 LARGE UP TACK
U+27D9 LARGE DOWN TACK
U+27DA LEFT AND RIGHT DOUBLE TURNSTILE
U+27DB LEFT AND RIGHT TACK
U+27DC LEFT MULTIMAP
U+27DD LONG RIGHT TACK
U+27DE LONG LEFT TACK
U+27DF UP TACK WITH CIRCLE ABOVE
U+27E0 LOZENGE DIVIDED BY HORIZONTAL RULE
U+27E1 WHITE CONCAVE-SIDED DIAMOND
U+27E2 WHITE CONCAVE-SIDED DIAMOND WITH LEFTWARDS TICK
U+27E3 WHITE CONCAVE-SIDED DIAMOND WITH RIGHTWARDS TICK
U+27E4 WHITE SQUARE WITH LEFTWARDS TICK
U+27E5 WHITE SQUARE WITH RIGHTWARDS TICK
U+27F0 UPWARDS QUADRUPLE ARROW
U+27F1 DOWNWARDS QUADRUPLE ARROW
U+27F2 ANTICLOCKWISE GAPPED CIRCLE ARROW
U+27F3 CLOCKWISE GAPPED CIRCLE ARROW
U+27F4 RIGHT ARROW WITH CIRCLED PLUS
U+27F5 LONG LEFTWARDS ARROW
U+27F6 LONG RIGHTWARDS ARROW
U+27F7 LONG LEFT RIGHT ARROW

U+27F8 LONG LEFTWARDS DOUBLE ARROW
U+27F9 LONG RIGHTWARDS DOUBLE ARROW
U+27FA LONG LEFT RIGHT DOUBLE ARROW
U+27FB LONG LEFTWARDS ARROW FROM BAR
U+27FC LONG RIGHTWARDS ARROW FROM BAR
U+27FD LONG LEFTWARDS DOUBLE ARROW FROM BAR
U+27FE LONG RIGHTWARDS DOUBLE ARROW FROM BAR
U+27FF LONG RIGHTWARDS SQUIGGLE ARROW
U+2900 RIGHTWARDS TWO-HEADED ARROW WITH VERTICAL STROKE
U+2901 RIGHTWARDS TWO-HEADED ARROW WITH DOUBLE VERTICAL STROKE
U+2902 LEFTWARDS DOUBLE ARROW WITH VERTICAL STROKE
U+2903 RIGHTWARDS DOUBLE ARROW WITH VERTICAL STROKE
U+2904 LEFT RIGHT DOUBLE ARROW WITH VERTICAL STROKE
U+2905 RIGHTWARDS TWO-HEADED ARROW FROM BAR
U+2906 LEFTWARDS DOUBLE ARROW FROM BAR
U+2907 RIGHTWARDS DOUBLE ARROW FROM BAR
U+2908 DOWNWARDS ARROW WITH HORIZONTAL STROKE
U+2909 UPWARDS ARROW WITH HORIZONTAL STROKE
U+290A UPWARDS TRIPLE ARROW
U+290B DOWNWARDS TRIPLE ARROW
U+290C LEFTWARDS DOUBLE DASH ARROW
U+290D RIGHTWARDS DOUBLE DASH ARROW
U+290E LEFTWARDS TRIPLE DASH ARROW
U+290F RIGHTWARDS TRIPLE DASH ARROW
U+2910 RIGHTWARDS TWO-HEADED TRIPLE DASH ARROW
U+2911 RIGHTWARDS ARROW WITH DOTTED STEM
U+2912 UPWARDS ARROW TO BAR
U+2913 DOWNWARDS ARROW TO BAR
U+2914 RIGHTWARDS ARROW WITH TAIL WITH VERTICAL STROKE
U+2915 RIGHTWARDS ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
U+2916 RIGHTWARDS TWO-HEADED ARROW WITH TAIL
U+2917 RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH VERTICAL STROKE
U+2918 RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE
U+2919 LEFTWARDS ARROW-TAIL
U+291A RIGHTWARDS ARROW-TAIL
U+291B LEFTWARDS DOUBLE ARROW-TAIL
U+291C RIGHTWARDS DOUBLE ARROW-TAIL
U+291D LEFTWARDS ARROW TO BLACK DIAMOND
U+291E RIGHTWARDS ARROW TO BLACK DIAMOND
U+291F LEFTWARDS ARROW FROM BAR TO BLACK DIAMOND
U+2920 RIGHTWARDS ARROW FROM BAR TO BLACK DIAMOND
U+2921 NORTH WEST AND SOUTH EAST ARROW
U+2922 NORTH EAST AND SOUTH WEST ARROW
U+2923 NORTH WEST ARROW WITH HOOK
U+2924 NORTH EAST ARROW WITH HOOK
U+2925 SOUTH EAST ARROW WITH HOOK
U+2926 SOUTH WEST ARROW WITH HOOK
U+2927 NORTH WEST ARROW AND NORTH EAST ARROW
U+2928 NORTH EAST ARROW AND SOUTH EAST ARROW
U+2929 SOUTH EAST ARROW AND SOUTH WEST ARROW
U+292A SOUTH WEST ARROW AND NORTH WEST ARROW
U+292B RISING DIAGONAL CROSSING FALLING DIAGONAL

U+292C FALLING DIAGONAL CROSSING RISING DIAGONAL
U+292D SOUTH EAST ARROW CROSSING NORTH EAST ARROW
U+292E NORTH EAST ARROW CROSSING SOUTH EAST ARROW
U+292F FALLING DIAGONAL CROSSING NORTH EAST ARROW
U+2930 RISING DIAGONAL CROSSING SOUTH EAST ARROW
U+2931 NORTH EAST ARROW CROSSING NORTH WEST ARROW
U+2932 NORTH WEST ARROW CROSSING NORTH EAST ARROW
U+2933 WAVE ARROW POINTING DIRECTLY RIGHT
U+2934 ARROW POINTING RIGHTWARDS THEN CURVING UPWARDS
U+2935 ARROW POINTING RIGHTWARDS THEN CURVING DOWNWARDS
U+2936 ARROW POINTING DOWNWARDS THEN CURVING LEFTWARDS
U+2937 ARROW POINTING DOWNWARDS THEN CURVING RIGHTWARDS
U+2938 RIGHT-SIDE ARC CLOCKWISE ARROW
U+2939 LEFT-SIDE ARC ANTICLOCKWISE ARROW
U+293A TOP ARC ANTICLOCKWISE ARROW
U+293B BOTTOM ARC ANTICLOCKWISE ARROW
U+293C TOP ARC CLOCKWISE ARROW WITH MINUS
U+293D TOP ARC ANTICLOCKWISE ARROW WITH PLUS
U+293E LOWER RIGHT SEMICIRCULAR CLOCKWISE ARROW
U+293F LOWER LEFT SEMICIRCULAR ANTICLOCKWISE ARROW
U+2940 ANTICLOCKWISE CLOSED CIRCLE ARROW
U+2941 CLOCKWISE CLOSED CIRCLE ARROW
U+2942 RIGHTWARDS ARROW ABOVE SHORT LEFTWARDS ARROW
U+2943 LEFTWARDS ARROW ABOVE SHORT RIGHTWARDS ARROW
U+2944 SHORT RIGHTWARDS ARROW ABOVE LEFTWARDS ARROW
U+2945 RIGHTWARDS ARROW WITH PLUS BELOW
U+2946 LEFTWARDS ARROW WITH PLUS BELOW
U+2947 RIGHTWARDS ARROW THROUGH X
U+2948 LEFT RIGHT ARROW THROUGH SMALL CIRCLE
U+2949 UPWARDS TWO-HEADED ARROW FROM SMALL CIRCLE
U+294A LEFT BARB UP RIGHT BARB DOWN HARPOON
U+294B LEFT BARB DOWN RIGHT BARB UP HARPOON
U+294C UP BARB RIGHT DOWN BARB LEFT HARPOON
U+294D UP BARB LEFT DOWN BARB RIGHT HARPOON
U+294E LEFT BARB UP RIGHT BARB UP HARPOON
U+294F UP BARB RIGHT DOWN BARB RIGHT HARPOON
U+2950 LEFT BARB DOWN RIGHT BARB DOWN HARPOON
U+2951 UP BARB LEFT DOWN BARB LEFT HARPOON
U+2952 LEFTWARDS HARPOON WITH BARB UP TO BAR
U+2953 RIGHTWARDS HARPOON WITH BARB UP TO BAR
U+2954 UPWARDS HARPOON WITH BARB RIGHT TO BAR
U+2955 DOWNWARDS HARPOON WITH BARB RIGHT TO BAR
U+2956 LEFTWARDS HARPOON WITH BARB DOWN TO BAR
U+2957 RIGHTWARDS HARPOON WITH BARB DOWN TO BAR
U+2958 UPWARDS HARPOON WITH BARB LEFT TO BAR
U+2959 DOWNWARDS HARPOON WITH BARB LEFT TO BAR
U+295A LEFTWARDS HARPOON WITH BARB UP FROM BAR
U+295B RIGHTWARDS HARPOON WITH BARB UP FROM BAR
U+295C UPWARDS HARPOON WITH BARB RIGHT FROM BAR
U+295D DOWNWARDS HARPOON WITH BARB RIGHT FROM BAR
U+295E LEFTWARDS HARPOON WITH BARB DOWN FROM BAR
U+295F RIGHTWARDS HARPOON WITH BARB DOWN FROM BAR

U+2960 UPWARDS HARPOON WITH BARB LEFT FROM BAR
U+2961 DOWNWARDS HARPOON WITH BARB LEFT FROM BAR
U+2962 LEFTWARDS HARPOON WITH BARB UP ABOVE LEFTWARDS HARPOON WITH BARB DOWN
U+2963 UPWARDS HARPOON WITH BARB LEFT BESIDE UPWARDS HARPOON WITH BARB RIGHT
U+2964 RIGHTWARDS HARPOON WITH BARB UP ABOVE RIGHTWARDS HARPOON WITH BARB DOWN
U+2965 DOWNWARDS HARPOON WITH BARB LEFT BESIDE DOWNWARDS HARPOON WITH BARB RIGHT
U+2966 LEFTWARDS HARPOON WITH BARB UP ABOVE RIGHTWARDS HARPOON WITH BARB UP
U+2967 LEFTWARDS HARPOON WITH BARB DOWN ABOVE RIGHTWARDS HARPOON WITH BARB DOWN
U+2968 RIGHTWARDS HARPOON WITH BARB UP ABOVE LEFTWARDS HARPOON WITH BARB UP
U+2969 RIGHTWARDS HARPOON WITH BARB DOWN ABOVE LEFTWARDS HARPOON WITH BARB DOWN
U+296A LEFTWARDS HARPOON WITH BARB UP ABOVE LONG DASH
U+296B LEFTWARDS HARPOON WITH BARB DOWN BELOW LONG DASH
U+296C RIGHTWARDS HARPOON WITH BARB UP ABOVE LONG DASH
U+296D RIGHTWARDS HARPOON WITH BARB DOWN BELOW LONG DASH
U+296E UPWARDS HARPOON WITH BARB LEFT BESIDE DOWNWARDS HARPOON WITH BARB RIGHT
U+296F DOWNWARDS HARPOON WITH BARB LEFT BESIDE UPWARDS HARPOON WITH BARB RIGHT
U+2970 RIGHT DOUBLE ARROW WITH ROUNDED HEAD
U+2971 EQUALS SIGN ABOVE RIGHTWARDS ARROW
U+2972 TILDE OPERATOR ABOVE RIGHTWARDS ARROW
U+2973 LEFTWARDS ARROW ABOVE TILDE OPERATOR
U+2974 RIGHTWARDS ARROW ABOVE TILDE OPERATOR
U+2975 RIGHTWARDS ARROW ABOVE ALMOST EQUAL TO
U+2976 LESS-THAN ABOVE LEFTWARDS ARROW
U+2977 LEFTWARDS ARROW THROUGH LESS-THAN
U+2978 GREATER-THAN ABOVE RIGHTWARDS ARROW
U+2979 SUBSET ABOVE RIGHTWARDS ARROW
U+297A LEFTWARDS ARROW THROUGH SUBSET
U+297B SUPERSET ABOVE LEFTWARDS ARROW
U+297C LEFT FISH TAIL
U+297D RIGHT FISH TAIL
U+297E UP FISH TAIL
U+297F DOWN FISH TAIL
U+2980 TRIPLE VERTICAL BAR DELIMITER
U+2981 Z NOTATION SPOT
U+2982 Z NOTATION TYPE COLON
U+2999 DOTTED FENCE
U+299A VERTICAL ZIGZAG LINE
U+299B MEASURED ANGLE OPENING LEFT
U+299C RIGHT ANGLE VARIANT WITH SQUARE
U+299D MEASURED RIGHT ANGLE WITH DOT
U+299E ANGLE WITH S INSIDE
U+299F ACUTE ANGLE
U+29A0 SPHERICAL ANGLE OPENING LEFT
U+29A1 SPHERICAL ANGLE OPENING UP
U+29A2 TURNED ANGLE
U+29A3 REVERSED ANGLE
U+29A4 ANGLE WITH UNDERBAR
U+29A5 REVERSED ANGLE WITH UNDERBAR
U+29A6 OBLIQUE ANGLE OPENING UP
U+29A7 OBLIQUE ANGLE OPENING DOWN
U+29A8 MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING UP AND RIGHT
U+29A9 MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING UP AND LEFT

U+29AA MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING DOWN AND RIGHT
U+29AB MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING DOWN AND LEFT
U+29AC MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING RIGHT AND UP
U+29AD MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING LEFT AND UP
U+29AE MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING RIGHT AND DOWN
U+29AF MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING LEFT AND DOWN
U+29B0 REVERSED EMPTY SET
U+29B1 EMPTY SET WITH OVERBAR
U+29B2 EMPTY SET WITH SMALL CIRCLE ABOVE
U+29B3 EMPTY SET WITH RIGHT ARROW ABOVE
U+29B4 EMPTY SET WITH LEFT ARROW ABOVE
U+29B5 CIRCLE WITH HORIZONTAL BAR
U+29B6 CIRCLED VERTICAL BAR
U+29B7 CIRCLED PARALLEL
U+29B9 CIRCLED PERPENDICULAR
U+29BA CIRCLE DIVIDED BY HORIZONTAL BAR AND TOP HALF DIVIDED BY VERTICAL BAR
U+29BB CIRCLE WITH SUPERIMPOSED X
U+29BD UP ARROW THROUGH CIRCLE
U+29BE CIRCLED WHITE BULLET
U+29BF CIRCLED BULLET
U+29C2 CIRCLE WITH SMALL CIRCLE TO THE RIGHT
U+29C3 CIRCLE WITH TWO HORIZONTAL STROKES TO THE RIGHT
U+29C5 SQUARED FALLING DIAGONAL SLASH
U+29C7 SQUARED SMALL CIRCLE
U+29C8 SQUARED SQUARE
U+29C9 TWO JOINED SQUARES
U+29CA TRIANGLE WITH DOT ABOVE
U+29CB TRIANGLE WITH UNDERBAR
U+29CC S IN TRIANGLE
U+29CD TRIANGLE WITH SERIFS AT BOTTOM
U+29CE RIGHT TRIANGLE ABOVE LEFT TRIANGLE
U+29CF LEFT TRIANGLE BESIDE VERTICAL BAR
U+29D0 VERTICAL BAR BESIDE RIGHT TRIANGLE
U+29D1 BOWTIE WITH LEFT HALF BLACK
U+29D2 BOWTIE WITH RIGHT HALF BLACK
U+29D3 BLACK BOWTIE
U+29D6 WHITE HOURGLASS
U+29D7 BLACK HOURGLASS
U+29DC INCOMPLETE INFINITY
U+29DD TIE OVER INFINITY
U+29DE INFINITY NEGATED WITH VERTICAL BAR
U+29DF DOUBLE-ENDED MULTIMAP
U+29E0 SQUARE WITH CONTOURED OUTLINE
U+29E1 INCREASES AS
U+29E2 SHUFFLE PRODUCT
U+29E6 GLEICH STARK
U+29E7 THERMODYNAMIC
U+29E8 DOWN-POINTING TRIANGLE WITH LEFT HALF BLACK
U+29E9 DOWN-POINTING TRIANGLE WITH RIGHT HALF BLACK
U+29EA BLACK DIAMOND WITH DOWN ARROW
U+29EB BLACK LOZENGE
U+29EC WHITE CIRCLE WITH DOWN ARROW

U+29ED BLACK CIRCLE WITH DOWN ARROW
U+29EE ERROR-BARRED WHITE SQUARE
U+29EF ERROR-BARRED BLACK SQUARE
U+29F0 ERROR-BARRED WHITE DIAMOND
U+29F1 ERROR-BARRED BLACK DIAMOND
U+29F2 ERROR-BARRED WHITE CIRCLE
U+29F3 ERROR-BARRED BLACK CIRCLE
U+29F4 RULE-DELAYED
U+29F6 SOLIDUS WITH OVERBAR
U+29F7 REVERSE SOLIDUS WITH HORIZONTAL STROKE
U+29FA DOUBLE PLUS
U+29FB TRIPLE PLUS
U+29FE TINY
U+29FF MINY
U+2A00 N-ARY CIRCLED DOT OPERATOR ⨀ BIGODOT
U+2A01 N-ARY CIRCLED PLUS OPERATOR ⨁ BIGOPLUS
U+2A02 N-ARY CIRCLED TIMES OPERATOR ⨂ BIGOTIMES
U+2A03 N-ARY UNION OPERATOR WITH DOT BIGUDOT
U+2A04 N-ARY UNION OPERATOR WITH PLUS BIGUPLUS
U+2A05 N-ARY SQUARE INTERSECTION OPERATOR BIGSQCAP
U+2A06 N-ARY SQUARE UNION OPERATOR BIGSQCUP
U+2A07 TWO LOGICAL AND OPERATOR
U+2A08 TWO LOGICAL OR OPERATOR
U+2A09 N-ARY TIMES OPERATOR BIGTIMES
U+2A0A MODULO TWO SUM
U+2A10 CIRCULATION FUNCTION
U+2A11 ANTICLOCKWISE INTEGRATION
U+2A12 LINE INTEGRATION WITH RECTANGULAR PATH AROUND POLE
U+2A13 LINE INTEGRATION WITH SEMICIRCULAR PATH AROUND POLE
U+2A14 LINE INTEGRATION NOT INCLUDING THE POLE
U+2A1D JOIN ⋈ JOIN
U+2A1E LARGE LEFT TRIANGLE OPERATOR
U+2A1F Z NOTATION SCHEMA COMPOSITION
U+2A20 Z NOTATION SCHEMA PIPING
U+2A21 Z NOTATION SCHEMA PROJECTION
U+2A32 SEMIDIRECT PRODUCT WITH BOTTOM CLOSED
U+2A33 SMASH PRODUCT
U+2A3C INTERIOR PRODUCT
U+2A3D RIGHTHAND INTERIOR PRODUCT
U+2A3E Z NOTATION RELATIONAL COMPOSITION
U+2A3F AMALGAMATION OR COPRODUCT
U+2A57 SLOPING LARGE OR
U+2A58 SLOPING LARGE AND
U+2A61 SMALL VEE WITH UNDERBAR
U+2A64 Z NOTATION DOMAIN ANTIRESTRICTION
U+2A65 Z NOTATION RANGE ANTIRESTRICTION
U+2A68 TRIPLE HORIZONTAL BAR WITH DOUBLE VERTICAL STROKE
U+2A69 TRIPLE HORIZONTAL BAR WITH TRIPLE VERTICAL STROKE
U+2A6A TILDE OPERATOR WITH DOT ABOVE
U+2A6B TILDE OPERATOR WITH RISING DOTS
U+2A6D CONGRUENT WITH DOT ABOVE
U+2ACD SQUARE LEFT OPEN BOX OPERATOR

U+2ACE SQUARE RIGHT OPEN BOX OPERATOR
U+2AD9 ELEMENT OF OPENING DOWNWARDS
U+2ADA PITCHFORK WITH TEE TOP
U+2ADC FORKING
U+2ADD NONFORKING
U+2ADE SHORT LEFT TACK
U+2ADF SHORT DOWN TACK
U+2AE0 SHORT UP TACK
U+2AE1 PERPENDICULAR WITH S
U+2AE2 VERTICAL BAR TRIPLE RIGHT TURNSTILE
U+2AE3 DOUBLE VERTICAL BAR LEFT TURNSTILE
U+2AE4 VERTICAL BAR DOUBLE LEFT TURNSTILE
U+2AE5 DOUBLE VERTICAL BAR DOUBLE LEFT TURNSTILE
U+2AE6 LONG DASH FROM LEFT MEMBER OF DOUBLE VERTICAL
U+2AE7 SHORT DOWN TACK WITH OVERBAR
U+2AE8 SHORT UP TACK WITH UNDERBAR
U+2AE9 SHORT UP TACK ABOVE SHORT DOWN TACK
U+2AEA DOUBLE DOWN TACK
U+2AEB DOUBLE UP TACK
U+2AEC DOUBLE STROKE NOT SIGN
U+2AED REVERSED DOUBLE STROKE NOT SIGN
U+2AEE DOES NOT DIVIDE WITH REVERSED NEGATION SLASH
U+2AEF VERTICAL LINE WITH CIRCLE ABOVE
U+2AF0 VERTICAL LINE WITH CIRCLE BELOW
U+2AF1 DOWN TACK WITH CIRCLE BELOW
U+2AF2 PARALLEL WITH HORIZONTAL STROKE
U+2AF3 PARALLEL WITH TILDE OPERATOR
U+2AF5 TRIPLE VERTICAL BAR WITH HORIZONTAL STROKE
U+2AF6 TRIPLE COLON OPERATOR
U+2AFB TRIPLE SOLIDUS BINARY RELATION
U+2AFC LARGE TRIPLE VERTICAL BAR OPERATOR
U+2AFE WHITE VERTICAL BAR
U+2AFF N-ARY WHITE VERTICAL BAR

Appendix G

Concrete Syntax

In this chapter, we describe the concrete syntax of Fortress programs in BNF notation. This syntax is "human-readable" in the sense that it does not describe uses of whitespaces and semicolons exactly. Instead, they are described as follows. Fortress has three different contexts influencing the whitespace-sensitivity of expressions:

statement Expressions immediately enclosed by a block expression are in a statement-like context. Multiple expressions can appear on a line if they are separated (or terminated) by semicolons. If an expression can legally end at the end of a line, it does; if it cannot, it does not. A prefix or infix operator that lacks its last operand prevents an expression from ending. For example,

$$
\begin{array}{l}
an = expression+ \\
\quad\quad spanning+ \\
\quad\quad four+ \\
\quad\quad lines \\
a = oneLiner \\
four(); on(); one(); line();
\end{array}
$$

nested An expression or list of expressions immediately enclosed by parentheses or braces is nested. Multiple expressions are separated by commas, and the end of a line does not end an expression. Because of this effect, the meaning of a several lines of code can change if they are wrapped in parentheses. Parentheses can also be used to ensure that a multiline expression is not terminated prematurely without paying special attention to line endings.

$$
\begin{array}{l}
lhs = rhs \\
-aSeparateExpression \\
postProfit(revenue \\
\quad\quad\quad -expenses)
\end{array}
$$

pasted Fortress has special syntax for matrix pasting. Within square brackets, whitespace-separated expressions are treated (depending on their type) as either matrix elements or submatrices within a row. Because whitespace is the separator, it also ends expressions where possible. In addition, newline-or-semicolon-separated rows are pasted vertically along their columns. Higher-dimensional pasting is expressed with repeated semicolons, but repeated newlines do not have the same effect.

$$
\begin{array}{l}
id2a = [1\ 0; 0\ 1\ ] \\
id2b = [1\ 0; \\
\quad\quad\quad 0\ 1\ ] \\
id2c = [1\ 0 \\
\quad\quad\quad 0\ 1\ ]
\end{array}
$$

$$cube_2 = [1\ 0; 0\ 1; ; 1 - 1; 1\ 1\ ]$$

A restricted form of the pasting syntax can also be used on the left hand side of variable declarations to express both declaration and submatrix decomposition.

$$\begin{array}{l} [top \\ \ bot\ ] = X \\ [left\ right\ ] = Y \\ Z = [top \cdot left\ top \cdot right; \\ \qquad bot \cdot left\ bot \cdot right\ ] \end{array}$$

Section 6.5 describes matrix unpasting in detail and includes more examples.

CompilationUnit	::=	*Component*
	|	*Api*
Component	::=	`component` *DottedId Import** *Export** *Decl** `end`
Api	::=	`api` *DottedId Import** *AbsDecl** `end`
DottedId	::=	*Id* (`.` *Id*)*
Import	::=	`import` *ImportFrom* `from` *DottedId*
	|	`import` *AliasedDottedIds*
ImportFrom	::=	`*` [`except` *Names*]
	|	*AliasedNames*
Names	::=	*Name*
	|	{ *NameList* }
Name	::=	*Id*
	|	`opr` *Op*
NameList	::=	*Name* (`,` *Name*)*
AliasedNames	::=	*AliasedName*
	|	{ *AliasedNameList* }
AliasedName	::=	*Id* [`as` *DottedId*]
	|	`opr` *Op* [`as` *Op*]
	|	`opr` *LeftEncloser RightEncloser* [`as` *LeftEncloser RightEncloser*]
AliasedNameList	::=	*AliasedName* (`,` *AliasedName*)*
AliasedDottedIds	::=	*AliasedDottedId*
	|	{ *AliasedDottedIdList* }
AliasedDottedId	::=	*DottedId* [`as` *DottedId*]
AliasedDottedIdList	::=	*AliasedDottedId* (`,` *AliasedDottedId*)*
Export	::=	`export` *DottedIds*
DottedIds	::=	*DottedId*
	|	{ *DottedIdList* }
DottedIdList	::=	*DottedId* (`,` *DottedId*)*
Decl	::=	*TraitDecl*
	|	*ObjectDecl*
	|	*FnDecl*
	|	*VarDecl*
	|	*DimUnitDecl*
	|	*TypeAlias*
	|	*TestDecl*
	|	*PropertyDecl*
	|	*ExternalSyntax*
TraitDecl	::=	*TraitHeader* (*MdDecl* | *AbsFldDecl* | *PropertyDecl*)* `end`
TraitHeader	::=	*TraitMod** `trait` *Id* [*StaticParams*] [*Extends*] [*Excludes*] [*Comprises*] [*Where*]

Extends ::= `extends` *TraitTypes*
Excludes ::= `excludes` *TraitTypes*
Comprises ::= `comprises` *MayTraitTypes*
TraitTypes ::= *TraitType*
| { *TraitTypeList* }
TraitTypeList ::= *TraitType* (, *TraitType*)*
MayTraitTypes ::= {}
| *TraitTypes*
Where ::= `where` { *WhereClauseList* }
WhereClauseList ::= *WhereClause* (, *WhereClause*)*
WhereClause ::= *Id Extends*
| *TypeAlias*
| *NatConstranint*
| *IntConstranint*
| *BoolConstraint*
| *UnitConstraint*
| *TypeRef* `coerces` *TypeRef*
| *TypeRef* `widens` *TypeRef*
ObjectDecl ::= *ObjectHeader* (*MdDef* | *FldDef* | *PropertyDecl*)* `end`
ObjectHeader ::= *ObjectMod** `object` *Id* [*StaticParams*] [([*ObjectParams*])] [*Extends*] *FnClauses*
ObjectParams ::= *ObjectParam*(, *ObjectParam*)*
| [*ObjectParam*(, *ObjectParam*)* ,] *ObjectVarargs*
| [*ObjectParam*(, *ObjectParam*)* ,] [*ObjectVarargs* ,] *ObjectKeyword* (, *ObjectKeyword*)*
ObjectVarargs ::= `transient` *Id* : *TypeRef* ...
ObjectKeyword ::= *ObjectParam* = *Expr*
ObjectParam ::= *FldMod** *PlainParam*
| `transient` *PlainParam*
FnDecl ::= *AbsFnDecl*
| *FnDef*
AbsFnDecl ::= *FnMod** *FnHeader*
| *Name* : *ArrowType*
FnDef ::= *FnMod** *FnHeader* = *Expr*
FnHeader ::= *Id* [*StaticParams*] *ValParam* [*IsType*] *FnClauses*
| *OpHeader*
OpHeader ::= `opr` *Op* [*StaticParams*] *ValParam* [*IsType*] *FnClauses*
| `opr` [*StaticParams*] *ValParam Op* [*IsType*] *FnClauses*
| `opr` [*StaticParams*] *LeftEncloser ValParams RightEncloser* [:= *ValParam*] [*IsType*] *FnClauses*
ValParam ::= *ParamId*
| ([*ValParams*])
ParamId ::= *Id*
| _
ValParams ::= *PlainParam*(, *PlainParam*)*
| [*PlainParam*(, *PlainParam*)* ,] *Id* : *TypeRef* ...
| [*PlainParam*(, *PlainParam*)* ,] [*Id* : *TypeRef* ... ,] *PlainParam* = *Expr* (, *PlainParam* = *Expr*)*
PlainParam ::= *ParamId* [*IsType*]
| *TypeRef*
IsType ::= : *TypeRef*
FnClauses ::= [*Throws*] [*Where*] [*Contract*]
Throws ::= `throws` *MayTraitTypes*
Contract ::= [*Requires*] [*Ensures*] [*Invariant*]
Requires ::= `requires` *Expr*+
Ensures ::= `ensures` (*Expr*+ [`provided` *Expr*])+
Invariant ::= `invariant` *Expr*+

VarDecl ::= *Vars* (= | :=) *Expr*
| *VarWTypes*
| *VarWoTypes* : *TypeRef* ... [(= | :=) *Expr*]
| *VarWoTypes* : *SimpleTupleType* [(= | :=) *Expr*]

Vars ::= *Var*
| (*Var* (, *Var*)$^+$)

Var ::= *VarMod** *Id* [*IsType*]

VarWTypes ::= *VarWType*
| (*VarWType* (, *VarWType*)$^+$)

VarWType ::= *VarMod** *Id IsType*

VarWoTypes ::= *VarWoType*
| (*VarWoType* (, *VarWoType*)$^+$)

VarWoType ::= *VarMod** *Id*

SimpleTupleType ::= (*TypeRef* , *TypeRefList*)

TypeRefList ::= *TypeRef* (, *TypeRef*)*

DimUnitDecl ::= `dim` *Id* [= *DimRef*] [`default` *Unit*]
| (`unit` | `SI_unit`) *Id*$^+$ [: *DimRef*] [= *Expr*]
| `dim` *Id* [= *DimRef*] (`unit` | `SI_unit`) *Id*$^+$ [= *Expr*]

TypeAlias ::= `type` *Id* [*StaticParams*] = *TypeRef*

TestDecl ::= `test` *Id* [*GeneratorList*] = *Expr*

PropertyDecl ::= `property` [*Id* =] [∀ *ValParam*] *Expr*

MdDecl ::= *AbsMdDecl*
| *MdDef*

AbsMdDecl ::= [`abstract`] *MdMod** *MdHeader*

MdDef ::= *MdMod** *MdHeader* = *Expr*
| *Coercion*

MdHeader ::= [(*Id* | `self`) .]*Id* [*StaticParams*] ([*MdParams*]) [*IsType*] *FnClauses*

MdParams ::= *MdParam*(, *MdParam*)*
| [*MdParam*(, *MdParam*)* ,] *Id* : *TypeRef* ...
| [*MdParam*(, *MdParam*)* ,] [*Id* : *TypeRef* ... ,] *MdParam* = *Expr* (, *MdParam* = *Expr*)*

MdParam ::= *ParamId* [*IsType*]
| `self`
| *TypeRef*

Coercion ::= [`widening`] `coercion` [*StaticParams*] (*Id IsType*) *CoercionClauses* = *Expr*

CoercionClauses ::= [*Throws*] [*CoercionWhere*] [*Contract*]

CoercionWhere ::= `where` { *CoercionWhereClauseList* }

CoercionWhereClauseList ::= *CoercionWhereClause* (, *CoercionWhereClause*)*

CoercionWhereClause ::= *WhereClause*
| *TypeRef* `widens or coerces` *TypeRef*

AbsFldDecl ::= *AbsFldMod** *Id IsType*

FldDef ::= *FldMod** *Id* [*IsType*] (= | :=) *Expr*

UniversalMod ::= `private` | `test`

TraitMod ::= `value` | *UniversalMod*

ObjectMod ::= *TraitMod*

FnMod ::= `atomic` | `io` | *UniversalMod*

VarMod ::= `var` | *UniversalMod*

MdMod ::= `getter` | `setter` | *FnMod*

AbsFldMod ::= `hidden` | `settable` | `wrapped` | *UniversalMod*

FldMod ::= `var` | *AbsFldMod*

StaticParams ::= ⟦*StaticParamList*⟧

StaticParamList ::= *StaticParam* (, *StaticParam*)*

StaticParam ::= *Id* [*Extends*] [`absorbs unit`]
| `nat` *Id*
| `int` *Id*
| `bool` *Id*
| `dim` *Id*
| `unit` *Id* [`:` *DimRef*] [`absorbs unit`]
| `opr` *Op*
| `ident` *Id*

TypeRef ::= *TraitType*
| *TupleType*
| *ArrowType*
| BottomType
| ( )
| (*TypeRef*)
| *DimType*

TraitType ::= *DottedId* [⟦ *StaticArgList* ⟧]
| { *TypeRef* ↦ *TypeRef* }
| ⟨ *TypeRef* ⟩
| *TypeRef* [[*ArraySize*]]
| *TypeRef* [*MatrixSize*]

ArraySize ::= *Extent* (, *Extent*)*

Extent ::= *NatRef*
| *NatRef* # *NatRef*

MatrixSize ::= *NatRef* (× *NatRef*)+

TupleType ::= (*TypeRef*(, *TypeRef*)+)
| ([*TypeRef*(, *TypeRef*)* ,] *TypeRef* ...)
| ([*TypeRef*(, *TypeRef*)* ,] [*TypeRef* ... ,] *Id* = *TypeRef* (, *Id* = *TypeRef*)*)

ArrowType ::= *ArrowTypeRef* → *ArrowTypeRef* [*Throws*]

ArrowTypeRef ::= *TypeRef* (× *TypeRef*)*
| *TypeRef* ^ *Number*

DimType ::= *DimRef*
| *TypeRef DimRef* | *TypeRef* · *DimRef*
| *TypeRef* / *DimRef* | *TypeRef* per *DimRef*
| *TypeRef UnitRef* | *TypeRef* · *UnitRef*
| *TypeRef* / *UnitRef* | *TypeRef* per *UnitRef*
| *TypeRef* `in` *DimRef*

DimRef ::= Unity
| *DottedId*
| *DimRef DimRef* | *DimRef* · *DimRef*
| *DimRef* / *DimRef* | *DimRef* per *DimRef*
| *DimRef* ^ *NatRef* | 1 / *DimRef* | (*DimRef*)
| *DUPreOp DimRef* | *DimRef DUPostOp*

UnitRef ::= dimensionless
| *DottedId*
| *UnitRef UnitRef* | *UnitRef* · *UnitRef*
| *UnitRef* / *UnitRef* | *UnitRef* per *UnitRef*
| *UnitRef* ^*NatRef* | 1/ *UnitRef* | (*UnitRef*)
| *DUPreOp UnitRef* | *UnitRef DUPostOp*

DUPreOp ::= square | cubic | inverse

DUPostOp ::= squared | cubed

StaticArgList ::= *StaticArg* (, *StaticArg*)*

StaticArg ::= *TypeRef*
| *NatRef*
| *IntRef*
| *BoolRef*
| *DimRef*
| *UnitRef*
| *Op*
| *Id*

NatRef ::= *Number*
| *Id*
| *NatRef NatRef*
| *NatRef* + *NatRef*
| *NatRef* · *NatRef*
| (*NatRef*)

IntRef ::= *NatRef*

BoolRef ::= *true*
| *false*
| *Id*
| *BoolRef* `AND` *BoolRef*
| *BoolRef* `OR` *BoolRef*
| (*BoolRef*)

Expr ::= *Flow*
| *Value*
| *DottedName*[⟦*StaticArgList* ⟧]
| `self`
| *Expr* . *Id*
| *Expr* . *Id*[⟦*StaticArgList* ⟧]([*ExprList*])
| *Expr Expr*
| *TraitType* . `coercion`[⟦*StaticArgList* ⟧](*Expr*)
| *Op Expr*
| *Expr Op* [*Expr*]
| *Expr AssignOp Expr*
| *Comprehension*
| *Expr* `as` *TypeRef*
| *Expr* `asif` *TypeRef*
| *UnitExpr*

Flow ::= *Do*
| `label` *Id* *Expr*$^+$ `end` *Id*
| `exit` [*Id*] [`with` *Expr*]
| `while` *Expr Do*
| `for` *GeneratorList Do*
| *Accumulator* [[*GeneratorList*]] *Expr*
| `if` *Expr* `then` *Expr*$^+$ (`elif` *Expr* `then` *Expr*$^+$)* [*Else*] `end`
| (`if` *Expr* `then` *Expr*$^+$ (`elif` *Expr* `then` *Expr*$^+$)* *Else* [`end`])
| `case` *Expr* [*Op*] `of` (*Expr* ⇒ *Expr*$^+$)$^+$ [*Else*] `end`
| `case` (`largest` | `smallest`) [*Op*] `of` (*Expr* ⇒ *Expr*$^+$)$^+$ `end`
| `typecase` *TypecaseBindings* `in` (*TypecaseTypeRefs* ⇒ *Expr*$^+$)$^+$ [*Else*] `end`
| `atomic` *Expr*
| `tryatomic` *Expr*
| `spawn` *Expr*
| `throw` *Expr*
| `try` *Expr*$^+$ [`catch` *Id* (*TraitType* ⇒ *Expr*$^+$)$^+$] [`forbid` *TraitTypes*] [`finally` *Expr*$^+$] `end`

Value	::=	*Literal*
	|	`fn` *ValParam* [*IsType*] [*Throws*] ⇒ *Expr*
	|	`object` [*Extends*] (*FldDef* | *MdDef*)* `end`
	|	*Aggregate*
	|	*LeftEncloser ExprList RightEncloser*
DottedName	::=	*DottedId*
	|	`opr` *Op*
ExprList	::=	*Expr* (, *Expr*)*
AssignOp	::=	:= | *Op* =
Do	::=	`do` *BlockElem** `end`
	|	`do` *BlockElem*+ `also` *Do*
	|	`at` *Expr Do*
BlockElem	::=	*Expr*[, *GeneratorList*]
	|	*LocalVarFnDecl*
GeneratorList	::=	*Generator* (, *Generator*)*
Generator	::=	*Id* ← *Expr*
	|	(*Id* , *IdList*) ← *Expr*
	|	*Expr*
IdList	::=	*Id* (, *Id*)*
Accumulator	::=	∑ | ∏ | `BIG` *Op*
Else	::=	`else` *Expr*+
TypecaseBindings	::=	*Id*
	|	*Binding*
	|	(*BindingList*)
Binding	::=	*Id* = *Expr*
BindingList	::=	*Binding* (, *Binding*)*
TypecaseTypeRefs	::=	*TypeRef*
	|	(*TypeRefList*)
Aggregate	::=	{ [*ExprList*] }
	|	{ *EntryList* }
	|	⟨ [*ExprList*] ⟩
	|	[(*Expr* | ;)*]
	|	(*Expr*(, *Expr*)+)
	|	([*Expr*(, *Expr*)* ,] *Expr* ...)
	|	([*Expr*(, *Expr*)* ,] [*Expr* ... ,] *Id* = *Expr* (, *Id* = *Expr*)*)
EntryList	::=	*Entry* (, *Entry*)*
Entry	::=	*Expr* ↦ *Expr*
Comprehension	::=	{ *Expr* | *GeneratorList* }
	|	{ *Expr* ↦ *Expr* | *GeneratorList* }
	|	⟨ *Expr* | *GeneratorList* ⟩
	|	[(*ArrayComprehensionLeft* | *GeneratorList*)+]
ArrayComprehensionLeft	::=	*Id* ↦ *Expr*
	|	(*Id* , *IdList*) ↦ *Expr*
LocalVarFnDecl	::=	*LocalVarDecl*
	|	*Id ValParam* [*IsType*] [*Throws*] = *Expr*
LocalVarDecl	::=	*LocalVars* (= | :=) *Expr*
	|	*LocalVarWTypes*
	|	*LocalVarWoTypes* : *TypeRef* ... [(= | :=) *Expr*]
	|	*LocalVarWoTypes* : *SimpleTupleType* [(= | :=) *Expr*]
LocalVars	::=	*LocalVar*
	|	(*LocalVar* (, *LocalVar*)+)
LocalVar	::=	*LocalVarWType*
	|	*LocalVarWoType*

LocalVarWTypes ::= *LocalVarWType*
| (*LocalVarWType* (, *LocalVarWType*)$^+$)
LocalVarWType ::= [`var`] *Id IsType*
LocalVarWoTypes ::= *LocalVarWoType*
| (*LocalVarWoType* (, *LocalVarWoType*)$^+$)
LocalVarWoType ::= [`var`] *Id*
| *Unpasting*
Unpasting ::= [*L-Elt* (*Paste L-Elt*)*]
L-Elt ::= *Id* [[*L-ArraySize*]]
| *Unpasting*
L-ArraySize ::= *L-Extent* ($\times$*L-Extent*)*
L-Extent ::= *Expr*
| *Expr* : *Expr*
| *Expr* # *Expr*
Paste ::= (*Whitespace* | ;)$^+$
UnitExpr ::= *UnitRef*
| *Expr UnitRef* | *Expr* $\cdot$ *UnitRef*
| *Expr* / *UnitRef* | *Expr* per *UnitRef*
| *Expr* `in` *UnitRef*
ExternalSyntax ::= `syntax` *OpenExpander Id CloseExpander* = *Expr*
OpenExpander ::= *Id* | *LeftEncloser*
CloseExpander ::= *Id* | *RightEncloser* | `end`
AbsDecl ::= *AbsTraitDecl*
| *AbsObjectDecl*
| *AbsFnDecl*
| *AbsVarDecl*
| *AbsDimUnitDecl*
| *AbsTypeAlias*
| *TestDecl*
| *PropertyDecl*
| *AbsExternalSyntax*
AbsTraitDecl ::= *TraitHeader* (*AbsMdDecl* | *AbsCoercion* | *ApiFldDecl* | *PropertyDecl*)* `end`
AbsObjectDecl ::= *ObjectHeader* (*AbsMdDecl* | *AbsCoercion* | *ApiFldDecl* | *PropertyDecl*)* `end`
AbsCoercion ::= [`widening`] `coercion` [*StaticParams*] (*Id IsType*) *CoercionClauses*
ApiFldDecl ::= *ApiFldMod** *Id IsType*
ApiFldMod ::= `hidden` | `settable` | *UniversalMod*
AbsVarDecl ::= *VarWTypes*
| *VarWoTypes* : *TypeRef* ...
| *VarWoTypes* : *SimpleTupleType*
AbsDimUnitDecl ::= `dim` *Id* [`default` *Unit*]
| (`unit` | `SI_unit`) *Id*$^+$ [: *DimRef*]
| `dim` *Id* (`unit` | `SI_unit`) *Id*$^+$
AbsTypeAlias ::= `type` *Id* [*StaticParams*]
AbsExternalSyntax ::= `syntax` *OpenExpander Id CloseExpander*

Appendix H

Generated Concrete Syntax

This grammar is automatically derived from a Fortress parser under development, and is included to help give insight into how some syntactic features (white-space sensitivity, optional semicolons) are supported. A GLR parser-generator is assumed. Some restrictions that could be enforced syntactically (e.g., which modifiers are allowed in which contexts) are instead assumed to be checked in a later phase of the compiler, and some language features are not yet implemented.

```
compilation_unit
    -> w api w
    -> w component w
    -> imports_opt
   exports_opt
   defs_opt w

api
    -> "api" w
      dotted
      imports_opt
      decls_opt
      w "end"

component
    -> "component" w
        dotted
        imports_opt
        exports_opt
        defs_opt w
        "end"

dotteds
    -> dotted
    -> dotted w "," w dotteds

dotted
    -> build_dotted

build_dotted
```

```
    -> IDENTIFIER
    -> IDENTIFIER "." build_dotted

imports_opt
    ->
    -> w imports

imports
    -> import
    -> import wr imports

import
    -> "import" wr aliased_names
    -> "import" wr import_ids

import_ids
    -> "*" wr "from" wr dotted
    -> "{" w ids w "}" wr "from" wr dotted

ids
    -> id
    -> id w "," w ids

id
    -> IDENTIFIER

aliased_names
    -> aliased_name
    -> aliased_name w "," w aliased_names

aliased_name
    -> dotted
    -> dotted wr "as" wr dotted

exports_opt
    ->
    -> w exports

exports
    -> "export" wr dotteds
    -> "export" wr dotteds w exports

decls_opt
    ->
    -> w decls

decls
    -> decl
    -> decl br decls

decl
    -> trait_decl
```

```
    -> fn_decl
    -> object_decl
    -> var_decl
    -> def_or_decl

defs_opt
    ->
    -> w defs

defs
    -> def
    -> def br defs

def
    -> trait_def
    -> fn_def
    -> object_def
    -> var_def
    -> def_or_decl

var_decl
    -> mods_opt id w is_type

var_def
    -> mods_opt id is_type_opt w "=" w no_newline_expr
    -> mods_opt id w ":" w type_ref w ":=" w no_newline_expr

is_type_opt
    ->
    -> w is_type

is_type
    -> ":" w type_ref

type_ref
    -> arg_type w "->" w ret_type throws_opt
    -> simple_type_ref

simple_type_ref
    -> "(" ")"
    -> dotted
    -> simple_type_ref w "[\" w type_args w "\]"
    -> type_ref w "[" w array_indices w "]"
    -> type_ref "^" nat_type
    -> type_ref "^" "(" w extent_range_minus_nat w ")"
    -> type_ref "^" "(" w extent_range w "BY" w extent_range w ")"
    -> type_ref w "..."
    -> "[" w type_ref w "|->" w type_ref w "]"
    -> "unity"
    -> type_ref sr type_ref
    -> type_ref w div w type_ref
    -> "(" w type_ref w ")"
```

```
fn_decl
    -> mods_opt
    fn_header
    is_ret_type_opt
    throws_opt
    where_opt
    contract_opt

fn_def
    -> mods_opt
    fn_header
    is_ret_type_opt
    throws_opt
    where_opt
    contract_opt
    w "=" w no_newline_expr

fn_header
    -> id
    type_params_opt
    w params
    -> id "." id
    type_params_opt
    w params
    -> "opr" w op
    type_params_opt
    w params
    -> "opr"
    type_params_opt
    w params
    w op
    -> "opr"
    type_params_opt
    w left_op_or_enc comma_sep_params_opt w right_op_or_enc
    -> "opr"
    type_params_opt
    w "[" w comma_sep_params w "]"
    w ":=" w "(" w param w ")"
    -> "opr"
    type_params_opt
    w "[" w comma_sep_params w "]"

type_params_opt
    ->
    -> w "[\" w comma_sep_type_params w "\]"

comma_sep_type_params
    -> type_param
    -> type_param w "," w comma_sep_type_params

type_param
```

```
    -> id extends_opt absorbs_opt
    -> "nat" w id
    -> "bool" w id
    -> "dim" w id
    -> "unit" w id absorbs_opt
    -> "opr" w op
    -> "ident" w id

absorbs_opt
    ->
    -> w "absorbs" w "unit"

throws_opt
    ->
    -> w "throws" w type_ref
    -> w "throws" w type_ref_list

where_opt
    ->
    -> w where

where
    -> "where" w "{" w  wcl: where_clause_list w "}"

where_clause_list
    -> where_clause w "," w where_clause_list
    -> where_clause

where_clause
    -> type_alias
    -> id w extends

contract_opt
    -> requires_opt ensures_opt invariant_opt

requires_opt
    ->
    -> w "requires" w "{" w comma_sep_exprs w "}"

ensures_opt
    ->
    -> w "ensures" w
  "{" prefix_ensures_clauses w ensures_clause w "}"

prefix_ensures_clauses
    ->
    -> w provided_clause w "," prefix_ensures_clauses

provided_clause
    -> comma_sep_exprs w "provided" w expr

ensures_clause
```

```
    -> provided_clause
    -> comma_sep_exprs

invariant_opt
    ->
    -> w "invariant" w "{" w comma_sep_exprs w "}"

trait_decl
    -> mods_opt
    "trait" w
    id
    type_params_opt
    extends_opt
    excludes_opt
    comprises_opt
    where_opt
    fn_decls_or_property_opt
    w "end"

trait_def
    -> mods_opt
    "trait" w
    id
    type_params_opt
    extends_opt
    excludes_opt
    comprises_opt
    where_opt
    fn_def_or_decls_or_property_opt
    w "end"

extends_opt
    ->
    -> w extends

extends
    -> "extends" w type_ref
    -> "extends" w type_ref_list_nonempty

type_ref_list
    -> type_ref_list_nonempty
    -> "{" w "}"

type_ref_list_nonempty
    -> "{" w type_refs w "}"

type_refs
    -> type_ref
    -> type_ref w "," w type_refs

excludes_opt
    ->
```

```
    -> w "excludes" w type_ref_list

comprises_opt
    ->
    -> w "comprises" w type_ref
    -> w "comprises" w "{" w "}"
    -> w "comprises" w "{" w type_refs w "}"

fn_def_or_decls_or_property_opt
    ->
    -> w fn_def_or_decls_or_property

fn_def_or_decls_or_property
    -> fn_def_or_decl_or_property
    -> fn_def_or_decl_or_property br fn_def_or_decls_or_property

fn_def_or_decl_or_property
    -> fn_def
    -> fn_decl
    -> property

fn_decls_or_property_opt
    ->
    -> w fn_decls_or_property

fn_decls_or_property
    -> fn_decl_or_property
    -> fn_decl_or_property br fn_decls_or_property

fn_decl_or_property
    -> fn_decl
    -> property

fn_decls_opt
    ->
    -> w fn_decls

fn_decls
    -> fn_decl
    -> fn_decl br fn_decls

object_decl
    -> mods_opt
  "object" w
  id
  type_params_opt
  params_opt
  extends_opt
  throws_opt
  where_opt
  contract_opt
  obj_decl_body_opt
```

```
  w "end"

object_def
    -> mods_opt
  "object" w
  id
  type_params_opt
  params_opt
  extends_opt
  throws_opt
  where_opt
  contract_opt
  obj_body_opt
  w "end"

params_opt
    ->
    -> w params

params
    -> "(" w ")"
    -> "(" w comma_sep_params w ")"

comma_sep_params_opt
    ->
    -> w comma_sep_params

comma_sep_params
    -> param
    -> param w "," w comma_sep_params

param
    -> mods_opt id is_type_opt default_value_opt

obj_body_opt
    ->
    -> w obj_body

obj_body
    -> obj_body_elem
    -> obj_body_elem br obj_body

obj_body_elem
    -> fn_def
    -> var_def

obj_decl_body_opt
    ->
    -> w obj_decl_body

obj_decl_body
    -> obj_decl_body_elem
```

```
    -> obj_decl_body_elem br obj_decl_body

obj_decl_body_elem
    -> fn_decl
    -> var_decl
    -> property

op_expr
    -> op_expr_result

op_expr_result
    -> opexpr_list

opexpr_list
    -> op_expr_no_enc
    -> enc op_expr_no_enc
    -> enc

op_expr_no_enc
    -> juxt_component
    -> juxt_component wr expr_follows_expr_w
    -> juxt_component op_follows_expr
    -> juxt_component wr op_follows_expr_w
    -> juxt_component enc_follows_expr
    -> juxt_component wr enc_follows_expr_w
    -> op
    -> op expr_follows_op
    -> op wr expr_follows_op_w
    -> op op_follows_op
    -> op wr op_follows_op_w
    -> op enc_follows_op
    -> op wr enc_follows_op_w

enc_follows_expr
    -> enc expr_follows_op
    -> enc wr expr_follows_expr_w
    -> enc op_follows_op
    -> enc wr op_follows_expr_w
    -> enc wr enc_follows_expr_w
    -> enc

enc_follows_expr_w
    -> enc op_expr_no_enc
    -> enc wr expr_follows_op_w
    -> enc wr op_follows_op_w
    -> enc wr enc_follows_op_w

enc_follows_op
    -> enc op_expr_no_enc

enc_follows_op_w
    -> enc op_expr_no_enc
```

```
expr_follows_expr_w
    -> juxt_component
    -> juxt_component op_follows_expr
    -> juxt_component wr op_follows_expr_w
    -> juxt_component wr expr_follows_expr_w
    -> juxt_component enc_follows_expr
    -> juxt_component wr enc_follows_expr_w

expr_follows_op
    -> juxt_component
    -> juxt_component op_follows_expr
    -> juxt_component wr op_follows_expr_w
    -> juxt_component wr expr_follows_expr_w
    -> juxt_component enc_follows_expr
    -> juxt_component wr enc_follows_expr_w

expr_follows_op_w
    -> juxt_component
    -> juxt_component op_follows_expr
    -> juxt_component wr op_follows_expr_w
    -> juxt_component wr expr_follows_expr_w
    -> juxt_component enc_follows_expr
    -> juxt_component wr enc_follows_expr_w

op_follows_op
    -> op wr expr_follows_op_w
    -> op expr_follows_op
    -> op wr op_follows_op_w
    -> op op_follows_op
    -> op enc_follows_op
    -> op wr enc_follows_op_w

op_follows_op_w
    -> op wr expr_follows_op_w
    -> op expr_follows_op
    -> op wr op_follows_op_w
    -> op op_follows_op
    -> op enc_follows_op
    -> op wr enc_follows_op_w

op_follows_expr
    -> op wr expr_follows_op_w
    -> op expr_follows_op
    -> op wr op_follows_op_w
    -> op op_follows_op
    -> op
    -> op enc_follows_op
    -> op wr enc_follows_op_w

op_follows_expr_w
    -> op wr expr_follows_op_w
```

```
    -> op wr op_follows_op_w
    -> op wr enc_follows_op_w

no_newline_op_expr
    -> no_newline_op_expr_result

no_newline_op_expr_result
    -> no_newline_opexpr_list

no_newline_opexpr_list
    -> no_newline_op_expr_no_enc
    -> enc no_newline_op_expr_no_enc
    -> enc

no_newline_op_expr_no_enc
    -> juxt_component
    -> juxt_component sr no_newline_expr_follows_expr_w
    -> juxt_component no_newline_op_follows_expr
    -> juxt_component sr no_newline_op_follows_expr_w
    -> juxt_component no_newline_enc_follows_expr
    -> juxt_component sr no_newline_enc_follows_expr_w
    -> op
    -> op no_newline_expr_follows_op
    -> op sr no_newline_expr_follows_op_w
    -> op no_newline_op_follows_op
    -> op sr no_newline_op_follows_op_w
    -> op no_newline_enc_follows_op
    -> op sr no_newline_enc_follows_op_w

no_newline_enc_follows_expr
    -> enc no_newline_expr_follows_op
    -> enc sr no_newline_expr_follows_expr_w
    -> enc no_newline_op_follows_op
    -> enc sr no_newline_op_follows_expr_w
    -> enc sr no_newline_enc_follows_expr_w
    -> enc

no_newline_enc_follows_expr_w
    -> enc no_newline_op_expr_no_enc
    -> enc sr no_newline_expr_follows_op_w
    -> enc sr no_newline_op_follows_op_w
    -> enc sr no_newline_enc_follows_op_w

no_newline_enc_follows_op
    -> enc no_newline_op_expr_no_enc

no_newline_enc_follows_op_w
    -> enc no_newline_op_expr_no_enc

no_newline_expr_follows_expr_w
    -> juxt_component
    -> juxt_component no_newline_op_follows_expr
```

```
    -> juxt_component sr no_newline_op_follows_expr_w
    -> juxt_component sr no_newline_expr_follows_expr_w
    -> juxt_component no_newline_enc_follows_expr
    -> juxt_component sr no_newline_enc_follows_expr_w

no_newline_expr_follows_op
    -> juxt_component
    -> juxt_component no_newline_op_follows_expr
    -> juxt_component sr no_newline_op_follows_expr_w
    -> juxt_component sr no_newline_expr_follows_expr_w
    -> juxt_component no_newline_enc_follows_expr
    -> juxt_component sr no_newline_enc_follows_expr_w

no_newline_expr_follows_op_w
    -> juxt_component
    -> juxt_component no_newline_op_follows_expr
    -> juxt_component sr no_newline_op_follows_expr_w
    -> juxt_component sr no_newline_expr_follows_expr_w
    -> juxt_component no_newline_enc_follows_expr
    -> juxt_component sr no_newline_enc_follows_expr_w

no_newline_op_follows_op
    -> op sr no_newline_expr_follows_op_w
    -> op no_newline_expr_follows_op
    -> op sr no_newline_op_follows_op_w
    -> op no_newline_op_follows_op
    -> op no_newline_enc_follows_op
    -> op sr no_newline_enc_follows_op_w

no_newline_op_follows_op_w
    -> op sr no_newline_expr_follows_op_w
    -> op no_newline_expr_follows_op
    -> op sr no_newline_op_follows_op_w
    -> op no_newline_op_follows_op
    -> op no_newline_enc_follows_op
    -> op sr no_newline_enc_follows_op_w

no_newline_op_follows_expr
    -> op sr no_newline_expr_follows_op_w
    -> op no_newline_expr_follows_op
    -> op sr no_newline_op_follows_op_w
    -> op no_newline_op_follows_op
    -> op
    -> op no_newline_enc_follows_op
    -> op sr no_newline_enc_follows_op_w

no_newline_op_follows_expr_w
    -> op wr no_newline_expr_follows_op_w
    -> op wr no_newline_op_follows_op_w
    -> op sr no_newline_enc_follows_op_w

no_space_op_expr
```

```
    -> no_space_op_expr_result

no_space_op_expr_result
    -> no_space_opexpr_list

no_space_opexpr_list
    -> no_space_op_expr_no_enc
    -> enc no_space_op_expr_no_enc
    -> enc

no_space_op_expr_no_enc
    -> juxt_component
    -> juxt_component no_space_op_follows_expr
    -> juxt_component no_space_enc_follows_expr
    -> op
    -> op no_space_expr_follows_op
    -> op no_space_op_follows_op
    -> op no_space_enc_follows_op

no_space_enc_follows_expr
    -> enc no_space_expr_follows_op
    -> enc no_space_op_follows_op
    -> enc

no_space_enc_follows_op
    -> enc no_space_op_expr_no_enc

no_space_expr_follows_op
    -> juxt_component
    -> juxt_component no_space_op_follows_expr
    -> juxt_component no_space_enc_follows_expr

no_space_op_follows_op
    -> op no_space_expr_follows_op
    -> op no_space_op_follows_op
    -> op no_space_enc_follows_op

no_space_op_follows_expr
    -> op no_space_expr_follows_op
    -> op no_space_op_follows_op
    -> op
    -> op no_space_enc_follows_op

expr
    -> op_expr
    -> tuple_expr
    -> flow_expr
    -> fn_expr
    -> object_expr
    -> assignment_expr
    -> type_ascription_expr
```

```
no_newline_expr
    -> no_newline_op_expr
    -> tuple_expr
    -> no_newline_flow_expr
    -> no_newline_fn_expr
    -> object_expr
    -> no_newline_assignment_expr
    -> no_newline_type_ascription_expr

no_space_expr
    -> no_space_op_expr
    -> tuple_expr
    -> no_space_flow_expr
    -> object_expr
    -> no_space_assignment_expr

type_ascription_expr
    -> expr w "as" w type_ref

no_newline_type_ascription_expr
    -> no_newline_expr s "as" w type_ref

asif_expr
    -> expr w "asif" w type_ref

no_newline_asif_expr
    -> no_newline_expr s "asif" w type_ref

no_newline_atomic_expr
    -> "atomic" w no_newline_expr

atomic_expr
    -> "atomic" w expr

no_newline_tryatomic_expr
    -> "tryatomic" w no_newline_expr

tryatomic_expr
    -> "tryatomic" w expr

no_newline_throw_expr
    -> "throw" w no_newline_expr

throw_expr
    -> "throw" w expr

no_newline_exit_expr
    -> "exit" id_opt no_newline_with_opt

exit_expr
    -> "exit" id_opt with_opt
```

```
id_opt
    ->
    -> w id

with_opt
    ->
    -> w "with" w expr

no_newline_with_opt
    ->
    -> w "with" w no_newline_expr

tuple_expr
    -> "(" w expr w "," w comma_sep_exprs w ")"

object_expr
    -> "object" extends_opt obj_body_opt w "end"

no_newline_fn_expr
    -> "fn" w params is_ret_type_opt throws_opt w "=>" w
      no_newline_expr

fn_expr
    -> "fn" w params is_ret_type_opt throws_opt w "=>" w expr

no_newline_accumulator
    -> "SUM" w "[" w generators w "]" w no_newline_expr
    -> "PRODUCT" w "[" w generators w "]" w no_newline_expr
    -> "BIG" w op_or_enc  w "[" w generators w "]" w no_newline_expr

accumulator
    -> "SUM" w "[" w generators w "]" w expr
    -> "PRODUCT" w "[" w generators w "]" w expr
    -> "BIG" w op_or_enc  w "[" w generators w "]" w expr

no_space_assignment_expr
    -> no_space_expr ":=" no_space_expr
    -> no_space_expr assign_op no_space_expr

no_newline_assignment_expr
    -> no_newline_expr s ":=" w no_newline_expr
    -> no_newline_expr s assign_op w no_newline_expr

assignment_expr
    -> expr w ":=" w expr
    -> expr w assign_op w expr

let_expr
    -> let_mutable
    -> let_immutable
    -> let_fun
```

```
let_fun
    -> let_fun_list

let_fun_list
    -> fn_def
    -> fn_def br let_fun_list

let_mutable
    -> "var" w lvals
    -> "var" w lvals s "=" w no_newline_expr
    -> "var" w lvals s ":=" w no_newline_expr
    -> typed_lvals s ":=" w no_newline_expr

let_immutable
    -> typed_lvals
    -> lvals s "=" w no_newline_expr

tuple_lvals
    -> "(" w ids w ")" s ":" s type_ref
    -> "(" w ids w ")" s ":" s "(" w type_refs w ")"

lvals
    -> lval
    -> "(" w comma_sep_lvals w ")"
    -> tuple_lvals

typed_lvals
    -> typed_lval
    -> "(" w comma_sep_typed_lvals w ")"
    -> tuple_lvals

comma_sep_lvals
    -> lval
    -> lval w "," w comma_sep_lvals

comma_sep_typed_lvals
    -> typed_lval
    -> typed_lval w "," w comma_sep_typed_lvals

lval
    -> typed_lval
    -> id
    -> unpasting

typed_lval
    -> id s ":" s type_ref

unpasting
    -> "[" w unpasting_elems w "]"

unpasting_elems
    -> unpasting_elem
```

```
    -> unpasting_elem rect_separator unpasting_elems

unpasting_elem
    -> unpasting
    -> id
    -> id s ":" w "[" w unpasting_dim w "]"

unpasting_dim
    -> extent_range w "BY" w extent_range
    -> extent_range w "BY" w unpasting_dim

comma_sep_exprs_opt
    ->
    -> w comma_sep_exprs

comma_sep_exprs
    -> expr
    -> expr w "," w comma_sep_exprs

juxt_component
    -> primary

exponentiation
    -> primary exp exponent
    -> primary exp_op

exponent
    -> id
    -> literal
    -> parenthesized

primary
    -> base_expr
    -> type_application
    -> bracket_expr
    -> tight_juxtaposition
    -> field_selection
    -> exponentiation

type_application
    -> primary "[\" w type_args w "\]"

bracket_expr
    -> primary "[" comma_sep_exprs_opt w "]"

tight_juxtaposition
#Ambiguity -- favor KeywordsExpr parses
    -> primary "(" w ")"
    -> primary "(" w expr w ")"
    -> primary tuple_expr
    -> primary keyword_args
```

```
args_opt
    ->
    -> w comma_sep_exprs w ","

keyword_args
# Ambiguity -- favor parses with largest number of keywords
    -> "(" args_opt w comma_sep_keywords w ")"

comma_sep_keywords
    -> id w "=" w expr
    -> id w "=" w expr w "," w comma_sep_keywords

field_selection
    -> primary "." id

base_expr
    -> parenthesized
    -> matched_enclosing_operator
    -> id
    -> base_value_expr
    -> comprehension

matched_enclosing_operator
    -> not_yet_matched_enclosing_operator

not_yet_matched_enclosing_operator
    -> left_op comma_sep_exprs_opt w right_op

left_op_or_enc
    -> enc
    -> left_op

right_op_or_enc
    -> enc
    -> right_op

left_op
    -> left_op_literal

right_op
    -> right_op_literal

left_op_literal
    -> "LC"
    -> "LF"
    -> "<|"
    -> "{"

right_op_literal
    -> "RC"
    -> "RF"
    -> "|>"
```

```
    -> "}"

comprehension
    -> set_comprehension
    -> list_comprehension
    -> map_comprehension
    -> rect_comprehension

comprehension_rhs
    -> expr
    -> generator
    -> expr w "," w comprehension_rhs
    -> generator w "," w comprehension_rhs

set_comprehension
    -> "{" w expr wr "|" wr comprehension_rhs w "}"

list_comprehension
    -> "<|" w expr wr "|" wr comprehension_rhs w "|>"

map_comprehension
    -> "[" w expr w "|->" w expr wr "|" wr comprehension_rhs w "]"

rect_comprehension
    -> "[" w rect_comp_clauses w "]"

rect_comp_clauses
    -> rect_comp_clause
    -> rect_comp_clause br rect_comp_clauses

rect_comp_clause
    -> "(" w comma_sep_exprs w ")" w "=" w expr wr
     "|" wr comprehension_rhs

parenthesized
    -> "(" w expr w ")"

base_value_expr
    -> literal

literal
    -> "(" w ")"
    -> INT
    -> FLOAT
    -> STRING
    -> CHAR
    -> map_expr
    -> rect_expr

rect_expr
    -> "[" w rect_elements w "]"
```

```
rect_elements
    -> no_space_expr
    -> no_space_expr rect_separator rect_elements

rect_separator
    -> sr
    -> nl
    -> w semicolons w

semicolons
    -> ";"
    -> ";" semicolons

map_expr
    -> "[" w comma_sep_entries w "]"

comma_sep_entries
    -> entry
    -> entry w "," w comma_sep_entries

entry
    -> expr w "|->" w expr

no_space_flow_expr
    -> label_expr
    -> do_expr
    -> for_expr
    -> spawn_expr
    -> if_expr
    -> try_expr
    -> case_expr
    -> type_case_expr
    -> while_expr

no_newline_flow_expr
    -> label_expr
    -> do_expr
    -> for_expr
    -> spawn_expr
    -> if_expr
    -> try_expr
    -> case_expr
    -> type_case_expr
    -> while_expr
    -> no_newline_accumulator
    -> no_newline_atomic_expr
    -> no_newline_tryatomic_expr
    -> no_newline_throw_expr
    -> no_newline_exit_expr

flow_expr
    -> label_expr
```

```
    -> do_expr
    -> for_expr
    -> spawn_expr
    -> if_expr
    -> try_expr
    -> case_expr
    -> type_case_expr
    -> while_expr
    -> accumulator
    -> atomic_expr
    -> tryatomic_expr
    -> throw_expr
    -> exit_expr

label_expr
    -> "label" w id w exprs w "end" w id

while_expr
    -> "while" w expr w do_expr

type_case_expr
    -> "typecase" w type_case

type_case
    -> type_case_bindings w "in" w type_case_clauses w "end"
    -> type_case_bindings w "in" w type_case_clauses br type_case_else w "end"

type_case_else
    -> "else" w "=>" w exprs

type_case_type_refs
    -> type_ref
    -> "(" w type_refs w ")"

type_clause
    -> type_case_type_refs w "=>" w exprs

type_case_clauses
    -> type_clause
    -> type_clause br type_case_clauses

type_case_bindings
    -> id
    -> bindings

bindings
    -> binding
    -> "(" w comma_sep_bindings w ")"

comma_sep_bindings
    -> binding
    -> binding w "," w comma_sep_bindings
```

```
binding
    -> id w "=" w expr

case_clauses
    -> case_clause
    -> case_clause br case_clauses

case_clause
    -> no_newline_expr w "=>" w exprs

case_expr
    -> "case" w case_prefix w "of" w case_suffix w "end"

case_prefix
    -> expr
    -> expr w op
    -> "largest"
    -> "smallest"

case_suffix
    -> case_clauses
    -> case_clauses w "else" w "=>" w exprs

try_expr
    -> "try" w exprs catch_opt forbid_opt finally_opt w "end"

catch_opt
    ->
    -> w "catch" w id w catch_clauses

forbid_opt
    ->
    -> w "forbid" w type_ref_list

finally_opt
    ->
    -> w "finally" w expr

catch_clauses
    -> type_ref w "=>" w exprs
    -> type_ref w "=>" w exprs br catch_clauses

if_expr
    -> "if" w expr w "then" w exprs w "end"
    -> "if" w expr w "then" w exprs w "else" w exprs w "end"
    -> "if" w expr w "then" w exprs w else_clauses w "else" w exprs w "end"
    -> "if" w expr w "then" w exprs w else_clauses w "end"

else_clauses
    -> else_clause
    -> else_clause w else_clauses
```

```
else_clause
    -> "elif" w expr w "then" w exprs

spawn_expr
    -> "spawn" expr_opt w do_expr

expr_opt
    ->
    -> w expr

for_expr
    -> "for" w generators w do_expr

no_newline_generators
    -> no_newline_generator
    -> no_newline_generator s "," w no_newline_generators

generators
    -> generator
    -> generator w "," w generators

no_newline_generator
    -> ids w "<-" w no_newline_expr

generator
    -> ids w "<-" w expr

do_expr
    -> "do" w exprs w "end"
    -> "do" w "end"

block_elem
    -> no_newline_expr
    -> let_expr
    -> no_newline_expr s "," w no_newline_generators

exprs
# Ambiguity -- favor LetExpr parses
    -> block_elem
    -> block_elem w ";"
    -> block_elem br exprs

default_value_opt
    ->
    -> w "=" w expr

is_ret_type_opt
    ->
    -> w ":" w ret_type

ret_type
```

```
    -> type_ref
    -> "(" w type_ref w "," w type_refs w ")"

arg_type
    -> type_ref
    -> named_arg
    -> "(" w arg_types w ")"

arg_types
    -> type_ref w "," w type_ref
    -> named_arg
    -> type_ref w "," w arg_types
    -> named_arg w "," w named_arg_types

named_arg_types
    -> named_arg
    -> named_arg w "," w named_arg_types

named_arg
    -> id w ":" w type_ref

array_indices
    -> array_extent_ranges

array_extent_ranges
    -> extent_range
    -> extent_range w "," w array_extent_ranges

extent_range
    -> nat_type
    -> extent_range_minus_nat

extent_range_minus_nat
    -> "#"
    -> nat_type w "#" w nat_type
    -> nat_type w "#"
    -> "#" w nat_type

type_args
    -> type_arg
    -> type_arg w "," w type_args

type_arg
    -> type_ref
    -> nat_type
    -> op

nat_type
    -> INT
    -> "-" INT
    -> id
    -> "(" w nat_type w ")"
```

```
    -> nat_type w "-" w INT
    -> nat_type sr nat_type
    -> nat_type w "+"  w nat_type

bool_type
    -> id
    -> "(" w bool_type w ")"
    -> bool_type w "AND" w bool_type
    -> bool_type w "OR"  w bool_type

mods_opt
    ->
    -> mods w

mods
    -> modifier
    -> modifier wr mods

space_sep_ids
    -> id
    -> id wr space_sep_ids

assign_op
    -> ASSIGNMENT_OPERATOR

exp_op
    -> EXPONENT_OPERATOR

op
    -> op_literal
    -> div

exp
    -> "^"

div
    -> "/"

op_literal
    -> "#"
    -> ":"
    -> "="
    -> "*"
    -> "+"
    -> "-"
    -> "BY"
    -> "<"
    -> "<="
    -> ">"
    -> ">="
    -> "in"
    -> OPERATOR
```

```
enc
    -> enc_literal

op_or_enc
    -> op
    -> enc

enc_literal
    -> "|"
    -> "||"
    -> "//"

def_or_decl
    -> "dim" w id equal_ty_opt default_unit_opt
    -> "dim" w id equal_ty_opt s unit_var
    -> unit_var
    -> type_alias
    -> test
    -> property

unit_var
    -> unit_keyword w space_sep_ids is_type_opt eq_expr_opt

eq_expr_opt
    ->
    -> w "=" w expr

unit_keyword
    -> "unit"
    -> "si_unit"

equal_ty_opt
    ->
    -> w "=" w type_ref

default_unit_opt
    ->
    -> w "default" w type_ref

type_alias
    -> "type" w id w "[\" w ids w "\]" w "=" w type_ref
    -> "type" w id w "=" w type_ref

test
    -> "test" w id w "[" w generators w "]" w "=" w expr

property
    -> "property" w id w "=" w "FORALL" w params w expr
    -> "property" w "FORALL" w params w expr
    -> "property" w id w "=" w w expr
    -> "property" w expr
```

```
modifier
    -> "abstract"
    -> "atomic"
    -> "getter"
    -> "hidden"
    -> "io"
    -> "private"
    -> "pure"
    -> "settable"
    -> "setter"
    -> "static"
    -> "test"
    -> "transient"
    -> "value"
    -> "var"
    -> "wrapped"

w    # Whitespace Optional
    ->
    -> wr

wr   # Whitespace Required
    -> TOK_WHITESPACE w
    -> TOK_NEWLINE w

s    # Space Optional
    ->
    -> sr

sr   # Space Required
    -> TOK_WHITESPACE s

nl   # Required Newline embedded in whitespace
    -> s TOK_NEWLINE w

br   # Line break
    -> nl
    -> s ";" w
```

Bibliography

[1] O. Agesen, L. Bak, C. Chambers, B.-W. Chang, U. Hlzle, J. Maloney, R. B. Smith, D. Ungar, and M. Wolczko. *The Self Programmer's Reference Manual*. `http://research.sun.com/self/release_4.0/Self-4.0/manuals/Self-4.1-Pgmers-Ref.pdf`, 2000.

[2] E. Allen, V. Luchangco, and S. Tobin-Hochstadt. Encapsulated Upgradable Components, Mar. 2005.

[3] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, Nov. 1994.

[4] R. D. Blumofe, C. F. Joerg, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 132–141, Montreal, Canada, 17–19 June 1998. ACM, SIGPLAN Notices.

[5] G. Bracha, G. Steele, B. Joy, and J. Gosling. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[6] R. Cartwright and G. Steele. Compatible genericity with run-time types for the Java Programming Language. In *OOPSLA*, 1998.

[7] W. Clinger. Macros that work. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 155–162. ACM Press, 1991.

[8] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.

[9] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *Journal of LISP and Symbolic Computation*, 5(4):295–326, 1992.

[10] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 229–236. ACM Press, September 2001.

[11] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1), Aug. 1996.

[12] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[13] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[14] X. Leroy, D. Doligez, J. Garrigue, D. Rmy, and J. Vouillon. *The Objective Caml System, release 3.08*. `http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf`, 2004.

[15] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A Visual Environment for Developing Context-Sensitive Term Rewriting Systems (system description). In V. van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA-04*, LNCS 3091, pages 301–311, Valencia, Spain, June 3-5, 2004. Springer.

[16] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[17] T. Millstein and C. Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, May 2002.

[18] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[19] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. Technical Report TM-449, MIT/LCS, 1991.

[20] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Mar. 1966.

[21] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *The Scala Language Specification*. `http://scala.epfl.ch/docu/files/ScalaReference.pdf`, 2004.

[22] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface Version 2.0*. `http://www.openmp.org/specs/mp-documents/fspec20_bars.pdf`, Nov. 2000.

[23] S. Peyton-Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[24] B. N. Taylor. Guide for the use of the international system of units (si). Technical report, United States Department of Commerce, National Institute of Standards and Technology, Apr. 1995.

[25] The Unicode Consortium. *The Unicode Standard, Version 4.0*. Addison-Wesley, 2003.

[26] Java(TM) 2 Platform Standard Edition 6.0 API Specification. `http://download.java.net/jdk6/docs/api/index.h`