THE REALIZATION OF DATA ABSTRACTIONS IN CHILL

Jurgen F.H Winkler
Siemens AG Munich
Federal Republic of Germany

ABSTRACT

The paper presents several realizations of data abstractions in CHILL: abstract object, abstract type, and polymorphic abstract type. Due to limitations in the actual version of CHILL two extensions are proposed: polymorphic types in the form of generally parameterized types, and assertions with expressions of the predicate calculus as their body. These generalized assertions permit the definition of the semantics of data abstractions in a direct manner.

1 INTRODUCTION

Abstraction and modularization are the two most important principles of the discipline of program construction (software engineering). The application of abstraction is necessary in order to reduce the complexity of the system under construction to such a degree that humans are able to comprehend them. A second method to achieve comprehensibility is the decomposition of the whole system into subsystems (strategy of divide and conquer). Modularization is the technical tool to achieve abstraction and decomposition. The application of abstraction and modularization results in a program structure that may be characterized as follows:

a) the program consists of an arbitrary number of textually self-contained program units (PUs);

b) a PU consists essentially of two parts: the interface (or specification part) to the outside world (external PUs), and the realization (or implementation, or inner) part, in which the entities specified in the interface are realized;

c) a PU pi may refer to another PU p2 by using some entity defined in the interface of p2;

d) a reduction of complexity is achieved because the interfaces are generally simpler than the realization parts. (The interfaces are essentially the 'backbone' of a (program) system.)

With respect to the contents we may distinguish different kinds of PUs: subprogram, process, pure data module, general data module, and abstract data type module.

PUs containing some local data (base) are the most typical building blocks of modularized programs. Examples for such PUs in existing programming languages are module and region in CHILL [ITU 81] and package in Ada [Ref 83]. Process (CHILL) and task (Ada) are concurrent realizations of such PUs.

Syntactically a general PU (GPU) is a collection of definitions of arbitrary entities (modes, locations, subprograms etc.). Such a pure grouping of definitions is not sufficient for obtaining GPUs with well defined and well adapted interface and body. Together with the partition into interface and body the grouping allows the realization of information hiding [Par 72a] in such a way that the user of a GPU can manipulate the local data base of the GPU only by using the entities defined in the interface of the GPU, i.e. only in such a manner explicitly allowed by the designer of the GPU. In this paper such a unique GPU is called an abstract object.

The theoretical aspects of such abstract objects are discussed in the field of abstract data types. In this field a lot of research has been carried through during the last ten years [Kle 83; KL 83].

Two problems arise in connection with the realization of GPUs in programming languages (esp. in CHILL):

P1: the realization of families of similar or identical abstract objects. To achieve this some sort of mode constructor for such abstract objects is necessary.

P2: the formal definition of the semantics of operations defined in the specification part of modules. This may be achieved by general assertions.

Section 2 of this paper deals with the realization of abstract objects and families of abstract objects in CHILL-84 [CIS 83]. The main result of this section is that the problems P1 and P2 cannot be completely solved by using CHILL-84. In section 3 we propose some enhancements of CHILL which solve the problems P1 and P2.

In the programming examples throughout the paper we use some constructs of CHILL which are contained in CHILL-84 [CIS 83] for the first time. These constructs are: prefixed names, renaming in GRANT and SETE statements, and dynamic allocation via ALLOCATE and TERMINATE. Further information on these constructs is given in the reference mentioned above.

2 DATA ABSTRACTIONS IN CHILL-84

2.1 ABSTRACT OBJECTS

The simplest form of data abstraction is the abstract object, which consists of a unique data base and a set of entities for the manipulation of this data base. A typical
example is a stack of elements of an arbitrary mode. As an abstract object this stack may be characterized as a sequence of elements, where this sequence may be manipulated according to the LIFO protocol. In most cases all elements of such a stack have the same mode.

In this section we show the realization of an abstract stack object in CHILL. The stack is realized as a module "stack_obj". In case of concurrent access a region instead of a module should be used.

Fig. 1 shows the definition of an abstract stack object named "stack_obj" as a CHILL module. The module is parameterized in such a way that the length and the mode of the elements of the stack, which is defined inside of "stack_obj", may be delivered by the user outside of this module.

Fig. 2 shows a potential use of the stack "stack_obj". The language rules of CHILL guarantee that the user is not able to manipulate the variables "stack" and "stack_index", which are only known inside of "stack_obj", directly. The user can only use the subprograms "push", "pop", "top", as indicated by the grant statements at the end of "stack_obj", and furthermore the exceptions defined inside of "stack_obj".

Thus, "stack_obj" is a robust and safe encapsulation of the data case consisting of "stack" and "stack_index".

```
stack_obj : MODULE

SEIZE elem_mode, Length;

SYN min = 1, min_1 = min-1,
max = min_1 + length;
DCL stack ARRAY[min : max] elem_mode,
stack_index INT[min_1 : max];
INIT := min_1;
push : PROC(elem_mode)
EXCEPTIONS(overflow);
CASE stack_index < max OF
  (TRUE) : stack_index := i;
  stack(stack_index) := e;
  RETURN;
  (FALSE) : CAUSE overflow;
  ESAC;
END push;

pop : PROC() EXCEPTIONS(underflow);
CASE stack_index > min_1 OF
  (TRUE) : stack_index := i;
  RETURN;
  (FALSE) : CAUSE underflow;
  ESAC;
END pop;

top : PROC() elem_mode
EXCEPTIONS(empty_stack);
CASE stack_index > min_1 OF
  (TRUE) : RETURN stack(stack_index);
  (FALSE) : CAUSE empty_stack;
  ESAC;
END top;

GRANT push, pop, top;

END stack_obj;
```

Fig. 1 Abstract stack

2.2 ABSTRACT TYPES

If we want to have several stacks similar to that defined in "stack_obj" we can define for each of these stacks one such module and name them eg "stack_obj_1", "stack_obj_2", etc.

This solution has some unpleasant properties:

a) all stacks have exactly the same characteristics (length and element mode);
b) the code for the access routines exists in multiple copies.

It is possible to vary the characteristics of the stacks contained in different modules, but the code of the corresponding access routines would still be very similar or even identical.

In order to avoid multiple copies of the stack routines our goal is to have one module (or region) "stack_mode_def" exporting a mode "stack_mode" that can be used to declare an arbitrary number of stack objects (locations in the sense of 2.202). The main problem is to guarantee that these locations of mode "stack_mode" have really the desired properties outside of "stack_mode_def". Especially only such manipulations shall be possible which are compatible with the LIFO protocol.

Fig. 3 shows the realization of such a module. The module is named "stack_mode_def" and exports the mode "stack_mode" which in this case has the required properties.

```
user : MODULE

SEIZE push, pop, top;

SYN Length = 10,000;
SYNMODE elem_mode = INT;

GRANT elem_mode, Length;
DCL e elem_mode;

push(10);
push(25);
e := 93;
pop(one empty_stack : push(0); END;
e := top;

END user;
```

Fig. 2 Use of the abstract stack
The characteristic properties of the solution depicted in Fig.3 are as follows:

a) all stacks declared using the mode "stack_mode" have the same element mode. This limitation will be dealt with in the following section of the paper.

b) all those stacks have the same number of elements. This limitation could be overcome by the use of dynamic array modes or by implementing the stacks as linked lists. A realization based on linked lists is depicted in Fig. 5.

c) all those stacks have the same structure, which in this case is an array structure. Because there is exactly one mode "stack_mode" this mode has exactly one structure. There is no language construct in CHILL which allows one mode to have different representations for different objects of this very mode [Mac 83]. But the structure of "stack_mode" could be changed, eg into a linked list, without any impairment of the user. This is depicted in Fig. 5.

d) all those stacks may only be manipulated using the following operations:

- assignment
- the granted routines "push", "pop", "top", and "initialize".

This implies that the objects of mode "stack_mode" really have the properties of stacks, ie they obey the LIFO-protocol.

The user module depicted in Fig.4 can use the stack facility defined in Fig.5 by using the following SEIZE statement:

SEIZE (flex_stack_mode_def-2) ! ALL;

Thus, only the module name in the rename clause has to be changed.

2.3 POLYMORPHIC ABSTRACT TYPES

There remains one major limitation with the solution depicted in Fig.5: all objects of mode "stack_mode" have the same element mode. Without stating it explicitly we have assumed throughout the paper that the stack objects shall be homogeneous, ie the components of one stack shall all have the same mode. But different stacks could have different modes as their element mode. We call a mode with such properties a polymorphic mode [Hor 74, Rey 83] ([Hol 78] uses the term "generic mode"), and if it is an abstract mode we call it a polymorphic abstract mode.

A general solution to this problem would be the module exporting the mode and operations for stacks (ie for the LIFO-protocol) in such a way that different stacks with different element modes may be declared using the facilities defined by this one module. Thus, this module would export a polymorphic abstract mode.
3 ENHANCEMENTS OF CHILL

3.1 PARAMETERIZED MODES

The proposal for the realization of polyomorphic modes consists of the parameterization of modes. CHILL allows already the parameterization of modes in a limited way: the upper bound of strings and arrays and the values of tag fields in structure modes may be parameters. In all these cases the values of such parameters are limited to discrete literal values.

In order to obtain polyomorphic modes the proposal also allows modes as parameters of modes. We call a mode with parameters a paramode and a mode without parameters simply a mode.

Paramodes are defined as follows:

<paramode definition statement> ::= PARMODE <paramode definition> ;
<paramode definition> ::= <identifier> ( <mode params> ) = <mode>
Neither <mode params> nor <mode> may contain an applied occurrence of the paramode being defined.

<identifier> ::= <simple name string> 
<mode params> ::= <mode param> / ( , <mode param> ) 
<mode param> ::= <identifier> List <mode param indication> 
<mode param indication> ::= VAL <mode> / MODE <mode class> 
<mode class> ::= ASSIGN / DISCRETE / FLOAT / ANY 
Further mode classes may be introduced.
<mode> ::= .... | 
<paramode name( act mode params ) > 
<act mode params> ::= <act mode param> / ( , <act mode param> ) 
<act mode param> ::= <identifier> 
(expression | <mode> | <mode name> )

Paramodes can be applied in two ways:

a) substituted: all formal parameters are substituted by appropriate actual ones. This application is one alternative of the production for <mode> (see rules above).

b) pure: no formal parameter is substituted. The application consists of a name for the paramode only. This form can be used in the specification of formal parameters of procedures and processes. In this case parameter compatibility is defined by the rule that the mode of the actual parameter must be an instance of the formal parameters paramode. Procedures and processes with such parameters are in a certain sense "generic" in that they may be applied to parameters with different modes [Mac 83]. It may be necessary to restrict this use of paramodes to the specification of first order parameters.

Fig. 5. A more flexible abstract stack type

Fig. 6 shows the realization of a module exporting a polyomorphic stack mode as a paramode.
poly_stack_mode_def: MODULE
SYNMODE Natural = RANGE(0:IntMax);
PARMODE stack_mode (Length VAL Natural, 
elen_m MODE ASSIGN) = 
STRUCT(data ARRAY(1 : Length):elem_m, 
tos RANGE(0 : Length));
push : PROC(stack stack_mode INOUT, 
elem stack.elem_m) 
EXCEPTIONS (overflow);
CASE stack.tos < stack.length OF 
(TRUE) : stack.tos := 1; 
(stack.data(stack.tos) := elem;
(FALSE) : CAUSE overflow;
ESAC;
END push;
pop : PROC(stack stack_mode INOUT) 
EXCEPTIONS (underflow);
CASE stack.tos > 0 OF 
(TRUE) : stack.tos := 1; 
(FALSE) : CAUSE underflow;
ESAC;
END pop;
top : PROC(stack stack_mode) 
RETURNS (stack.elem_m) 
EXCEPTIONS (empty_stack);
CASE stack.tos > 0 OF 
(TRUE) : RETURN stack.data 
(stack.tos);
(FALSE) : CAUSE empty_stack;
ESAC;
END top;
initialize : PROC(stack stack_mode INOUT); 
stack.tos := 0;
END initialize;
GRANT stack_mode FORBID ALL, 
Natural;
push, pop, top, initialize PREFIXED;
END poly_stack_mode_def;

Fig. 6. Polymorphic abstract stack type

An example for the use of the polymorphic abstract data type is given in Fig.7.

3.2 SEMANTICS OF DATA ABSTRACTIONS

As mentioned in the introduction a modular program consists of a set of GPUs, where the relations between these GPUs are defined via the interfaces of the GPUs. The interface of a GPU describes the behaviour of the GPU as seen by the other GPUs. If such a GPU is supposed to define a certain data abstraction the interface must define
a) the mode(s) of the abstract objects;
b) the functionality of the operations belonging to the data abstraction;
c) the behaviour of the operations (in the following this is called the semantics of the data abstraction).

user : MODULE
SEIZE (poly_stack_mode_def-) ! ALL;
/* Natural, 
stack_mode, push, pop, top, initialize */
SYNMODE int_stack_100 = stack_mode(100,INT);
PARMODE int_stack(Length VAL Natural) = 
(stack_mode(Length,Length, elem_m,INT));
/* the length of the stacks is not yet fixed */
DCL stack1 int_stack_100, 
stack2 int_stack(127), 
stack3 stack_mode(S12, BOOL);
initialize(stack1);
initialize(stack2);
initialize(stack3);
push(stack1,12);
push(stack2,100);
push(stack3,TRUE);
pop(stack1) ON empty_stack : 
push(stack1,0); END;
push(stack1, top(stack2));
END user;

Fig. 7 Use of poly_stack_mode_def

In the terminology of abstract data types a) and b) together form the signature of the type. The semantics in the sense mentioned in c) is usually defined by axioms eg in the form of equations or derivation rules. The facilities contained in CHILL and in the preceding part of this paper allow the specification of the signature of a data abstraction only but not the specification of the semantics, ie there is no guarantee that the procedures of a CHILL module implement the functions implied by their English names. The definition of the signature of a data abstraction can be given as a specification module. In Fig.8 the signature of the polymorphic abstract stack mode defined in Fig.6 is given as a specification module.

poly_stack_spec : SPEC MODULE
SYNMODE Natural = RANGE(0:IntMax);
PARMODE stack_mode (Length VAL Natural, 
elen_m MODE ASSIGN) = 
STRUCT(data ARRAY(1 : Length):elem_m, 
tos RANGE(0 : Length));
push : PROC(stack stack_mode INOUT, 
elem stack.elem_m) 
EXCEPTIONS (overflow);
pop : PROC(stack stack_mode INOUT) 
EXCEPTIONS (underflow);
top : PROC(stack stack_mode) 
STACKSTACK(stack_mode elem_m) 
EXCEPTIONS (empty_stack);
initialize : PROC(stack stack_mode INOUT);
GRANT stack_mode FORBID all, 
Natural;
push, pop, top, initialize PREFIXED;
END poly_stack_mode_def;

Fig.8 Specification of poly_stack_mode_def

179
At the time being there is no general and efficient method available for the translation of algebraic specifications into programs formulated in an imperative programming language as eg CHILL. The effect of a procedure can also be specified in a more direct way by using assertions [Par 72, Jon 80]. Since CHILL already contains an ASSERT statement the following proposal is based on this ASSERT statement.

The use of assertions for the purpose of specification has the further advantage that such assertions can be used as a starting point for the construction of a verified program using eg the method of predicate transformers [Gri 81].

As an example we take the popular problem of sorting an array of elements. Our goal is to formulate a specification module which defines syntax (signature) and semantics of a generally applicable sort facility. In order to be generally applicable the sort procedure contained in this module must be able to accept arrays with different element modes. This is possible if the parameter mode of this procedure is a paramode as introduced in the preceding section.

If we want to define the effect of the sort procedure in a convenient way using the ASSERT statement we encounter some limitations of this statement. The ASSERT statement in CHILL consists essentially of a Boolean expression, i.e. a formula of the pure propositional calculus. In this framework it is not possible to give an expression stating that an array with an arbitrary number of elements is ordered. It could be done by defining a procedure with a boolean result. But it would be more convenient to express this directly by a predicate in the ASSERT statement. In order to accomplish this we introduce the quantifiers ALL and EX with their usual meaning into the ASSERT statement.

A second problem arises if we follow the usual practice and perform the sorting in situ. In this case the parameter of the sort procedure will be an INOUT parameter and in the general case it will be modified in a certain respect inside the procedure. On the other hand only certain modifications of the parameter to be sorted are allowed. The modifications allowed are exactly those which do not change the (null) set of values contained in the array. Thus, there are more degrees of constancy as eg mentioned in [GH 81]. In order to solve this problem we introduce a predefined polymorphic function FREQ which takes two parameters: an array of a certain element mode and a value of this very element mode. It returns the number of occurrences of the given value in the given array.

In the specification of the effect of a procedure we use three different forms of assertions:

a) Pre-assertion : ASSERT PRE . . .
Such an assertion shall hold after parameter transmission and before execution of the first statement of the procedure body.

b) Post-assertion : ASSERT POST . . .
Such an assertion shall hold after execution of the last statement of the procedure body and before the execution of the next statement in the program.

c) Invariant : ASSERT . . .
Such an assertion shall hold like a pre-assertion and like a post-assertion. The expression CONST(e) as part of an invariant means that e shall deliver the very same value at pre-time and at post-time.

The specification of the general sort facility is now given in Fig. 9.

Fig. 10 shows a use of the sort facility defined in figure 8.

general_sort : SPEC MODULE

PARM mode vect_m (ind_m MODE DISCR,
elem_m MODE ASSIGN) =
ARRAY(ind_m) elem_m;

sort :
PROC (vector vect_m INOUT,
order PROC (vector elem_m IN,
vector elem_m IN) BOOL);
ASSERT (ALL i IN vector.ind_m:
CONST(FREQ(vector, vector(i))));
ASSERT POST (ALL i IN
vector.ind_m.MIN = PRED(vector.ind_m.MAX):
order(vector(i), vector(SUC(i))) = TRUE);
END sort;

GRANT vect_m, sort;

END general_sort;

Fig.9. Specification of a sort facility

user : MODULE

SEIZE vect_m, sort;

SYNMODE my.ind_m = RANGE(1:10);
int_leg : PROC(a,b INT) (BOOL);
RETURN a <= b;
END int_leg;

DCL my_vect vect_m (my.ind_m, INT) =
(2,5,6,2,3,9,27,17,9,1);
sort (my_vect, int_leg);

END user;

Fig.10. Use of the sort facility
4 REFERENCES


Hol 78 Holager, Per Generic Mode Facilities in Mary = [HS 78c: 117..133].


Mor 74 Morris, James H. Jr. Towards more flexible type systems = (Rob 74 : 377..384).


Par 72a Parnas, D.L. On the criteria To Be Used in Decomposing Systems into Modules CACM 15,12(1972)1053..1058.


Rey 83 Reynolds, John C. Types, Abstraction and Parametric Polymorphism = (Mas 83: 913..923].
