SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **32**:669–701 (DOI: 10.1002/spe.454)

A safe variant of the unsafe integer arithmetic of Java™

Jürgen F. H. Winkler*,†

Friedrich-Schiller University, Institute of Computer Science, D-07740 Jena, Germany

SUMMARY

Computers are finite machines and, therefore, the arithmetic operations in a programming language are different from their mathematical counterparts. These restrictions seem not to have been, in general, fully appreciated in programming languages and in computer science textbooks. One example is the programming language Java, which makes it difficult to warn the user in cases in which arithmetic operations produce incorrect results. In this paper we look at integer arithmetic in Java and develop a safe variant of the arithmetic operations in Java. The design of the safe variant of +, $-_{un}$, $-_{bin}$, *, / and ** for the types `byte`, `short`, `int` and `long` reveals a number of deficiencies of Java in integer arithmetic, floating point arithmetic and program structure. Some of these deficiencies are also present in other contemporary programming languages. The paper therefore ends with some proposals for the design of the arithmetic elements of programming languages. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: safe arithmetic operations; program structure; Java; arithmetics in programming languages

'Now it is obvious that no *finite* machine can include infinity'
Charles Babbage, 1864

1. INTRODUCTION

Computers are finite machines and, therefore, the arithmetic operations in a programming language are different from their mathematical counterparts. This has already been mentioned in the first paper of Hoare on program correctness [1], but he does not usually mention it in later papers. These restrictions seem not to have been, in general, fully appreciated in programming languages and in programming [2,3]. One example is the programming language Java, which makes it difficult to warn the user in cases in which arithmetic operations produce incorrect results. In this paper we look at integer arithmetic in Java and develop safe variants of the arithmetic operations in Java.

*Correspondence to: Jürgen F. H. Winkler, Friedrich-Schiller University, Institute of Computer Science, D-07740 Jena, Germany.
†E-mail: jwinkler@acm.org

Received 1 June 2001
Revised 7 February 2002
Accepted 7 February 2002

In the discussion of arithmetic operations in programming languages we have to distinguish between the operation in the programming language (l-op) and their mathematical counterpart (m-op). We use the following notation for this purpose:

(a) in program text the operation symbols $+$, $-$ etc. denote the l-ops;
(b) outside the program text $+$, $-$ etc. denote the m-ops in $\mathbb{Z}$ and $+_{\text{Jint}}$, $-_{\text{Jint}}$ etc. denote the l-ops in Java for the corresponding predefined integer type of Java. If we do not have a specific programming language in mind we denote the l-ops by $+_{\text{L}}$, $-_{\text{L}}$ etc.

For an integer type in a programming language the set of possible values is limited to a finite interval $[\min \ldots \max] \subset \mathbb{Z}$. This also holds for long numbers in Lisp or Mathematica. In languages like Ada, C, Java and Pascal the interval typically has the form $[-2^{n-1} \ldots 2^{n-1} - 1]$, where a typical value of n is 32. In the rest of this paper we assume $n \geqslant 2$. $n = 1$ works also for $+$, $-_{\text{bin}}$, $*$, $/$, but not for $**$ and $-_{\text{un}}$.

The arithmetic operations which are closed in $\mathbb{Z}$ are usually not closed in $[\min \ldots \max]$, e.g. $\max + 1 \notin [\min \ldots \max]$. There are different possibilities to define the semantics of such operations. Hoare discusses three in [1].

Basically, there are two methods to cope with the problem that the m-ops are not closed in the restricted domain (DR).

(a) Define the l-op in such a way that it is closed in DR, e.g.

$$a +_{\text{L}} b = \begin{cases} a + b \in [\min \ldots \max] & \mapsto a + b \\ a + b > \max & \mapsto \max \\ a + b < \min & \mapsto \min \end{cases}$$

This is analogous to case (2) in [1].
Java uses the following definition for a signed integer:

$$a +_{\text{J}} b = \begin{cases} a + b \in [\min \ldots \max] & \mapsto a + b \\ a + b > \max & \mapsto a + b + 2 * \min \\ a + b < \min & \mapsto a + b - 2 * \min \end{cases} \tag{1}$$

This definition is the same as machine arithmetic when ignoring overflow.
(b) Signal an abnormal behavior, e.g. by raising an exception (e.g. Ada) or aborting the program (e.g. Pascal[‡]).

From a practical point of view method (a) is not useful because in most cases the result of abnormal cases of the l-op is incorrect and makes no sense. If such an incorrect operation is used in the definition of another function this function too may become incorrect. One example is the factorial function. In C or Java we obtain $23! = 8128\_29161\_78948\_25984$, which is quite different from the correct value $258\_52016\_73888\_49766\_40000$ [2].

[‡]This is not completely correct, because such errors need not be detected [4, 3.1, 6.7.2.2].

In most cases method (b) is appropriate. The Java specification [5] is very much aware of the fact that most operations may not be executable in a normal manner but may instead result in erroneous or abnormal behavior, which is called 'abrupt completion' [4, 14.1]. Gosling *et al.* [5] specify in most cases of non-normal behavior that an exception must be thrown. Unfortunately, this is not required for almost all arithmetic errors; only division and remainder with a divisor 0 must result in an exception. All other range violations are masked due to the definition of the l-op in a manner analogous to (1). The language specification requires such a behavior and, therefore, a Java system is not even allowed to throw an exception when a range violation occurs during the evaluation of an arithmetic expression [5, ch. 15]. In this case compatibility to C++ (and C) does not seem to lead to good results. C# uses the same strategy in the unchecked mode, but also provides a checked mode, in which overflow yields an exception [6, 14.5.12].

If the incorrect results produced by the arithmetic operations in Java cannot be tolerated in the program, the programmer must check the operands and the result explicitly. In this paper we present safe implementations of the operations $+$, $-$, $*$, $/$ and $**$, where $**$ is not predefined in Java. $+$ and $-$ come in two forms, namely unary and binary, where $-_{\mathrm{un}}$ is typically called negation. Unary $+$ is the identity and therefore does not pose any problems. Negation is not closed in the integer types of Java and has therefore also to be implemented in a safe way.

This paper is organized as follows. In Section 2 we present the basic properties of integer machine arithmetic as far as they are used later on in the paper. Section 3 describes the integer operations as they are defined in Java. Some properties of floating point machine arithmetic are mentioned in Section 4 and Section 5 mentions the basic properties of floating point arithmetic in Java. Section 6 contains the conditions which the arguments of the various operations must fulfill in order to obtain a correct result. The implementation of the safe operations is derived in Section 7. Some programs are rather complicated because Java does not allow the user to control the rounding of arithmetic operations. From a structural point of view, pure object-orientation and the lack of generics lead to a somewhat poor program structure and to a cumbersome syntax of the calls of the implemented functions. Section 8 lists the deficiencies of Java observed in this paper and Section 9 offers some conclusions, especially concerning the definition of arithmetics in programming languages.

2. INTEGER MACHINE ARITHMETIC

Java defines integer arithmetic via two's-complement arithmetic (TCA) as it is used in contemporary processors. We therefore present briefly those aspects of machine arithmetic which are used in the rest of the paper.

One characteristic of TCA is the use of bit vectors of a fixed number n of bits for the two's-complement representation (TCR) of numbers. We obtain two characteristic sets:

$$\mathrm{ru}_n = 0 \ldots 2^n - 1, \quad \text{the range of unsigned numbers with } n \text{ bits}$$
$$\mathrm{rs}_n = -2^{n-1} \ldots 2^{n-1} - 1, \quad \text{the range of signed numbers with } n \text{ bits.}$$

In TCA the numbers of rs_n are internally represented as elements of ru_n according to the following mappings:

$$\begin{aligned}
\mathrm{tc}_n &\in \mathrm{rs}_n \to \mathrm{ru}_n, \quad \text{the TCR of } x \text{ with } n \text{ bits} \\
\mathrm{tc}_n(x) &= \{0 \leqslant x \leqslant 2^{n-1}-1 \mapsto x \mid -2^{n-1} \leqslant x \leqslant -1 \mapsto x + 2^n\} \\
\mathrm{nb}_n &\in \mathrm{ru}_n \to \mathrm{rs}_n, \quad \text{the number represented by the internal value } x \\
\mathrm{nb}_n(x) &= \{0 \leqslant x \leqslant 2^{n-1}-1 \mapsto x \mid 2^{n-1} \leqslant x \leqslant 2^n - 1 \mapsto x - 2^n\}
\end{aligned}$$

For the definition of a function consisting of several cases we also use, apart from the usual format of (1), a linear format according to the following definition:

$$f(a) = \{\mathrm{c1} \mapsto \mathrm{e1} \mid \mathrm{c2} \mapsto \mathrm{e1}\} :\equiv (\mathrm{c1} \Rightarrow f(a) = \mathrm{e1}) \wedge (\mathrm{c2} \Rightarrow f(a) = \mathrm{e2})$$

This linear format is more convenient in proofs involving such a function. This can be seen in the proof below.

It is easy to see that the following two identities hold:

$$\mathrm{nb}_n(\mathrm{tc}_n(x)) = x, \quad \mathrm{tc}_n(\mathrm{nb}_n(x)) = x$$

When n is understood from the context we sometimes also omit the index n.

If we have two TCR formats of different lengths we can easily transform the TCRs of x. Let $x \in \mathrm{rs}_n$ and $n < m$. The transformation from $\mathrm{tc}_n(x)$ into $\mathrm{tc}_m(x)$ is done by sign extension, which means that the additional bits are all the same as the most significant bit of $\mathrm{tc}_n(x)$. In arithmetic terms this is defined as follows:

$$\mathrm{tc}_m(x) = \mathrm{se}_{n,m}(\mathrm{tc}_n(x)),$$

where

$$\begin{aligned}
\mathrm{se}_{n,m} &\in \mathrm{ru}_n \to \mathrm{ru}_m, \quad \text{the sign extension of an } n\text{-bit TCR into an } m\text{-bit TCR} \\
\mathrm{se}_{n,m}(y) &= \{0 \leqslant y \leqslant 2^{n-1}-1 \mapsto y \mid 2^{n-1} \leqslant y \leqslant 2^n - 1 \mapsto y + \mathrm{sum}(2^j, j, n \ldots m-1)\}
\end{aligned}$$

We now have to show that $\mathrm{se}_{n,m}$ does the job:

$$\begin{aligned}
\mathrm{se}_{n,m}(\mathrm{tc}_n(x)) &= \{0 \leqslant \mathrm{tc}_n(x) \leqslant 2^{n-1}-1 \mapsto \mathrm{tc}_n(x) \mid 2^{n-1} \leqslant \mathrm{tc}_n(x) \leqslant 2^n - 1 \\
&\quad \mapsto \mathrm{tc}_n(x) + \mathrm{sum}(2^j, j, n \ldots m-1)\} \\
&= \{0 \leqslant x \leqslant 2^{n-1}-1 \mapsto x \mid -2^{n-1} \leqslant x \leqslant -1 \\
&\quad \mapsto x + 2^n + \mathrm{sum}(2^j, j, n \ldots m-1)\}, \quad \text{by definition of } \mathrm{tc}_n \\
&= \{0 \leqslant x \leqslant 2^{n-1}-1 \mapsto x \mid -2^{n-1} \leqslant x \leqslant -1 \mapsto x + \mathrm{sum}(2^j, j, 0 \ldots n-1) + 1 \\
&\quad + \mathrm{sum}(2^j, j, n \ldots m-1)\}, \quad \text{as } 2^n = \mathrm{sum}(2^j, j, 0 \ldots n-1) + 1, n > 0 \\
&= \{0 \leqslant x \leqslant 2^{n-1}-1 \mapsto x \mid -2^{n-1} \leqslant x \leqslant -1 \\
&\quad \mapsto x + \mathrm{sum}(2^j, j, 0 \ldots m-1) + 1\} \\
&= \{0 \leqslant x \leqslant 2^{n-1}-1 \mapsto x \mid -2^{n-1} \leqslant x \leqslant -1 \\
&\quad \mapsto x + 2^m\}, \quad \text{as } 2^n = \mathrm{sum}(2^j, j, 0 \ldots n-1) + 1, n > 0 \\
&= \mathrm{tc}_m(x), \quad \text{by } n < m, \text{definition of } \mathrm{tc}_m, -2^{n-1} \leqslant x \leqslant 2^{n-1} - 1
\end{aligned}$$

$\mathrm{se}_{n,m}$ has the additional property that $\mathrm{se}_{n,m}(y) \bmod 2^n = y$ which is based on the fact that $\mathrm{sum}(2^j, j, n \ldots m-1)$ is a multiple of 2^n. If we apply this to $\mathrm{tc}_n(x)$ we see that the transformation in the opposite direction is even simpler: $\mathrm{tc}_m(x) \bmod 2^n = \mathrm{se}_{n,m}(\mathrm{tc}_n(x)) \bmod 2^n = \mathrm{tc}_n(x)$.

In machine arithmetic, addition and other operations yield two results: a result value and flags which indicate, e.g., whether the value is in rs_n [7, 3.6.3]. Since Java requires that those flags are ignored we define the operations only with respect to the result value.

For addition, we obtain

$$+_{sn} \in \mathrm{rs}_n \times \mathrm{rs}_n \to \mathrm{rs}_n, \quad \text{addition in the set } \mathrm{rs}_n \text{ using two's-complement and ignoring overflow}$$

$$a +_{sn} b = \mathrm{nb}_n((\mathrm{tc}_n(a) + \mathrm{tc}_n(b)) \bmod 2^n)$$

$$= \begin{cases} a \geqslant 0 \wedge b \geqslant 0 \wedge 0 \leqslant a+b \leqslant 2^{n-1}-1 & \mapsto a+b, \quad \text{mc (mathematically correct)} \\ a \geqslant 0 \wedge b \geqslant 0 \wedge 2^{n-1} \leqslant a+b \leqslant 2^n-2 & \mapsto a+b-2^n \\ a \geqslant 0 \wedge b < 0 & \mapsto a+b, \quad \text{mc} \\ a < 0 \wedge b \geqslant 0 & \mapsto a+b, \quad \text{mc} \\ a < 0 \wedge b < 0 \wedge -2^{n-1} \leqslant a+b < 0 & \mapsto a+b, \quad \text{mc} \\ a < 0 \wedge b < 0 \wedge -2^n \leqslant a+b < -2^{n-1} & \mapsto a+b+2^n \end{cases}$$

We observe that $+_{sn}$ does not always give the mathematically correct result:

$$1\_000\_000\_000 +_{s32} 1\_000\_000\_000 = 2\_000\_000\_000$$

$$2\_000\_000\_000 +_{s32} 2\_000\_000\_000 = -294\_967\_296 = 4\_000\_000\_000 - 2^{32}$$

For subtraction we obtain analogously

$$-_{sn} \in \mathrm{rs}_n \times \mathrm{rs}_n \to \mathrm{rs}_n, \quad \text{subtraction in the set } \mathrm{rs}_n \text{ using two's-complement and ignoring overflow}$$

$$a -_{sn} b = \mathrm{nb}_n((\mathrm{tc}_n(a) + \mathrm{tc}_n(-b)) \bmod 2^n)$$

$$= \begin{cases} a \geqslant 0 \wedge b \leqslant 0 \wedge 0 \leqslant a-b \leqslant 2^{n-1}-1 & \mapsto a-b, \quad \text{mc} \\ a \geqslant 0 \wedge b < 0 \wedge 2^{n-1} \leqslant a-b \leqslant 2^n-1 & \mapsto a-b-2^n \\ a \geqslant 0 \wedge b > 0 & \mapsto a-b, \quad \text{mc} \\ a < 0 \wedge b \leqslant 0 & \mapsto a-b, \quad \text{mc} \\ a < 0 \wedge b > 0 \wedge -2^{n-1} \leqslant a-b < -1 & \mapsto a-b, \quad \text{mc} \\ a < 0 \wedge b > 0 \wedge -2^n+1 \leqslant a-b < -2^{n-1} & \mapsto a-b+2^n \end{cases}$$

As for addition, we do not always obtain the mathematically correct result.

Negation can be derived from subtraction because $-a = 0 - a$. We therefore obtain

$$-_{sn} \in \mathrm{rs}_n \to \mathrm{rs}_n, \quad \text{negation in the set } \mathrm{rs}_n \text{ using two's-complement and ignoring overflow}$$

$$-_{sn} a = \mathrm{nb}_n((\mathrm{tc}_n(0) + \mathrm{tc}_n(-a)) \bmod 2^n)$$

$$= \begin{cases} -2^{n-1}+1 \leqslant a \leqslant 2^{n-1}-1 & \mapsto -a, \quad \text{mc} \\ a = -2^{n-1} & \mapsto a \end{cases}$$

We observe that $-_{sn}$ works correctly for all but one value, which is the 'additional' negative value -2^{n-1}.

For multiplication we obtain a more complicated definition:

$*_{sn} \in \mathrm{rs}_n \times \mathrm{rs}_n \to \mathrm{rs}_n$, multiplication in the set rs_n using two's-complement and ignoring overflow

$$a *_{sn} b = \mathrm{nb}_n((\mathrm{tc}_n(a) * \mathrm{tc}_n(b)) \bmod 2^n)$$

$$= \begin{cases}
a \geqslant 0 \wedge b \geqslant 0 \wedge 0 \leqslant a * b \leqslant 2^{n-1} - 1 \\
\quad \mapsto a * b, \quad \mathrm{mc} \\
a > 0 \wedge b > 0 \wedge 2^{n-1} - 1 < a * b \leqslant 2^{2n-2} - 2^n + 1 \wedge 0 \leqslant (a * b) \bmod 2^n \leqslant 2^{n-1} - 1 \\
\quad \mapsto (a * b) \bmod 2^n \\
a > 0 \wedge b > 0 \wedge 2^{n-1} - 1 < a * b \leqslant 2^{2n-2} - 2^n + 1 \wedge 2^{n-1} \leqslant (a * b) \bmod 2^n \leqslant 2^n - 1 \\
\quad \mapsto (a * b) \bmod 2^n - 2^n \\
a \geqslant 0 \wedge b \leqslant 0 \wedge -2^{n-1} \leqslant a * b \leqslant 0 \\
\quad \mapsto a * b, \quad \mathrm{mc} \\
a > 0 \wedge b < 0 \wedge -2^{2n-2} + 2^{n-1} \leqslant a * b < -2^{n-1} \wedge 0 \leqslant (a * b) \bmod 2^n \leqslant 2^{n-1} - 1 \\
\quad \mapsto (a * b) \bmod 2^n \\
a > 0 \wedge b < 0 \wedge -2^{2n-2} + 2^{n-1} \leqslant a * b < -2^{n-1} \wedge 2^{n-1} \leqslant (a * b) \bmod 2^n \leqslant 2^n - 1 \\
\quad \mapsto (a * b) \bmod 2^n - 2^n \\
a \leqslant 0 \wedge b \geqslant 0 \wedge -2^{n-1} \leqslant a * b \leqslant 0 \\
\quad \mapsto a * b, \quad \mathrm{mc} \\
a < 0 \wedge b > 0 \wedge -2^{2n-2} + 2^{n-1} \leqslant a * b < -2^{n-1} \wedge 0 \leqslant (a * b) \bmod 2^n \leqslant 2^{n-1} - 1 \\
\quad \mapsto (a * b) \bmod 2^n \\
a < 0 \wedge b > 0 \wedge -2^{2n-2} + 2^{n-1} \leqslant a * b < -2^{n-1} \wedge 2^{n-1} \leqslant (a * b) \bmod 2^n \leqslant 2^n - 1 \\
\quad \mapsto (a * b) \bmod 2^n - 2^n \\
a \leqslant 0 \wedge b \leqslant 0 \wedge 0 \leqslant a * b \leqslant 2^{n-1} - 1 \\
\quad \mapsto a * b, \quad \mathrm{mc} \\
a < 0 \wedge b < 0 \wedge 2^{n-1} - 1 < a * b \leqslant 2^{2n-2} \wedge 0 \leqslant (a * b) \bmod 2^n \leqslant 2^{n-1} - 1 \\
\quad \mapsto (a * b) \bmod 2^n \\
a < 0 \wedge b < 0 \wedge 2^{n-1} - 1 < a * b \leqslant 2^{2n-2} \wedge 2^{n-1} \leqslant (a * b) \bmod 2^n \leqslant 2^n - 1 \\
\quad \mapsto (a * b) \bmod 2^n - 2^n
\end{cases}$$

Since multiplication grows faster than addition we obtain an incorrect result in more cases. Some examples are

$$4 *_{s32} 536\_870\_911 = 2\_147\_483\_644$$
$$4 *_{s32} 536\_870\_912 = -2\_147\_483\_648 = (2\_147\_483\_648 \bmod 2^{32}) - 2^{32}$$
$$4 *_{s32} 1\_610\_612\_735 = 2\_147\_483\_644 = 6\_442\_450\_940 \bmod 2^{32}$$
$$4 *_{s32} 1\_610\_612\_736 = -2\_147\_483\_648 = (6\_442\_450\_944 \bmod 2^{32}) - 2^{32}$$

Division is different from the other three operations in that it often yields results outside $\mathbb{Z}$, e.g. $5/2$. In programming languages integer division is usually defined in such a way that it yields integer

numbers as a result, i.e. if necessary it includes some form of rounding. We use rounding towards zero as is done in Java [8, 15.17.2] and obtain the following definition:

$$/_{sn} \in \mathrm{rs}_n \times \mathrm{rs}_n \to \mathrm{rs}_n \cup \{\text{exception}\}, \quad \text{division in the set } \mathrm{rs}_n \text{ ignoring overflow}$$

$$a /_{sn} b = \begin{cases} b = 0 & \mapsto \text{exception}, \quad \text{mc} \\ a \geqslant 0 \wedge b > 0 & \mapsto \lfloor a/b \rfloor, \quad \text{mc} \\ a \geqslant 0 \wedge b < 0 & \mapsto \lceil a/b \rceil, \quad \text{mc} \\ a < 0 \wedge b > 0 & \mapsto \lceil a/b \rceil, \quad \text{mc} \\ a < 0 \wedge b < -1 & \mapsto \lfloor a/b \rfloor, \quad \text{mc} \\ -2^{n-1} + 1 \leqslant a < 0 \wedge b = -1 & \mapsto -a, \quad \text{mc} \\ a = -2^{n-1} \wedge b = -1 & \mapsto a \end{cases}$$

We almost always obtain the correct result, only the last case ($a = -2^{n-1} \wedge b = -1$) yields a wrong result.

For the notation of exponentiation we use additionally to the traditional notation a^b a notation with an explicit operator $a ** b$. $**$ is the mathematical operation and $**_{sn}$, $**_{\mathrm{Jint}}$ etc. are the operations in the machine, respectively in Java. Since exponentiation is not one of the five basic operations it is not so well defined as these. In particular, the case $0 ** 0$ may be defined differently: sometimes it is illegal or undefined (e.g. Algol 60), or sometimes $0 ** 0 = 1$ (e.g. Ada 83). We use the following definition:

$$** \in \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \cup \{\text{undefined}\}$$

$$a ** b = \begin{cases} b = 0 & \mapsto 1 \\ b > 0 & \mapsto a * a * \cdots * a, \quad b \text{ times} \\ a \neq 0 \wedge b < 0 & \mapsto 1/(a * a * \cdots * a), \quad -b \text{ times}, / \in \mathbb{Z} \times \mathbb{Z} \hookrightarrow \mathbb{Z} \\ a = 0 \wedge b < 0 & \mapsto \text{undefined} \end{cases} \tag{2}$$

We do not give a language or machine oriented definition for $**$ because Java does not contain $**$ as a predefined integer operation. Java contains the operation 'double pow (double a, double b)' [9]. $**$ is also typically not provided by contemporary microprocessors (e.g. the Intel Pentium [10]).

3. INTEGER ARITHMETIC IN JAVA

In Java there are five integer types with the following ranges [4, 4.2.1]:

`byte:`	$-128 \ldots 127$	8-bit TCR
`short:`	$-32768 \ldots 32767$	16-bit TCR
`int:`	$-2147483648 \ldots 2147483647$	32-bit TCR
`long:`	$-9223372036854775808 \ldots 9223372036854775807$	64-bit TCR
`char:`	'\u0000' ... '\uffff' $= 0 \ldots 65535$	16-bit unsigned representation

`char` is primarily a character type, but according to the traditional low-level viewpoint of C in the area of data types the arithmetic operations are also defined for `char`. In the rest of the paper we limit the discussion therefore to the types `byte`, `short`, `int` and `long`.

In Java, integer arithmetic operations are actually only performed on values of type `int` or type `long`:

> 'Binary numeric promotion is performed on the operands (§5.6.2). The type of a multiplicative expression is the promoted type of its operands. If this promoted type is int or long, then integer arithmetic is performed; . . . If an integer multiplication overflows, then the result is the low-order bits of the mathematical product as represented in some sufficiently large two's-complement format. As a result, if overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two operand values.' [8, 15.17].

Addition and subtraction are defined analogously. The operation / yields an integer result by rounding towards zero. If the value of the divisor of / is 0, then an ArithmeticException is thrown. Apart from this, / is defined analogously to $*$. Promotion of operands of type `byte` and `short` means conversion to type `int` by sign extension of the TCR and analogously for the promotion from `int` to `long`.

Since the result type is the *promoted type*, we run into further problems when using `byte` or `short` operands, as we see in the following example. The compilation of

```
short vs = 17;
                          // line 5
vs = vs-vs;    // should lead to a compile-time error (5.2)
```

with Java™ 2 SDK, Standard Edition, Version V1.3.0 leads to a compile-time error:

```
D:\JFHW\Projekte\Fin_Add\java-ex\assign-compat>javac AssignCompat04.java
AssignCompat04.java:6: possible loss of precision
found   : int
required: short
      vs = vs-vs;     // should lead to a compile-time error (5.2)
           ^
1 error
```

This error message is based on the following language rules:

> 'A compile-time error occurs if the type of the right-hand operand cannot be converted to the type of the variable by assignment conversion (§5.2).' [8, 15.26]

> '5.2 Assignment Conversion
>
> Assignment conversion occurs when the value of an expression is assigned (§15.26) to a variable: the type of the expression must be converted to the type of the variable. Assignment contexts allow the use of an identity conversion (§5.1.1), a widening primitive conversion (§5.1.2), or a widening reference conversion (§5.1.4). In addition, a narrowing primitive conversion may be used if all of the following conditions are satisfied:
>
> - The expression is a constant expression of type byte, short, char or int.
> - The type of the variable is byte, short, or char.

- The value of the expression (which is known at compile time, because it is a constant expression) is representable in the type of the variable.

> If the type of the expression cannot be converted to the type of the variable by a conversion permitted in an assignment context, then a compile-time error occurs. . . .
>
> If the type of an expression can be converted to the type of a variable by assignment conversion, we say the expression (or its value) is assignable to the variable or, equivalently, that the type of the expression is assignment compatible with the type of the variable.' [8, 5.2]

If the variable has type `int` the program is legal and is executed without any warning, even in those cases in which an incorrect result is computed:

```
int i = 0;
i = 2000000000+2000000000;
System.out.print("2000000000+2000000000 = " + i);
```

Execution:

```
D:\JFHW\Projekte\Fin_Add\java-ex\addition>java Addition02
2000000000+20000000000 =   -294967296
```

If the type of the variable is `short` the program can easily be legalized by using a cast. This also works in those cases in which an incorrect result is computed:

```
short vs = 20000;
vs = (short) (vs+vs);       // should work (5.5)
System.out.print("20000+20000 =   " + vs);
```

Execution:

```
D:\JFHW\Projekte\Fin_Add\java-ex\assign-compat>java AssignCompat06
20000+20000 =   -25536
```

Whereas the definition of assignment compatibility and especially the conditions for the application of a narrowing primitive conversion suggests that Java is very conscious of the finite ranges of integer types, this is a fallacious idea. For the types `int` and `long` errors apart from division by zero are completely ignored and the same holds for the other integer types when a cast is used. The situation is especially confusing because the application of conversions (promotions) in the cases of `short` and `byte` is asymmetric: the operands are *automatically* converted into `int`, but the result is `int` and must, if need arises, be *explicitly* converted (cast) back to the original type.

If we sum up this discussion, we observe that integer arithmetic in Java is defined to be unsafe and rather confusing.

4. FLOATING POINT MACHINE ARITHMETIC

The topic of this paper is integer arithmetic. However, we will see in later sections that we could use some floating point operations as, e.g., the function log( ). We therefore mention briefly some properties of floating point machine arithmetic. Today most processors implement floating point arithmetic according to the IEEE 754 standard. This standard defines four different formats for floating point numbers: single (32 bit), single extended, double (64 bit) and double extended. The numbers are defined as triples consisting of sign, mantissa and exponent. Apart from proper numbers, IEEE 754 defines some triples, which represent entities like positive infinity or negative infinity. Despite the existence of these special triples, events such as overflow or underflow result in an exception[§]. These exceptions may be enabled or disabled by the user.

In floating point arithmetic rounding occurs much more often than in integer arithmetic. IEEE 754 gives the programmer a very detailed control over the rounding mode of an operation. The following rounding modes are defined:

RTN	round to nearest
RTPI	round toward positive infinity
RTNI	round toward negative infinity
RTZ	round toward zero

RTN is the default rounding mode, but depending on the computation it can be necessary to use one of the other rounding modes.

5. FLOATING POINT ARITHMETIC IN Java

The floating point arithmetic of Java is based on IEEE 754, but it does not support this standard fully and therefore has a number of deficiencies [11]. Java provides two floating point types, `float` and `double`, which correspond to the IEEE 754 32-bit and 64-bit formats. The arithmetic operations for floating point types in Java do not yield any error indication in case the result value is not a proper number. All operations in Java use the rounding mode RTN. In this paper this limitation of Java will lead to serious problems (see Section 7.1).

6. CONDITIONS FOR SAFE COMPUTATION

The safe integer operations we want to develop shall yield either the mathematically correct result or throw an exception. For addition and the type rs_n this leads to the following specification:

$$+_{\mathrm{sf}n} \in \mathrm{rs}_n \times \mathrm{rs}_n \to \mathrm{rs}_n \cup \{\text{exception}\}$$
$$a +_{\mathrm{sf}n} b = \{a + b \in \mathrm{rs}_n \mapsto a + b \mid a + b \notin \mathrm{rs}_n \mapsto \text{exception}\}$$

[§]What we call 'exception' is called 'trap' in IEEE 754. We use 'exception' because this is the typical term in the field of programming languages.

The 'f' in $+_{sfn}$ stands for 'safe'. For the other operations and types we obtain analogous definitions. For the Java type `int` we obtain, e.g.,

$$+_{Jfint} \in \texttt{int} \times \texttt{int} \to \texttt{int} \cup \{\text{exception}\}$$
$$a +_{Jfint} b = \{a + b \in \texttt{int} \mapsto a + b \mid a + b \notin \texttt{int} \mapsto \text{exception}\}$$

Since we do not have $+$ available in the program, we cannot use the condition $a + b \in \texttt{int}$ directly. It is also easy to see that it is fallacious to try to use $a +_{Jint} b \in \texttt{int}$ because this is always true. The latter holds because $+_{Jint} = +_{s32}$. It is also not possible to use the conditions in the definitions of Section 2 directly, because they are based on the mathematical operations. A general rule for the construction of the conditions is to use Java operations only in such cases in which they yield the correct result. A consequence of this is that we have to distinguish more cases than in a purely mathematical formulation of the conditions.

We derive the safety conditions for the general case of TCA with n bits, where $n \geqslant 2$, $\text{rs}_n = [\text{min} \ldots \text{max}]$, $\text{min} = -\text{max} - 1$ and $\text{max} = 2^{n-1} - 1$. The safety condition can be either a correctness condition or an incorrectness condition. The general correctness condition for operation $\circ_{sfn}$ is

$$\text{min} \leqslant a, b, a \circ_{sfn} b \leqslant \text{max}$$

6.1. Addition

In the definition of $+_{sn}$ in Section 2, we observe that there are four cases in which the result is correct and two cases in which it is incorrect. These two cases are:

$$a > 0 \wedge b > 0 \wedge \text{max} < a + b \leqslant 2 * \text{max} \mapsto a + b - (2 * \text{max} + 2) \qquad \in \text{min} \ldots -2$$
$$a < 0 \wedge b < 0 \wedge 2 * \text{min} \leqslant a + b < \text{min} \mapsto a + b - 2 * \text{min} \qquad \in 0 \ldots \text{max}$$

As is well known, overflow only occurs when both operands have the same sign. The result of $+_{sn}$ then has the opposite sign. This means that the incorrectness condition can also be written as

$$(a > 0 \wedge b > 0 \wedge a +_{sn} b < 0) \vee (a < 0 \wedge b < 0 \wedge a +_{sn} b \geqslant 0)$$

In this case we can use $+_{sn}$ in the safety condition.

6.2. Subtraction

The two incorrect cases of $-_{sn}$ in Section 2 are

$$a \geqslant 0 \wedge b < 0 \wedge \text{max} < a - b \leqslant \text{max} - \text{min} \mapsto a - b + 2 * \text{min} \qquad \in \text{min} \ldots -1$$
$$a < 0 \wedge b > 0 \wedge \text{min} - \text{max} \leqslant a - b < \text{min} \mapsto a - b - 2 * \text{min} \qquad \in 1 \ldots \text{max}$$

The incorrectness condition is

$$(a \geqslant 0 \wedge b < 0 \wedge a -_{sn} b < 0) \vee (a < 0 \wedge b > 0 \wedge a -_{sn} b > 0)$$

6.3. Negation

The incorrectness condition is

$$a = \text{min}$$

6.4. Multiplication

The incorrectness condition is

$$(a > 0 \wedge (b < \lceil \min/a \rceil \vee b > \lfloor \max/a \rfloor)) \\ \vee (a = -1 \wedge b = \min) \vee (a < -1 \wedge (b < \lceil \max/a \rceil \vee b > \lfloor \min/a \rfloor))$$

Since division is only used in cases in which $/_{sn}$ is mc we can use $/_{sn}$ which also does the necessary rounding:

$$(a > 0 \wedge (b < \min/_{sn} a \vee b > \max/_{sn} a)) \\ \vee (a = -1 \wedge b = \min) \vee (a < -1 \wedge (b < \max/_{sn} a \vee b > \min/_{sn} a))$$

6.5. Division

The incorrectness condition is

$$b = 0 \vee (a = \min \wedge b = -1)$$

6.6. Exponentiation

The first and the third alternative of (2) never result in a range violation. The fourth alternative is already mathematically undefined.

Therefore, we have only to analyze $b > 0 \wedge \min \leqslant a ** b \leqslant \max$.

If $-1 \leqslant a \leqslant 1$ then $1 \leqslant b \leqslant \max$.
If $a \geqslant 2$ then $1 \leqslant b \leqslant \lfloor \lg \max / \lg a \rfloor (= \lfloor \mathrm{ld} \max / \mathrm{ld} a \rfloor)$.
If $a \leqslant -2$ then the range of admissible values of b depends on the evenness of n.

If n is even then $n-1$ is odd and therefore $-2^{n-1} = (-2)^{n-1}$. For any $a \leqslant -2$ and $b > 0$ with $|a ** b| = |\min|$ b is odd, i.e. we never obtain $a ** b = -\min$ and thus the constraint for b is $1 \leqslant b \leqslant \lfloor \lg |\min| / \lg |a| \rfloor$.
If n is odd then $n-1$ is even.

If $|a| = 2^m$ then it depends on the fact whether m is a factor of $(n-1)$. In this case $|a|^{(n-1)/m} = 2^{n-1}$.

If $(n-1)/m$ is odd $a ** ((n-1)/m) = -2^{n-1}$. The constraint for b is therefore $1 \leqslant b \leqslant \lfloor \lg |\min| / \lg |a| \rfloor$.
If $(n-1)/m$ is even $a ** ((n-1)/m) = 2^{n-1} = \max + 1$. The constraint for b is therefore $1 \leqslant b \leqslant \lfloor \lg \max / \lg |a| \rfloor$.

If $|a|$ is not a power of 2 then $|a|^{\lfloor \lg |\min| / \lg |a| \rfloor} \leqslant |\min| - 1 = \max$ and therefore, the constraint on b is $1 \leqslant b \leqslant \lfloor \lg \max / \lg a \rfloor$.

In the case of Java n is always even.

For Java the incorrectness condition is therefore

$$(a = 0 \wedge b < 0) \vee (a \geqslant 2 \wedge b > \lfloor \lg \max / \lg a \rfloor) \vee (a \leqslant -2 \wedge b > \lfloor \lg |\min| / \lg |a| \rfloor)$$

7. IMPLEMENTATION

The implementation of the safe arithmetic functions has two aspects: (1) implementation of the algorithm and (2) design of the program structure.

7.1. Implementation of the algorithm

Since there are four different integer types and six different operations we have to implement 24 operations of the following kinds:

$$\mathrm{op} \in \mathrm{rs}_n \times \mathrm{rs}_n \to \mathrm{rs}_n \cup \{\mathrm{exception}\}, \quad \text{where } \mathrm{rs}_n \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \texttt{long}\} \wedge \mathrm{op} \in \{+, -, *, /, **\}$$

and

$$\mathrm{op} \in \mathrm{rs}_n \to \mathrm{rs}_n \cup \{\mathrm{exception}\}, \quad \text{where } \mathrm{rs}_n \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \texttt{long}\} \wedge \mathrm{op} \in \{-\}$$

Since widening conversions between integer types do not lead to any loss of information [8, 5.1.2] we have two alternatives for the implementation of these operations:

(1) perform first the computation, possibly in some larger type, and then check whether the result is in the given type (post-check strategy); an example for this is

```
public static short Add(short L, short R) {
   int result = (int)(L+R);      // L and R are automatically widened to int
   if (result<MinS || result>MaxS)
      throw new IllegalArithArgsException(ShortTy, AddOp, L, R);
   else return (short)result;
}
```

(2) check first whether the operand values are legal and compute then the result in the result type (pre-check strategy); an example for this is

```
public static long Mul(long L, long R) {
    if (( L>0 && (R<MinL/L || R>MaxL/L) ) ||
        ( L==-1 && R==MinL) || (L<-1 && (R<MaxL/L || R>MinL/L) ))
       throw new IllegalArithArgsException(LongTy, MulOp, L, R);
    else return L*R;
}
```

The post-checked solution is usually the simpler one. Since Java also supports arithmetic with arbitrary long integer numbers (class `BigInteger`) we could, in principle, implement most of the 24 operations in the post-checked fashion[¶]. From a dynamic point of view the use of `BigInteger` means that the operations will become rather slow. Therefore we do not use `BigInteger` in this paper but develop implementations of the 24 operations using the basic types only.

[¶]Even `BigInteger` is not big enough. If we assume the representation used in `BigInteger` the value of $-2^{63} ** (2^{63} - 1)$ needs about 68 billion GB of storage.

If an exception is thrown we add as additional information an indication of the type, an indication of the operation and the two operands. For the indication of type and operation we define two enumeration types using a scheme similar to that proposed by Cairns [12], because Java does not support enumeration types directly.

For $rs_n \in \{\texttt{byte}, \texttt{short}, \texttt{int}\} \wedge op \in \{+, -, *\}$ the post-checked solution can be used. The most critical of these is $*$. However, we observe that $(-2^{n-1}) * (-2^{n-1}) = 2^{2n-2} < 2^{2n-1} - 1$ (for $n \geqslant 2$). For $rs_n = \text{long} \wedge op \in \{+, -, *\}$ we have to use the pre-checked solution.

For negation we always use the pre-checked solution because the check of this solution involves only one comparison operation.

The operation / is easy, because we only have to check for $a = \min \wedge b = -1$. The check for $b = 0$ is done by the predefined operation.

For $**$ we always have to use the pre-checked solution because two operands of type `byte` may lead to a result not in `long`: $(-2^7) ** (2^7 - 1) = -2 ** (7 * 127) = -2^{889} < -2^{63}$. For the computation of the result we can use `StrictMath.pow` which is of type `double` $\times$ `double` $\rightarrow$ `double`. All values of `byte`, `short` and `int` can be converted to `double` without loss of information [8, 5.1.2]. If the arguments of `StrictMath.pow` are integer values and the result R is in the range of `int` then R will be an integer value represented in the format `double` [9, pow]. All integer numbers of `float` and `double`, whose values are in `byte`, `short`, `int` or `long`, can be converted without loss of information from floating point form into TCR [8, 5.1.3]. Together these facts imply that `StrictMath.pow` can be used to compute $**$ for `byte`, `short` and `int`, if we first check the values of the arguments. For `long` we cannot use `StrictMath.pow` because not all values of `long` can be represented exactly in `double` [8, 5.1.2]. One example is `Long.MAX_VALUE` which has the value $2^{63} - 1$, whereas `(double)Long.MAX`$_{\text{V}}$`ALUE` $= 2^{63}$. To be more exact, beginning with 2^{53} the difference between successive numbers of `double` is greater than 1. We will therefore use repeated multiplication for this case. This problem could be avoided if Java also offered the double extended format of IEEE 754, which contains at least 64 bits for the mantissa. Kahan and Darcy [11] also propose support of the double extended format.

The algorithms for the 24 operations can now be represented by the following eight characteristic patterns:

(1) post-checked: $+$, $-_{\text{bin}}$, $*$ for `byte`, `short`, `int` (nine cases)

```
int Mul(int L, int R) {
   long result = (long)L * (long)R;
   if (result<MinI || result>MaxI)
      throw new IllegalArithArgsException(IntTy, AddOp, L, R);
   else return (int)result;
}
```

(2) post-checked: $+$ for `long`

```
long Add(long L, long R) {
   long result = L+R;
   if (L>0 && R>0 && result<0 || L<0 && R<0 && result>=0)
      throw new IllegalArithArgsException(LongTy, AddOp, L, R);
   else return result;
}
```

(3) post-checked: − for `long`

```
long Sub(long L, long R) {
   long result = L-R;
   if (L>=0 && R<0 && result<0 || L<0 && R>0 && result>0)
      throw new IllegalArithArgsException(LongTy, SubOp, L, R);
   else return result;
}
```

(4) pre-checked negation for `byte`, `short`, `int`, `long` (four cases)

```
short Neg(short L) {
  if (L==MinS)
     throw new IllegalArithArgsException(ShortTy, NegOp, L);
  else return (short)(-L);
}
```

(5) pre-checked: ∗ for `long`

```
final long Mul(long L, long R) {
    if ( ( L>0 && (R<MinL/L || R>MaxL/L) ) ||
         ( L==-1 && R==MinL) || (L<-1 && (R<MaxL/L || R>MinL/L) ) )
       throw new IllegalArithArgsException(LongTy, MulOp, L, R);
    else return L*R;
}
```

(6) pre-checked: / for `byte`, `short`, `int`, `long` (four cases)

```
short Div(short L, short R) {
  if (L==MinS && R==-1)
     throw new IllegalArithArgsException(ShortTy, DivOp, L, R);
  else return (short)(L/R);
}
```

(7) pre-checked: ∗∗ for `byte`, `short`, `int` (three cases)

```
int Exp(int L, int R) {
  if (L==0 && R<0) ||
     (L>=2 && R > (int)StrictMath.floor(LogMaxI/StrictMath.log((double)L) )) ||
     (L<=-2 && R > (int)StrictMath.floor(LogAbsMinI/StrictMath.log(-(double)L) ))
     throw new IllegalArithArgsException(IntTy, ExpOp, L, R);
  else return (int)StrictMath.pow((double)L,(double)R);
}
```

(8) pre-checked ∗∗ for `long`

```
long Exp(long L, long R) {
  if (L==0 && R<0) ||
     (L>=2 && R > (long)StrictMath.floor(LogMaxL/StrictMath.log((double)L) )) ||
     (L<=-2 && R > (long)StrictMath.floor(LogAbsMinL/StrictMath.log(-(double)L) ))
     throw new IllegalArithArgsException(LongTy, ExpOp, L, R);
  else { if (R==0) return 1;
         if (R==1) return L;
         if (L==0 && R>0) return 0;
         if (L==1) return 1;
         if (L==-1) return ((R%2 == 0)? 1: -1)
```

```
            if (R < 0) return (long)1/Exp(L,-R);
            { long res = 1;
              for (byte i=1; i<=R; i++) res = res*L;
              return res; }
           } // end if
    } // end exp
```

In the eight patterns the expressions for the conditions are mostly transliterations of the corresponding mathematical expressions given in Section 4. We now have to check whether the Java expressions yield a correct result.

Patterns (1)–(4), (6). All operations are mathematically exact.

Pattern (5). Integer division involves rounding, which can lead to errors in an expression like $R > \text{Max}\,L/L$ if $\text{Max}\,L/_{\text{J}}L$ is too large. Fortunately, integer division in Java does RTZ [8, 15.17.2] which means $\text{Max}\,L/_{\text{J}}L \leqslant \text{Max}\,L/L$. An analogous reasoning holds for $R < \text{Min}\,L/L$.

Pattern (7), (8). The expression (long)StrictMath.floor(LogMax L/StrictMath.log((double)L)) involves several operations which may lead to rounding errors. In fact these rounding errors lead in several cases to an incorrect result. One example is exp((long)2,(long)63) which gives the result -9223372036854775808 which is obviously wrong. The reason is that (long)StrictMath.floor(LogMax L/StrictMath.log((double)2)) $=$ 63, whereas the mathematically correct value is 62.

The expression

$$R > \text{(long)StrictMath.floor(LogMax}\,L/\text{StrictMath.log((double)2)} \equiv R > \text{Limit}_{\text{J}} \tag{3}$$

involves several kinds of rounding errors. We will therefore look at the computation of this expression in more detail. We will see that the floating point arithmetic of Java has serious deficiencies.

The expression (3) is the condition for illegal values of R. In order to really catch all illegal values of R

$$\text{Limit}_{\text{J}} \leqslant \text{Limit}_{\text{M}} \tag{4}$$

must hold, where Limit_{M} means the mathematically correct value of Limit. (4) may be violated if $\log(\text{Max}\,L)_{\text{J}}$ is too large or if $\log(2)_{\text{J}}$ is too small or if $\log(\text{Max}\,L)_{\text{J}}/_{\text{J}}\log(2)_{\text{J}}$ is too large. We call an intermediate value v of the computation of Limit_{J} *unsafe* if v threatens the validity of (4) and we call v *incorrect* if it implies that (4) does not hold.

Let us now look at the computation of Limit_{J} in detail. There are seven steps in this computation.

(1) Since Java only provides a log function for floating point arguments we have to compute (double)Max L, where $\text{Max}\,L = \text{java.lang.Long.MAX\_VALUE} = 2^{63} - 1$. We obtain $\text{DMax}\,L = \text{(double)Max}\,L = 2^{63} > \text{Max}\,L$ because Max L is not an element of double and because the conversion long $\rightarrow$ double uses RTN [8, 5.1.2]. In this case RTN is unsafe because it is essentially RTPI, whereas RTNI is the appropriate rounding mode. However, despite the fact that $\text{DMax}\,L > \text{Max}\,L$ this is not the reason for the incorrect value of Limit_{J}. If we use RTNI we obtain $\text{DMax}\,L_{\text{RTNI}} = \text{pred}(\text{DMax}\,L) = \text{DMax}\,L - 1024$. If we compute the logarithm using StrictMath.log( ) we obtain

$$\text{StrictMath.log(DMax}\,L) = 43.66827237527655\_114490698906593024730682373046875$$

and

$$\text{StrictMath.log}(\text{DMax}\, L_{\text{RTNI}}) = 43.66827237527655\_1144906989065930247306823730468 75$$

i.e. the log( ) function, which computes an approximation of the natural logarithm, yields the same value in both cases. Whether this value is safe is the topic of step (2).

(2) If we compute ln(Max L) using Mathematica 2.2.3 we obtain

$$\text{N[Log[Max}\, L\text{], 67]} = 43.66827237527655\_4493177203434616573345349885712854975400919993965 89$$

which is greater than StrictMath.log(DMax L), despite the fact that DMax L $>$ Max L. This means that in this case RTN is RTNI and that StrictMath.log(DMax L) is *safe*.

(3) As in step (1) we first have to convert the integer argument value into `double`. We obtain (double)2 = 2.0 which is mathematically exact and therefore *safe*.

(4) For the natural logarithm of 2.0 we obtain

$$\text{StrictMath.log((double)2)} = 0.693147180559945\_2862267639829951804131269454956054 6875$$

$$\text{N[Log[2], 67]} = 0.693147180559945\_3094172321214581765680755001343602552541206800094934$$

which means that the value computed in Java is smaller than the value computed by Mathematica to 67 places and therefore the value computed in Java is *unsafe* because it threatens the condition (4). Java uses RTN, which in this case is RTNI; but RTPI is the appropriate rounding mode. Using RTPI the result would be

$$\text{succ(StrictMath.log((double)2))} = 0.693147180559945\_3972490664455108344554901123046875$$

which is greater than N[Log[2],67].

(5) Up until now we have had a somewhat indecisive situation in that the dividend of the fraction is safe and the divisor is unsafe. Additionally division itself may round in the wrong direction. In Java we obtain

$$\text{Fract}_{\text{J}} = \text{StrictMath.log}(\text{DMax}\, L)/_{\text{J}}\, \text{StrictMath.log}((\text{double})2) = 63.0$$

which is too large and therefore *unsafe*. In this step we observe already that the result is not only unsafe but also *incorrect*.

The dominant reason is that $/_{\text{J}}$ uses RTN, which is RTPI in this case. If we use the division operation of Mathematica and compute this fraction we obtain

$$\text{StrictMath.log}(\text{DMax}\, L)/_{\text{M}}\, \text{StrictMath.log}((\text{double})2) = 62.99999999999999\_7277087471757319875812111439366 02$$

which is safe. For this division RTNI is the appropriate rounding mode. With this we obtain

$$\text{StrictMath.log}(\text{DMax}\, L)/_{\text{JRTNI}}\, \text{StrictMath.log}((\text{double})2) = \text{pred}(\text{StrictMath.log}(\text{DMax}\, L)/_{\text{J}}\, \text{StrictMath.log}((\text{double})2)) = 62.99999999999999\_2894572642398998141288757324218 75$$

which is even safer.

(6) $\text{StrictMath.floor}(\text{Fract}_\text{J}) = \text{StrictMath.floor}(63.0) = 63.0$. This result is also *incorrect* because the argument is a whole number and is itself incorrect. The computation itself is exact in this step. This step is always safe because floor( ) is a rounding operation with the properties of RTNI.

(7) $(\text{long})\text{StrictMath.floor}(\text{Div}_\text{J}) = (\text{long})63.0 = 63$. Again the result is *incorrect* because the argument is incorrect and the conversion is exact in this case.

To obtain the correct result we have to compute

$$(\text{long})\text{StrictMath.floor}(\text{StrictMath.log}_{\text{RTNI}}((\text{double}_{\text{RTNI}})\text{Max}\, L)/_{\text{JRTNI}}\, \text{StrictMath.log}_{\text{RTPI}}((\text{double})2) \tag{5}$$

If we simulate the computation of (5) in Java using the required rounding modes we obtain the following results:

$① = (\text{double}_{\text{RTNI}})\text{Max}\, L = \text{pred}((\text{double})\text{Max}\, L) = 9223372036854774784$	safe
$② = \text{StrictMath.log}_{\text{RTNI}}(①)$	
$= 43.668272375276551144906989065930247306823730468 75$	safe
$③ = (\text{double})2 = 2.0$	exact
$④ = \text{StrictMath.log}_{\text{RTPI}}(③)$	
$= 0.693147180559945397249066445510834455490112 3046875$	safe
$⑤ = ②/_{\text{JRTNI}}\, ④ = 62.99999999999998578914528479799628257751464843 75$	safe
$⑥ = \text{StrictMath.floor}(⑤) = 62.0$	safe
$⑦ = (\text{long})\, ⑥ = 62$	safe and correct

Since the operations with the appropriate rounding modes are not available in Java we cannot use the checks for exp( ) given in the patterns (7) and (8). We use therefore the following check for the cases $L \geqslant 2$ or $L \leqslant -2$: we try to compute $\exp(L, R)$ in `long` (for `byte`, `short` and `int`) or in `double` (for `long`). As soon as the value is too large or too small we signal an error. We thus obtain the patterns (7′) and (8′):

(7′) pre-checked: $**$ for `byte`, `short`, `int` (three cases)

```
int Exp(int L, int R) {
      if (L==0 && R<0) throw new IllegalArithArgsException(IntTy, ExpOp, L, R);
      if ((L>=2 || L<=-2) && R>0)
         { long ExpVal = 1;
           for (byte i = 1; i<=R; i++)
               { ExpVal = ExpVal*L;
                 if (ExpVal>(long)MaxI || ExpVal<(long)MinI)
                    throw new IllegalArithArgsException(IntTy, ExpOp, L, R); }
           return (int)ExpVal; }
      return (int)StrictMath.pow((double)L,(double)R);
}
```

(8′) pre-checked: $**$ for `long`

```
long Exp(long L, long R) {
   if (L==0 && R<0) throw new IllegalArithArgsException(LongTy, ExpOp, L, R);
   if ((L>=2 || L<=-2) && R>0)
```

```
        { double ExpVal = 1.0;
          double DL = (double)L;
          double DMaxL = (double)MaxL; // DMaxL = MaxL+1
          double DMinL = (double)MinL; // DMinL = MinL
          for (byte i = 1; i<=R; i++)
              { ExpVal = ExpVal*DL;
                if (ExpVal>=DMaxL || ExpVal<DMinL)
                   throw new IllegalArithArgsException(LongTy, ExpOp, L, R);   }
         } // end if
      if (R==0) return 1;
      if (R==1) return L;
      if (L==0 && R>0) return 0;
      if (L==1) return 1;
      if (L==-1) return ((R%2 == 0)? 1 : -1);
      if (R < 0) return (long)1/Exp(L,-R);
      { long res = 1;
        for (byte i=1; i<=R; i++) res = res*L;
        return res; }
  }
```

Again support of the double extended format would be useful, because we could then also use pattern (7′) for `long`.

Caveat. Despite our efforts these functions are only partially safe when compared with functions in a language with range checks, as e.g. Ada. If we call Add(byte, byte) as Add($100+100$, 20) we obtain the result -36. In a language with range checking an exception will be thrown because $100+100 \notin$ `byte`.

7.2. Design of the program structure

The design of the program structure does not only fix the structure of the program but also the possible syntactic forms of the application (call) of the arithmetic functions. We will see that the rules of Java do not allow the typical infix form, which is usually used for the basic arithmetic operations in programming languages. Examples are the expressions $10000 * 100000$ and $i * 100000$.

Due to the dominance of object-orientation in Java the definitions of the functions must be components of some object type (class) TD. The functions can be defined either as static (i.e. type-specific) or as dynamic (i.e. instance-specific) components of TD.

The application of the functions will usually be in some other class TA which is different from TD. In general TA will not be a successor of TD but rather some 'foreign' class. In the following all applications of the functions are formulated as if they were in some foreign class.

7.2.1. Instance-specific functions

If we define the functions as instance-specific we can do this in two forms:

(I1) `public int Mul(int R) { ... }  // asymmetric`
(I2) `public int Mul(int L, int R) { ... }  // symmetric`

Form (I1) is more typical for instance-specific functions. The left operand is the current object (= instance) to which Mul( ) is applied. A straightforward realization of `SafeIntType` would then be


```
class SafeIntType extends int {
   public int Mul(int R) { ... }
   . . .
}
```

Unfortunately, this is not possible because `int` is not a class in Java. Java contains the class `Integer` which is a wrapper class of `int`. Using `Integer` as base type we obtain

```
class SafeIntType extends Integer {
   public int Mul(int R) { ... }
   . . .
}
```

However, this is also not legal because `Integer` is a final class.

We have therefore to implement `SafeIntType` as a new wrapper class of `int`:

```
class SafeIntType {
   public int Mul(int R) { ... }
   private int Val;
   . . .
}
```

The main drawback of this solution is that it does not allow us to use an integer literal or an `int` (or `byte`, `short`, `long`) variable as left operand, i.e. the following two applications are not legal:

```
1)     ... 10000.Mul(100000) ...
2)     int i = 5000;
       ... i.Mul(100000) ...
```

These applications have to be written as:

```
1a)     SafeIntType TenThousand = new SafeIntType(10000);
        ... TenThousand.Mul(100000) ...
2a)     SafeIntType i = new SafeIntType(5000);
        ... i.Mul(100000) ...
```

If we use the symmetric definition of Mul we obtain

```
class SafeIntType {
   public int Mul(int L, int R) { ... }
   . . .
}
```

We then have to do the two examples in the following way:

```
1b)     SafeIntType Dummy = new SafeIntType();
        ... Dummy.Mul(10000, 100000) ...
2b)     SafeIntType Dummy = new SafeIntType();
        int i = 5000;
        ... Dummy.Mul(i, 100000) ...
```

which is also a little bit cumbersome. In particular, the object `Dummy` is only needed because it is required by the language rules; apart from this it is superfluous.

7.2.2. *Type-specific functions*

If we define the functions as type-specific we can do this only in the symmetric form. We obtain

```
class SafeIntType {
   public static int Mul(int L, int R) { ... }
   . . .
}
```

and have to formulate the two examples as follows:

```
1c)     ... SafeIntType.Mul(10000, 100000) ...
2c)     int i = 5000;
        ... SafeIntType.Mul(i, 100000) ...
```

If we want to compute a more complicated expression the name of the type clutters the notation: $i * 1000 + 10 * b$ becomes

```
SafeIntType.Add(SafeIntType.Mul(i, 1000), SafeIntType.Mul(10, b) ) .
```

The only thing we can do is to chose a shorter identifier for the class, e.g. sit (= safe int type). With this we obtain

```
sit.Add(sit.Mul(i, 1000), sit.Mul(10, b) )
```

which is still more difficult to read than the infix notation. In a language with operator overloading (Algol 68 [13], Ada [14], C++ [15, 13.5]) we could define the functions in such a way that the expression could be written in the infix form $i * 1000 + 10 * b$. In Java the nearest approximation to this is Add(Mul(i, 1000), Mul(10,b)) which is only possible inside the class 'sit' or in some successor of 'sit'. Java does not contain an import clause like the use clause of Ada which allows direct access to components of foreign packages [14, 8.4]. Despite the fact that C# contains operator definitions and allows for the direct use of newly defined operators we cannot define the operators in such a way that $i * 1000 + 10 * b$ is possible. The reason is that at least one of the parameters of a binary operator must be of the type in whose definition the operator is declared. In our case the types of both parameters are predefined types [5, 17.9].

7.2.3. *One or more classes?*

The last aspect of the design of the program structure is the question whether we should group the functions into one or into different classes. There are three possibilities:

- one class for each integer type;
- one class for each operation;
- one class for all functions.

If we create one class for each integer type we obtain the following program structure:


```
package SafeIntegerArithmetic;
   classes for the new exception and for the enumeration types for the type
           indication and the kind of the operation
   class SafeByte //contains MinB, MaxB and the functions for +, −bin, −un, *, /, **
   class SafeShort
   class SafeInt
   class SafeLong
```

A typical application then looks like

```
import SafeIntegerArithmetic.*;
class TA {
      . . . SafeLong.Add(A, 2001) . . .
```

If we create one class for each operation we obtain the following program structure:

```
package SafeIntegerArithmetic
   classes for the new exception and for the enumeration types for the type
           indication, the operation kind and the constants MaxB, MinB, etc.
   class SafeAdd // contains byte Add(byte, byte), . . .
   class SafeSub
   class SafeNeg
   class SafeMul
   class SafeDiv
   class SafeExp
```

A typical application then looks like

```
import SafeIntegerArithmetic.*;
class TA {
      . . . SafeAdd.Add(A, 2001) . . .
```

If we create one class for all operations we can do it in the following way:

```
package SafeIntegerArithmetic;
   classes for the new exception and for the enumeration types for the type
           indication, the operation kind and the constants MaxB, MinB, etc.
   class sia; contains all 24 functions, sia = safe integer arithmetic
```

A typical application then looks like

```
import SafeIntegerArithmetic.*;
class TA {
       . . . sia.Add(A, 2001) . . .
```

There is one problem with this design. At least the classes `sia` and `IllegalArithArgsException` must be public. In file-based Java implementations one compilation unit can contain at most one public class [8, 7.6]. If we do not want to distribute the classes into several compilation units (= files) we can just define one class `sia` which contains the classes for the enumeration constants and for the new exception as nested classes. Such nested classes may be public or non-public. `sia` itself could be a component of the package `java.lang`. We obtain thus the following program structure:


```
package java.lang;
   public class sia {           // sia = safe integer arithmetic
      public final static class PrimTypeIndTy // definition of an enumeration type
      public final static class OperationKindTy // definition of an enumeration type
      public static class IllegalArithArgsException extends IllegalArgumentException
                        // the exception type to signal illegal arguments
      public static final byte Add(byte L, byte R)
                       // etc. until
      public static final long Exp(long L, long R)
}
```

A typical application then looks like

```
import java.lang.*; // only for documentation purposes
class TA {
       . . . sia.Add(A, 2001) . . .
```

In Appendix A, which contains all 24 functions, we use this design.

The code of the 24 functions contains much repetition and code duplication because of the similarities given by the eight similarity patterns presented in Section 7.1. The elements of one of those groups typically differ in the type of the operands and the result and by the basic operation. This can be seen in the following three functions in which the differences are highlighted:

```
public static final byte Add(byte L, byte R) {
   short result = (short)(L+R);
   if (result<MinB || result>MaxB)
      throw new IllegalArithArgsException(ByteTy, AddOp, L, R);
   else return (byte)result;
}

public static final short Add(short L, short R) {
   int result = (int)(L+R);
   if (result<MinS || result>MaxS)
      throw new IllegalArithArgsException(ShortTy, AddOp, L, R);
   else return (short)result;
}

public static final byte Mul(byte L, byte R) {
   short result = (short)(L*R);
   if (result<MinB || result>MaxB)
      throw new IllegalArithArgsException(ByteTy, AddOp, L, R);
      else return (byte)result;
}
```

There are the following differences between these three functions:

- different argument type;
- different name of the function;
- different result type;
- different type of the local variable ‘result’;
- different base operation;
- different limits (**MinB**, **MinS**, etc.);
- different enumeration constants (**ByteTy**, **AddOp**, etc.).

These kinds of variation candidates for the application of genericity. Since Java does not support genericity these similarities cannot be exploited directly. Genericity is supported by Ada and C++. We show how the nine functions of pattern (1) can be formulated as one generic function and nine instantiations. We assume that ArgResType'First $< 0 <$ ArgResType'Last holds. The functions work equally well when we always use `Long_Long_Integer` as the type of `result`. We obtain the following program, which differs from a legal Ada program only in some minor points:

```
GENERIC
   TYPE ArgResType IS Range <>;
   WITH FUNCTION BasicOp (L,R: IN ArgResType) Return ArgResType;
FUNCTION Pattern1(L, R: IN ArgResType) Return ArgResType IS
   AFirst : constant ArgResType := ArgResType'First;
   ALast : constant ArgResType := ArgResType'Last;
   SUBTYPE BigResType IS Long_Long_Integer Range -(AFirst*AFirst) .. ALast*ALast;
   Result : BigResType := BigResType(BasicOp(L, R));
BEGIN
   IF Result < AFirst OR Result > ALast
   THEN raise IllegalArithArgsException;
   ELSE return ArgResType(result);
   END IF;
END Pattern1;
```

The main difference to the Java programs is in exception handling. Ada has no facility to add additional information to an exception as is possible in Java through the use of constructors with parameters and corresponding fields in the exception type.

Using the generic function we can now create the specific functions as follows:

```
Function "+" IS new Pattern1(ArgResType => byte, BasicOp => "+");
Function "+" IS new Pattern1(ArgResType => short, BasicOp => "+");
```

etc. until

```
Function "*" IS new Pattern1(ArgResType => int, BasicOp => "*");
```

If we use the functions developed in this paper in a Java program we are faced with two problems:

- the formulas get a little bit cumbersome;
- it takes longer to evaluate a formula.

If safe arithmetic is necessary in our program there are several possibilities to avoid or at least mitigate these problems.

(a) Develop safe functions with better performance. This might be possible because performance is not the primary goal in this paper.
(b) Change the definition of Java such that the arithmetic operations are safe. The compiler can then generate more efficient code because overflow is usually signalled by contemporary processors.
(c) Include all the operations in their safe form in the processor and change Java in such a way that the arithmetic operations have the same semantics as those in the modified processor.

We prefer solutions (c) and (b) (in this order) over (a) because (a) does not avoid the negative effect on the readability of the formulas and because it may still result in some inefficiency. Last but not least (b) and (c) are far better from the language user point of view. It is far better to provide well-designed language elements, i.e. to spend the effort once in the language specification and the compiler than to put the burden onto the language users with the consequence that the effort has to be spent several times.

8. DEFICIENCIES OF Java

The following list mentions the deficiencies of the Java programming language, which we have observed in this paper:

- unsafe definition of arithmetic operations;
- only one rounding mode for arithmetic operations;
- no support for the double extended format of IEEE 754;
- no possibility to structure numeric literals, as e.g. 100_000;
- no log( ) function for `long`;
- no operator definitions;
- no enumeration types;
- no possibility to access attributes of basic types, as e.g. int'First in Ada;
- no generics;
- no aliasing, as e.g. 'renaming' in Ada;
- no use clause;
- only one public class per compilation unit.

9. CONCLUSIONS

Apart from the deficiencies of Java, which have been pointed out in Section 8, there are some more observations and conclusions, which concern programming and programming languages in general.

9.1. Finite arithmetic is different from infinite arithmetic

The definitions of arithmetic operations in finite domains are less elegant than their mathematical counterparts. Examples are the definitions given in Section 2 or the definition of StrictMath.pow( ) in Java, which consists of 17 cases [9]. Additionally, the arithmetic operations in programming languages typically do not have all the algebraic properties of their mathematical counterparts. In a language with range checking and the integer type Int32 the value of $(2\_000\_000\_000 + 2\_000\_000\_000) - 2\_000\_000\_000$ may not be 2_000_000_000.

For rational numbers these aspects of real computers have been accepted to a certain degree as the widespread implementation of IEEE 754 in microprocessors shows. However, they have not yet found their way completely into programming languages and computer science textbooks. For integer arithmetic they are not yet completely implemented in contemporary processors. The DIV and IDIV instructions for integer values of the Intel processors, e.g., provide only the rounding mode RTZ [10].

9.2. Good formulas for computations are not just transliterations of their mathematical counterparts

One example is the formula for the arithmetic mean of two floating point numbers. The mathematical definition is mean $= (a + b)/2$. If we use this formula in Ada or Java we get the correct answer only in about 75% of the possible argument pairs (a, b). If we compute the mean of n values the percentage for which the mathematical formula works on the computer decreases as n increases. The formula $a/2 + b/2$ yields the correct answer for almost all possible argument pairs, but it may give an incorrect result if one of the divisions involves rounding. One example is dPredDZero/2.0 + dPredDZero/2.0 $=$ -0.0, whereas (dPredDZero+dPredDZero)/2.0 $=$ dPredDZero, where dPredDZero is the predecessor of zero in `double` including the denormalized numbers, i.e. it is the largest `double` value less than 0.0; its value is approximately -4.9E-324 and its IEEE 754 representation as hexstring is 8000000000000001. A better but more complicated formula for the mean, which gives a reasonable result for all pairs, is given in [16]; but even Kahan's formula can be improved because it is not always commutative. This leads to an even more complicated formula [17].

Fortran programmers must pay special attention to the fact that the compiler might replace a given formula by a *mathematically* equivalent one [18, 7.1.7.3]. $a/2.0 + b/2.0$ could therefore be replaced by $(a + b)/2.0$ which is mathematically equivalent. People building 'optimizing' compilers would perhaps be proud if their compiler did such a transformation. The Fortran programmer would have to write $(a/2.0) + (b/2.0)$ because 'any expression in parentheses shall be treated as a data entity' [18, 7.1.7.2].

C and C++ programmers are also at risk because the language definitions suggest that there is no problem at all: 'the result of the binary + operator is the sum of the operands' [19, 6.3.6, 15, 5.7(3)].

In computer science textbooks we typically find the formula $(a + b)/2$ which is especially annoying because mean( ) is closed in float'min . . . float'max.

If the function is not closed, as e.g. $a + b$, then we have to live with the limitation that we cannot compute the result for all possible argument pairs. However, for closed functions computer science textbooks should contain formulas or algorithms which work for all possible argument pairs. Kahan and Darcy propose that numerically good formulas should already be presented in school books [11].

9.3. The ranges of the arguments should be adapted to the range of the result

One infamous example for this is the factorial function, which is typically specified like

```
Function Fac (n: integer) return integer
```

If we assume integer $=$ Int32 the largest value of n for which $n!$ is contained in Int32 is 12. This means a better specification is

```
Function Fac(n: Int32 range 0..12) return Int32
```

Functions can also be represented in a tabular form. In the case of the factorial with this very limited argument range the tabular form is the most efficient one [20].

9.4. What is needed in programming languages?

- Integer arithmetic with range checking and support for all four rounding modes for integer division is required.
- The definition of some operations would be simpler if the ranges of integer types were symmetric with respect to zero.
- For integer types it could also be useful to define some values which do not represent proper numbers, e.g. infinities or NaNs. Among those values could also be a value to represent the *undefined value*, which could be used to initialize variables which are not initialized by the programmer at creation time as in

```
i : integer;
```

- Mathematical functions should also be implemented for the integer types of the language because not all integer values can be represented exactly in the floating point type of the same length. This may be relevant both for the types of the arguments and for the result type. Pascal, e.g., defines exp( ) for integer and for real arguments, but the result type is always `real` [6, 6.6.6.2]. This may lead to a loss of accuracy for the result.
- The language should provide to the programmer all properties defined in IEEE 754. This can be done, e.g., by providing different modes of expression evaluation. Ada supports a strict and a relaxed mode, where the strict mode is still different from IEEE 754 [14, G.2]. In Java and C# [5] floating point arithmetic is based on IEEE 754, but as mentioned in [11] there are a number of deficiencies in the floating point arithmetic of Java. The checked mode in C# only checks range violations in integer arithmetic [5, 14.5.12]. Unfortunately, the default mode is 'unchecked'.
- Functions, that yield approximate results, should support all four rounding modes. An example for this requirement is formula (5), in which log is used with RTNI and with RTPI.

The bottom line of these requirements seems to be the observation already made by Babbage in the 19th century, 'Now it is obvious that no *finite* machine can include infinity' [21, 124], and the fact that this observation has effects on arithmetic operations in machines and programs should no longer be neglected.

APPENDIX A. THE CLASS FOR SAFE INTEGER ARITHMETIC

```
/**
 * Title         <p> Safe Integer Arithmetic
 * Description   <p>
 * Copyright     Copyright (c) <p> J F H Winkler, 2000, 2001
 * Company       <p> FSU, Jena, Germany
 * @author       J F H Winkler
 * @version      1.0
 * @date         2001.Mar.18
 */

   import java.lang.Byte;
   import java.lang.Short;
   import java.lang.Integer;
   import java.lang.Long;
```

```
   import java.lang.StrictMath;
   import java.lang.IllegalArgumentException;

public class sia{                        // sia = safe integer arithmetic

   public final static class PrimTypeIndTy {
      protected static final PrimTypeIndTy ByteTy = new PrimTypeIndTy("ByteTy");
      protected static final PrimTypeIndTy ShortTy = new PrimTypeIndTy("ShortTy");
      protected static final PrimTypeIndTy IntTy = new PrimTypeIndTy("IntTy");
      protected static final PrimTypeIndTy LongTy = new PrimTypeIndTy("LongTy");
      public String GetVal() {return this.Val;}
      private PrimTypeIndTy(String Val) {this.Val = Val;}
      private String Val;
   }

   public static final PrimTypeIndTy ByteTy  = PrimTypeIndTy.ByteTy;
   public static final PrimTypeIndTy ShortTy = PrimTypeIndTy.ShortTy;
   public static final PrimTypeIndTy IntTy   = PrimTypeIndTy.IntTy;
   public static final PrimTypeIndTy LongTy  = PrimTypeIndTy.LongTy;

   public final static class OperationKindTy {
      protected static final OperationKindTy AddOp = new OperationKindTy("AddOp");
      protected static final OperationKindTy SubOp = new OperationKindTy("SubOp");
      protected static final OperationKindTy NegOp = new OperationKindTy("NegOp");
      protected static final OperationKindTy MulOp = new OperationKindTy("MulOp");
      protected static final OperationKindTy DivOp = new OperationKindTy("DivOp");
      protected static final OperationKindTy ExpOp = new OperationKindTy("ExpOp");
      public String GetVal() {return this.Val;}
      private OperationKindTy(String Val) {this.Val = Val;}
      private String Val;
   }
   public static final OperationKindTy AddOp = OperationKindTy.AddOp;
   public static final OperationKindTy SubOp = OperationKindTy.SubOp;
   public static final OperationKindTy NegOp = OperationKindTy.NegOp;
   public static final OperationKindTy MulOp = OperationKindTy.MulOp;
   public static final OperationKindTy DivOp = OperationKindTy.DivOp;
   public static final OperationKindTy ExpOp = OperationKindTy.ExpOp;

   public static class IllegalArithArgsException extends IllegalArgumentException {
     public IllegalArithArgsException (PrimTypeIndTy type, OperationKindTy operation,
                                       long L, long R) {
        this.TypeIndication = type;
        this.OperationKind = operation;
        this.L = L; this.R = R;
        System.out.println("illegal args: " +
                           "op = " + OperationKind.GetVal() +
                           ", type = " + TypeIndication.GetVal() +
                           ", L = " + L + ", R = " + R);
     }

     public IllegalArithArgsException (PrimTypeIndTy type,
                                       OperationKindTy operation, long L) {
        this.TypeIndication = type;
        this.OperationKind = operation;
        this.L = L;
        System.out.println("illegal args: " +
                           "op = " + OperationKind.GetVal() +
                           ", type = " + TypeIndication.GetVal() + ", L = " + L );
```



```
  }

  public PrimTypeIndTy GetType() { return this.TypeIndication; }
  public OperationKindTy GetOperation() { return this.OperationKind; }
  public long GetLeftOp() { return this.L; }
  public long GetRightOp() { return this.R; }

  protected PrimTypeIndTy TypeIndication;
  protected OperationKindTy OperationKind;
  protected long L;
  protected long R;
}

public static final byte  MaxB = java.lang.Byte.MAX_VALUE;
public static final byte  MinB = java.lang.Byte.MIN_VALUE;
public static final short MaxS = java.lang.Short.MAX_VALUE;
public static final short MinS = java.lang.Short.MIN_VALUE;
public static final int   MaxI = java.lang.Integer.MAX_VALUE;
public static final int   MinI = java.lang.Integer.MIN_VALUE;
public static final long  MaxL = java.lang.Long.MAX_VALUE;
public static final long  MinL = java.lang.Long.MIN_VALUE;


// ------------------------------ ADDITION ------------------------------

public static final byte Add(byte L, byte R) {
   short result = (short)(L+R);
   if (result<MinB || result>MaxB)
      throw new IllegalArithArgsException(ByteTy, AddOp, L, R);
   else return (byte)result;
}

public static final short Add(short L, short R) {
   int result = (int)(L+R);
   if (result<MinS || result>MaxS)
      throw new IllegalArithArgsException(ShortTy, AddOp, L, R);
   else return (short)result;
}

public static final int Add(int L, int R) {
   long result = (long)L + (long)R;
   if (result<MinI || result>MaxI)
      throw new IllegalArithArgsException(IntTy, AddOp, L, R);
   else return (int)result;
}

public static final long Add(long L, long R) {
   long result = L+R;
   if (L>0 && R>0 && result<0 || L<0 && R<0 && result>=0)
      throw new IllegalArithArgsException(LongTy, AddOp, L, R);
   else return result;
}


// ------------------------------ SUBTRACTION ------------------------

public static final byte Sub(byte L, byte R) {
   short result = (short)(L-R);
```



```
   if (result<MinB || result>MaxB)
      throw new IllegalArithArgsException(ByteTy, AddOp, L, R);
   else return (byte)result;
}

public static final short Sub(short L, short R) {
   int result = (int)(L-R);
   if (result<MinS || result>MaxS)
      throw new IllegalArithArgsException(ShortTy, AddOp, L, R);
   else return (short)result;
}

public static final int Sub(int L, int R) {
   long result = (long)L - (long)R;
   if (result<MinI || result>MaxI)
      throw new IllegalArithArgsException(IntTy, AddOp, L, R);
   else return (int)result;
}

public static final long Sub(long L, long R) {
   long result = L-R;
   if (L>=0 && R<0 && result<0 || L<0 && R>0 && result>0)
      throw new IllegalArithArgsException(LongTy, SubOp, L, R);
   else return result;
}


// ------------------------------ NEGATION -------------------------

public static final byte Neg(byte L) {
   if (L==MinB)
      throw new IllegalArithArgsException(ByteTy, NegOp, L);
   else return (byte)(-L);
}

public static final short Neg(short L) {
   if (L==MinS)
      throw new IllegalArithArgsException(ShortTy, NegOp, L);
   else return (short)(-L);
}

public static final int Neg(int L) {
   if (L==MinI)
      throw new IllegalArithArgsException(IntTy, NegOp, L);
   else return (int)(-L);
}

public static final long Neg(long L) {
   if (L==MinL)
      throw new IllegalArithArgsException(LongTy, NegOp, L);
   else return (long)(-L);
}


// ------------------------------ MULTIPLICATION ------------------------

public static final byte Mul(byte L, byte R) {
   short result = (short)(L*R);
```

```
   if (result<MinB || result>MaxB)
      throw new IllegalArithArgsException(ByteTy, AddOp, L, R);
   else return (byte)result;
}

 public static final short Mul(short L, short R) {
    int result = (int)(L*R);
    if (result<MinS || result>MaxS)
       throw new IllegalArithArgsException(ShortTy, AddOp, L, R);
    else return (short)result;
}

public static final int Mul(int L, int R) {
   long result = (long)L * (long)R;
   if (result<MinI || result>MaxI)
      throw new IllegalArithArgsException(IntTy, AddOp, L, R);
   else return (int)result;
}

public static final long Mul(long L, long R) {
   if ( ( L>0 && (R<MinL/L || R>MaxL/L) ) ||
        ( L==-1 && R==MinL) || (L<-1 && (R<MaxL/L || R>MinL/L) ) )
      throw new IllegalArithArgsException(LongTy, MulOp, L, R);
   else return L*R;
}


// ------------------------------ DIVISION -------------------------

public static final byte Div(byte L, byte R) {
   if (L==MinB && R==-1)
      throw new IllegalArithArgsException(ByteTy, DivOp, L, R);
   else return (byte)(L/R);
}

public static final short Div(short L, short R) {
   if (L==MinS && R==-1)
      throw new IllegalArithArgsException(ShortTy, DivOp, L, R);
   else return (short)(L/R);
}

public static final int Div(int L, int R) {
   if (L==MinI && R==-1)
      throw new IllegalArithArgsException(IntTy, DivOp, L, R);
   else return (int)(L/R);
}

public static final long Div(long L, long R) {
   if (L==MinL && R==-1)
      throw new IllegalArithArgsException(LongTy, DivOp, L, R);
   else return L/R;
}


// ------------------------------ EXPONENTIATION ----------------------

public static byte Exp(byte L, byte R) {
   if (L==0 && R<0) throw new IllegalArithArgsException(ByteTy, ExpOp, L, R);
```



```
      if ((L>=2 || L<=-2) && R>0)
         { short ExpVal = 1;
           for (byte i = 1; i<=R; i++)
               { ExpVal = (short)(ExpVal*L);
                 if (ExpVal>(short)MaxB || ExpVal<(short)MinB)
                    throw new IllegalArithArgsException(ByteTy, ExpOp, L, R); }
           return (byte)ExpVal; }
       return (byte)StrictMath.pow((double)L,(double)R);
   }

   public static short Exp(short L, short R) {
      if (L==0 && R<0) throw new IllegalArithArgsException(ShortTy, ExpOp, L, R);
      if ((L>=2 || L<=-2) && R>0)
         { int ExpVal = 1;
           for (byte i = 1; i<=R; i++)
               { ExpVal = ExpVal*L;
                 if (ExpVal>(int)MaxS || ExpVal<(int)MinS)
                    throw new IllegalArithArgsException(ShortTy, ExpOp, L, R); }
           return (short)ExpVal; }
      return (short)StrictMath.pow((double)L,(double)R);
   }

   public static int Exp(int L, int R) {
       if (L==0 && R<0) throw new IllegalArithArgsException(IntTy, ExpOp, L, R);
       if ((L>=2 || L<=-2) && R>0)
          { long ExpVal = 1;
            for (byte i = 1; i<=R; i++)
                { ExpVal = ExpVal*L;
                  if (ExpVal>(long)MaxI || ExpVal<(long)MinI)
                     throw new IllegalArithArgsException(IntTy, ExpOp, L, R); }
            return (int)ExpVal; }
       return (int)StrictMath.pow((double)L,(double)R);
   }

   public static long Exp(long L, long R) {
      if (L==0 && R<0) throw new IllegalArithArgsException(LongTy, ExpOp, L, R);
      if ((L>=2 || L<=-2) && R>0)
         { double ExpVal = 1.0;
           double DL = (double)L;
           double DMaxL = (double)MaxL; // DMaxL = MaxL+1
           double DMinL = (double)MinL; // DMinL = MinL
           for (byte i = 1; i<=R; i++)
               { ExpVal = ExpVal*DL;
                 if (ExpVal>=DMaxL || ExpVal<DMinL)
                    throw new IllegalArithArgsException(LongTy, ExpOp, L, R); }
         } // end if
      if (R==0) return 1;
      if (R==1) return L;
      if (L==0 && R>0) return 0;
      if (L==1) return 1;
      if (L==-1) return ((R%2 == 0)? 1 : -1);
      if (R < 0) return (long)1/Exp(L,-R);
      { long res = 1;
        for (byte i=1; i<=R; i++) res = res*L;
        return res; }
   }
} // end of class sia
```


ACKNOWLEDGEMENT

I am very grateful to Stefan Kauer for several hints for improvement of an earlier version of this paper.

REFERENCES

1. Hoare CAR. An axiomatic basis for computer programming. *Communications of the ACM* 1969; **12**(10):576–580, 583.
2. Winkler JFH, Kauer S. Proving assertions is also useful. *SIGPLAN Notices* 1997; **32**(3):38–41.
3. Winkler JFH. Functions not equivalent. *IEEE Software* 1990; **7**(3):10.
4. International Standard ISO/IEC 7185-1:1990(E). *Information Technology—Programming Languages—Pascal* (2nd edn). ISO/IEC: Geneva, 1990.
5. Gosling J, Joy B, Steele GL, Bracha G. *The Java Language Specification* (2nd edn). Addison-Wesley: Boston, MA, 2000.
6. ECMA. *C# Language Specification*, Standard ECMA-334, December 2001.
7. Intel Corporation. *Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture*. 1997.
8. Gosling J, Joy B, Steele GL, Bracha G. *The Java Language Specification* (2nd edn). http://java.sun.com/docs/books/jls/index.html [10 August 2000].
9. Java 2 Platform SE v. 1.3: Class StrictMath. http://java.sun.com/j2se/1.3/docs/api/java/lang/StrictMath.html [18 October 2000].
10. Intel Corporation. *Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference*. 1999.
11. Kahan W, Darcy JD. How Java's floating-point hurts everyone everywhere. http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf [4 November 2000].
12. Cairns PA. Enumerated types in Java. *Software—Practice and Experience* 1999; **29**(3):291–297.
13. van Wijngaarden A (ed.). Report on the algorithmic language ALGOL 68. *Numerische Mathematik* 1969; **14**:79–218.
14. International Standard ISO/IEC 8652:1995(E). *Information Technology—Programming Languages—Ada* (2nd edn). ISO/IEC: Geneva, 1995.
15. International Standard ISO/IEC 14882:1998(E). *Programming Languages—C++* (1st edn). ISO/IEC: Geneva, 1998
16. Kahan W. Analysis and refutation of the LCAS. *SIGPLAN Notices* 1992; **27**(1):61–74.
17. Winkler JFH. Characteristics of computing and informatics. *Internal Paper*, Friedrich-Schiller University, Institute of Computer Science, 2001.
18. International Standard ISO/IEC 1539-1:1997(E). *Information Technology—Programming Languages—Fortran. Part 1: Base Language* (1st edn). ISO/IEC: Geneva, 1997.
19. International Standard ISO/IEC 9899:1990(E). *Programming Languages—C* (1st edn). ISO/IEC: Geneva, 1990.
20. Winkler JFH, Nievergelt J. Wie soll die Fakultätsfunktion programmiert werden? (How should the factorial function be implemented?). *Informatik-Spektrum* 1989; **12**(4):220–221.
21. Babbage C. *Passages from a Life of a Philosopher*. Longman, Green, Longman, Roberts and Green: London, 1864. Cambell-Kelly M (ed.). *The Works of Charles Babbage*, vol. 11. William Pickering: London, 1989.

