

Formalizing Java's Two's-Complement Integral Type in Isabelle/HOL

Nicole Rauch¹

Universität Kaiserslautern, Germany

Burkhart Wolff

Albert-Ludwigs-Universität Freiburg, Germany

Abstract

We present a formal model of the Java two's-complement integral arithmetics. The model directly formalizes the arithmetic operations as given in the Java Language Specification (JLS). The algebraic properties of these definitions are derived. Underspecifications and ambiguities in the JLS are pointed out and clarified. The theory is formally analyzed in Isabelle/HOL, that is, machine-checked proofs for the ring properties and divisor/remainder theorems etc. are provided. This work is suited to build the framework for machine-supported reasoning over arithmetic formulae in the context of Java source-code verification.

Key words: Java, Java Card, formal semantics, formal methods, tools, theorem proving, integer arithmetic.

1 Introduction

Admittedly, modelling numbers in a theorem prover is not really a “sexy subject” at first sight. Numbers are fundamental, well-studied and well-understood, and everyone is used to them since school-mathematics. Basic theories for the naturals, the integers and real numbers are available in all major theorem proving systems (e.g. [11,26,21]), so why care?

However, numbers as specified in a concrete processor or in a concrete programming language semantics are oriented towards an efficient implementation on a machine. They are finite datatypes and typically based on bit-fiddling definitions. Nevertheless, they often possess a surprisingly rich theory (ring properties, for example) that also comprises a number of highly non-standard and tricky laws with non-intuitive and subtle preconditions.

¹ Partially funded by IST VerifiCard (IST-2000-26328)

In the context of program verification tools (such as the B tool [2], KIV [3], LOOP [5], and Jive [20], which directly motivated this work), efficient numerical programs, e.g. square roots, trigonometric functions, fast Fourier transformation or efficient cryptographic algorithms represent a particular challenge. Fortunately, theorem proving technology has matured to a degree that the analysis of realistic machine number specifications for widely-used programming languages such as Java or C now is a routine task [13].

With respect to the formalization of integers, we distinguish two approaches:

- (1) the *partial approach*: the arithmetic operations $+$ $-$ $*$ $/$ $\%$ are only defined on an interval of (mathematical) integers, and left undefined whenever the result of the operation is outside the interval (c.f. [4], which is mainly geared towards this approach).
- (2) the *wrap-around approach*: integers are defined on $[-2^{n-1} .. 2^{n-1} - 1]$, where in case of overflow the results of the arithmetic operations are mapped back into this interval through modulo calculations. These numbers can be equivalently realized by bitstrings of length n in the widely-used two's-complement representation system [10].

While in the formal methods community there is a widespread reluctance to integrate machine number models and therefore a tendency towards either (infinite) mathematical integers or the first approach (“either remain fully formal but focus on a smaller or simpler language [...]; or remain with the real language, but give up trying to achieve full formality.” [23]), we strongly argue in favour of the second approach for the following reasons:

- (1) In a wrap-around implementation, certain properties like “Maxint + 1 = Minint” hold. This has the consequence that crucial algebraic properties such as the associativity law “ $a + (b + c) = (a + b) + c$ ” hold in the wrap-around approach, but not in the partial approach. The wrap-around approach is therefore more suited for automated reasoning.
- (2) Simply using the mathematical operators on a subset of the mathematical integers does not handle surprising definitions of operators appropriately. E.g. in Java the result of an integer division is always rounded towards zero, and thus the corresponding modulo operation can return negative values. This is unusual in mathematics. Therefore, this naïve approach does not only disregard overflows and underflows but also disregards unconventionally defined operators.
- (3) The Java type `int` is defined in terms of wrap-around in the Java Language Specification [12], so why should a programmer who strictly complies to it in an efficient program be punished by the lack of formal analysis tools?
- (4) Many parts of the JLS have been analyzed formally — so why not the part concerning number representations? There are also definitions and claimed properties that should be checked; and there are also possible inconsistencies or underspecifications as in all other informal specifications.

As technical framework for our analysis we use Isabelle/HOL and the Isar proof documentation package, whose output is directly used throughout this paper (for lack of space, however, we will not present any proofs here. The complete documentation will be found in a forthcoming technical report). Isabelle [21] is a *generic* theorem prover, i.e. new object logics can be introduced by specifying their syntax and inference rules. Isabelle/HOL is an instance of Isabelle with Church’s *higher-order logic* (HOL) [11], a classical logic with equality enriched by total polymorphic higher-order functions. In HOL, induction schemes can be expressed inside the logic, as well as (total) functional programs. Isabelle’s methodology for safely building up large specifications is the decomposition of problems into *conservative extensions*. A conservative extension introduces new constants (by *constant definitions*) and types (by *type definitions*) only via axioms of a particular, machine-checked form; a proof that conservative extensions preserve consistency can be found in [11]. Among others, the HOL library provides conservative theories for the logical type *bool*, for the numbers such as *int* and for bitstrings *bin*.

1.1 Related Work

The formalization of IEEE floating point arithmetics has attracted the interest of researchers for some time [8,1], e.g. leading to concrete, industry strength verification technologies used in Intel’s IA64 architecture [13].

In hardware verification, it is a routine task to verify two’s complement number operations and their implementations on the gate level. Usually, variants of *binary decision diagrams* are used to represent functions over bit words canonically; thus, if a trusted function representation is identical to one generated from a highly complex implementation, the latter is verified. Meanwhile, addition, multiplication and restricted forms of operations involving division and remainder have been developed [15]. Unfortunately, it is known that one variant particularly suited for one operation is inherently intractable for another, etc. Moreover, general division and remainder functions have been proven to be intractable by word-level decision diagrams (WLDD) [24]. For all these reasons, the approach is unsuited to investigate the *theory* of two’s complement numbers: for example, the theorem `JavaInt-div-mod` (see Section 4.2), which involves a mixture of all four operations, can only be proven up to a length of 9 bits, even with leading edge technology WLDD packages².

Amazingly, *formalized theories* of the two’s complement number have only been considered recently; i.e. Fox formalized 32-bit words and the ARM processor for HOL [9], and Bondyfalat developed a (quite rudimentary) bit words theory with division in the AOC project [6]. In the context of Java and the JLS, Jacobs [16] presented a fragment of the theory of integral types. This work (like ours) applies to Java Card as well since the models of the four smaller integral types (excluding `long`) of Java and Java Card are identical

² Thanks to Marc Herbstritt [14] to check this for us!

[25, § 2.2.3.1]. However, although our work is in spirit and scope very similar to [16], there are also significant differences:

- We use standard integer intervals as reference model for the arithmetic operations as well as two’s-complement bitstrings for the bitshift and the bitwise AND, OR, XOR operations (which have not been covered by [16] anyway). Where required, we prove lemmas that show the isomorphy between these two representations.
- While [16] just presents the normal behavior of arithmetic expressions, we also cover the exceptional behavior for expressions like “x / 0” by adding a second theory layer with so-called strictness principles (see Sect. 6).
- [16] puts a strong emphasis on widening and narrowing operations which are required for the Java types `short` and `byte`. We currently only concentrate on the type `int` and therefore did not model widening and narrowing yet.

The Java Virtual Machine (JVM) [19] has been extensively modelled in the Project Bali [22]. However, the arithmetic operations in this JVM model are based on mathematical integers. Since our work is based on the same system, namely Isabelle2002, our model of a two’s-complement integer datatype could replace the mathematical integers in this JVM theory.

1.2 Outline of this Paper

Section 2 introduces the core conservative definitions and the addition and multiplication, Section 3 presents the division and remainder theory and Section 4 gives the bitwise operations. Sections 2, 3 and 4 examine the *normal behavior*, while Section 5 describes the introduction of *exceptional behavior* into our arithmetic theory leading to operations which can deal with exceptions that may occur during calculations.

2 Formalizing the Normal Behavior Java Integers

The formalization of Java integers models the primitive Java type `int` as closely as possible. The programming language Java comes with a quite extensive language specification [12] which tries to be accurate and detailed. Nonetheless, there are several white spots in the Java integer specification which are pointed out in this paper. The language Java itself is platform-independent. The bit length of the data type `int` is fixed in a machine-independent way. This simplifies the modelling task. The JLS states about the integral types:

Java Language Specification [12], §4.2

“The integral types are byte, short, int, and long, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two’s-complement integers, respectively, and char, whose values are 16-bit unsigned integers representing Unicode characters. [...] The values of the integral types are integers in the following ranges: [...] For int, from −2147483648 to 2147483647, inclusive”

The Java `int` type and its range are formalized in Isabelle/HOL [21] this way:

constdefs

BitLength :: nat	BitLength $\equiv$ 32
MinInt-int :: int	MinInt-int $\equiv - (2 ^ (\text{BitLength} - 1))$
MaxInt-int :: int	MaxInt-int $\equiv 2 ^ (\text{BitLength} - 1) - 1$

Now we can introduce a new type for the desired integer range:

typedef Javalnt = {i. MinInt-int $\leq$ i $\wedge$ i $\leq$ MaxInt-int}

This construct is the Isabelle/HOL shortcut for a type definition which defines the new type **Javalnt** isomorphic to the set of integers between MinInt-int and MaxInt-int. The isomorphism is established through the automatically provided (total) functions **Abs-Javalnt** :: int $\Rightarrow$ **Javalnt** and **Rep-Javalnt** :: **Javalnt** $\Rightarrow$ int and the two axioms $y : \{i. \text{MinInt-int} \leq i \wedge i \leq \text{MaxInt-int}\} \implies \text{Rep-Javalnt} (\text{Abs-Javalnt } y) = y$ and $\text{Abs-Javalnt} (\text{Rep-Javalnt } x) = x$. **Abs-Javalnt** yields an arbitrary value if the argument is outside of the defining interval of **Javalnt**.

We define **MinInt** and **MaxInt** to be elements of the new type **Javalnt**:

constdefs

MinInt :: Javalnt	MinInt $\equiv$ Abs-Javalnt MinInt-int
MaxInt :: Javalnt	MaxInt $\equiv$ Abs-Javalnt MaxInt-int

In Java, calculations are only performed on values of the types **int** and **long**. Values of the three smaller integral types are widened first:

Java Language Specification [12], §4.2.2

“If an integer operator other than a shift operator has at least one operand of type **long**, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type **long**. If the other operand is not **long**, it is first widened (§5.1.4) to type **long** by numeric promotion (§5.6). Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type **int**. If either operand is not an **int**, it is first widened to type **int** by numeric promotion. The built-in integer operators do not indicate overflow or underflow in any way.”

This paper describes the formalization of the Java type **int**, therefore conversions between the different numerical types are not in the focus of this work. The integer types **byte** and **short** can easily be added as all calculations are performed on the type **int** anyways, so the only operations that need to be implemented are the widening to **int**, and the cast operations from **int** to **byte** and **short**, respectively. The Java type **long** can be added equally easily as our theory uses the bit length as a parameter, so one only need to change the definition of the bit length (see above) to gain a full theory for the Java type **long**, and again only the widening operations need to be added. Therefore, we only concentrate on the Java type **int** in the following.

Our model of Java **int** covers all side-effect-free operators. This excludes the operators **++** and **--**, both in pre- and postfix notation. These operators return the value of the variable they are applied to while modifying the value stored in that variable independently from returning the value. We do not treat assignment of any kind either as it represents a side-effect as well. This

also disallows combined operators like $a += b$ etc. which are a shortcut for $a = a + b$. This is in line with usual specification languages, e.g. JML [17], which also allows only side-effect-free operators in specifications. From a logical point of view, this makes sense as the specification is usually regarded as a set of predicates. In usual logics, predicates are side-effect-free. Thus, expressions with side-effects must be treated differently, either by special Hoare rules or by program transformation.

In our model, all operators are defined in Isabelle/HOL, and their properties as described in the JLS are proven, which ensures the validity of the definitions in our model. In the following, we quote the definitions from the JLS and present the Isabelle/HOL definitions and lemmas.

Our standard approach of defining the arithmetic operators on **JavaInt** is to convert the operands from **JavaInt** to Isabelle `int`, to apply the corresponding Isabelle `int` operation, and to convert the result back to **JavaInt**. The first conversion is performed by the representation function `Rep-JavaInt` (see above). The inverse conversion is performed by the function `Int-to-JavaInt`:

`Int-to-JavaInt :: int  $\Rightarrow$  JavaInt`

`Int-to-JavaInt (x::int) $\equiv$ Abs-JavaInt(
 ((x + ($-\text{MinInt}$))) mod (2 * ($-\text{MinInt}$))) + MinInt)`

This function first adds ($-\text{MinInt}$) to the argument and then performs a modulo calculation by $2 * (-\text{MinInt})$ which maps the value into the interval $[0 .. 2 * (-\text{MinInt}) - 1]$ (which is equivalent to only regarding the lowest 32 bits), and finally subtracts the value that was initially added. This definition is identical to the function `Abs-JavaInt` on arguments which are already in **JavaInt**. Larger or smaller values are mapped to **JavaInt** values, extending the domain to `int`.

This standard approach is not followed for operators that are explicitly defined on the bit representation of the arguments. Our approach differs from the approach used by Jacobs [16] who exclusively uses bit representations for the integer representation as well as the operator definitions.

2.1 Unary Operators

This section gives the formalizations of the unary operators $+$, $-$ and the bitwise complement operator $\sim$. The unary plus operator on `int` is equivalent to the identity function. This is not very challenging, thus we do not elaborate on this operator. In the JLS, the unary minus operator is defined in relation to the binary minus operator described below.

Java Language Specification [12], §15.15.4

“At run time, the value of the unary minus expression is the arithmetic negation of the promoted value of the operand. For integer values, negation is the same as subtraction from zero.⁽¹⁾

[...] negation of the maximum negative int or long results in that same maximum negative number.⁽²⁾

[...] For all integer values x , $-x$ equals $(\sim x) + 1$.⁽³⁾”

The unary minus operator is formalized as

$\text{uminus-def} : - (x::\text{JavaInt}) \equiv \text{Int-to-JavaInt} (- \text{Rep-JavaInt } x)$

We prove the three properties described in the JLS:

- (1) **lemma** uminus-property: $0 - x = - (x::\text{JavaInt})$
- (2) **lemma** uminus-MinInt: $- \text{MinInt} = \text{MinInt}$
- (3) **lemma** uminus-bitcomplement: $(\sim x) + 1 = - x$

Note that the unary minus operator has two fixed points: 0 and MinInt. This leads some unexpected results, e.g. $\text{Math.abs}(\text{MinInt}) = \text{MinInt}$, a negative number. Also, many of the lemmas presented in this paper do not hold for MinInt and therefore exclude that value in their assumptions.

The bitwise complement operator is defined by unary and binary minus:

Java Language Specification [12], §15.15.5

“At run time, the value of the unary bitwise complement expression is the bitwise complement of the promoted value of the operand; note that, in all cases, $\sim x$ equals $(-x)-1$.”

This is formalized in Isabelle/HOL as follows:

constdefs

$\text{JavaInt-bitcomplement} :: \text{JavaInt} \Rightarrow \text{JavaInt}$

$\text{JavaInt-bitcomplement } (x::\text{JavaInt}) \equiv (-x) - (1::\text{JavaInt})$

We use the notations $\sim$ and JavaInt-bitcomplement interchangeably.

3 Additive and Multiplicative Operators

3.1 Additive Operators

This section formalizes the binary $+$ operator and the binary $-$ operator.

Java Language Specification [12], §15.18.2

“The binary $+$ operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The binary $-$ operator performs subtraction, producing the difference of two numeric operands.⁽¹⁾ [...] Addition is a commutative operation if the operand expressions have no side effects. Integer addition is associative when the operands are all of the same type⁽²⁾ [...] If an integer addition overflows, then the result is the low-order bits of the mathematical sum as represented in some sufficiently large two’s-complement format.⁽³⁾ If overflow occurs, then the sign of the result is not the same as the sign of the mathematical sum of the two operand values.⁽⁴⁾ For both integer and floating-point subtraction, it is always the case that $a-b$ produces the same result as $a+(-b)$.⁽⁵⁾”

- (1) These two operators are defined in the standard way described above. We only give the definition of the binary $+$ operator:
defs (overloaded)
 $\text{add-def} : x + y \equiv \text{Int-to-JavaInt } (\text{Rep-JavaInt } x + \text{Rep-JavaInt } y)$
- (2) This behavior is captured by the two lemmas

lemma JavaInt-add-commute: $x + y = y + (x::\text{JavaInt})$

lemma JavaInt-add-assoc: $x + y + z = x + (y + z::\text{JavaInt})$

(3) This requirement is already fulfilled by the definition.

(4) This specification can be expressed as

lemma JavaInt-add-overflow-sign :

$(\text{MaxInt-int} < \text{Rep-JavaInt } a + \text{Rep-JavaInt } b) \longrightarrow (a + b < 0)$

This is a good example of how inexact several parts of the Java Language Specification are. If indeed only overflow, i.e. regarding two operands whose sum is larger than `MaxInt`, is meant here, then why pose such a complicated question? “the sign of the mathematical sum of the two operand values” will always be positive in this case, so why talk about “the sign of the result is not the same”? It would be much clearer to state “the sign of the result is always negative”. But what if the authors also wanted to describe underflow, i.e. negative overflow, which is sometimes also referred to as “overflow”? In §4.2.2 the JLS states “The built-in integer operators do not indicate overflow or underflow in any way.” Thus, the term “underflow” is known to the authors and is used in the JLS. Why do they not use it in the context quoted above? This would also explain the complicated phrasing of the above formulation.

To clarify these matters, we add the lemma

lemma JavaInt-add-underflow-sign :

$(\text{Rep-JavaInt } a + \text{Rep-JavaInt } b < \text{MinInt-int}) \longrightarrow (0 \leq a + b)$

(5) This has been formalized as

lemma diff-uminus: $a - b = a + (-b::\text{JavaInt})$

3.2 Multiplication Operator

The multiplication operator is described and formalized as follows:

Java Language Specification [12], §15.17.1

“The binary `*` operator performs multiplication, producing the product of its operands. Multiplication is a commutative operation if the operand expressions have no side effects. [...] integer multiplication is associative when the operands are all of the same type”

defs (overloaded)

`times-def` : $x * y \equiv \text{Int-to-JavaInt } (\text{Rep-JavaInt } x * \text{Rep-JavaInt } y)$

The commutativity and associativity are proven by the lemmas

lemma JavaInt-times-commute: $(x::\text{JavaInt}) * y = y * x$

lemma JavaInt-times-assoc: $(x::\text{JavaInt}) * y * z = x * (y * z)$

Java Language Specification [12], §15.17.1

“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.”

This is again implicitly fulfilled by our standard modelling.

4 Division and Remainder Operators

4.1 Division Operator

In Java, the division operator produces the first surprise if compared to the mathematical definition of division, which is also used in Isabelle/HOL:

Java Language Specification [12], §15.17.2

“The binary / operator performs division, producing the quotient of its operands. [...] Integer division rounds toward 0. That is, the quotient produced for operands n and d that are integers after binary numeric promotion (§5.6.2) is an integer value q whose magnitude is as large as possible while satisfying $|d \times q| \leq |n|$; moreover, q is positive when $|n| \geq |d|$ and n and d have the same sign, but q is negative when $|n| \geq |d|$ and n and d have opposite signs.⁽¹⁾ There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for its type, and the divisor is -1, then integer overflow occurs and the result is equal to the dividend.⁽²⁾ Despite the overflow, no exception is thrown in this case. On the other hand, if the value of the divisor in an integer division is 0, then an `ArithmeticException` is thrown.⁽³⁾”

This definition points out a major difference between the definition of division in Isabelle/HOL and Java. If the signs of dividend and divisor are different, the results differ by one because Java rounds towards 0 whereas Isabelle/HOL floors the result. Thus, the naïve approach of modelling Java integers by partialization of the corresponding operations of a theorem prover gives the wrong results in these cases.

We model the division by performing case distinctions:

defs (overloaded)

```
div-def : (x::JavaInt) div y ≡
  if ((0 < x ∧ y < 0) ∨ (x < 0 ∧ 0 < y))
    ∧ ¬ (∃ z. Rep-JavaInt x = z * Rep-JavaInt y) then
    Int-to-JavaInt ( Rep-JavaInt x div Rep-JavaInt y ) + 1
  else
    Int-to-JavaInt( Rep-JavaInt x div Rep-JavaInt y )
```

The properties mentioned in the language report are formalized as follows:

(1) **lemma** quotient-sign-plus :

$$\begin{aligned} & ((\text{abs } d \leq \text{abs } n) \vee (n = \text{MinInt})) \wedge (n \neq \text{MinInt} \vee d \neq -1) \\ & \wedge (\text{neg } (\text{Rep-JavaInt } n) = \text{neg } (\text{Rep-JavaInt } d)) \wedge d \neq 0 \\ & \implies 0 < (n \text{ div } d) \end{aligned}$$

lemma quotient-sign-minus :

$$\begin{aligned} & ((\text{abs } d \leq \text{abs } n) \vee (n = \text{MinInt})) \wedge (n \neq \text{MinInt} \vee d \neq -1) \\ & \wedge (\text{neg } (\text{Rep-JavaInt } n) \neq \text{neg } (\text{Rep-JavaInt } d)) \wedge d \neq 0 \\ & \implies (n \text{ div } d) < 0 \end{aligned}$$

The predicate “neg” holds iff the value of its argument is less than zero. We have to treat the case $n = \text{MinInt}$ separately because the `abs` function on `JavaInt` produces an unusable result for `MinInt` (see above).

Again, the phrasing in the JLS is quite imprecise as the “one special

case that does not satisfy this rule” refers to both of the lemmas above.

- (2) **lemma** `JavaInt-div-minusone` : `MinInt div -1 = MinInt`
- (3) is not modelled by the theory presented in this section because this theory does not introduce a bottom element for integers in order to treat exceptional cases. Our model returns 0 in this case due to the definition of the total function `div` in Isabelle/HOL. Exceptions are handled by the next theory layer (see Sect. 6) which adds a bottom element to `JavaInt` and lifts all operations in order to treat exceptions appropriately.

Again, the JLS is not very elaborate in (2) regarding the sign of the resulting value if the magnitude of the dividend is less than the magnitude of the divisor. It would have been clearer had they stated the result instead of letting the reader derive the result from the presented inequalities.

4.2 Remainder Operator

The remainder operator is closely related to the division operator. Thus, it does not conform to standard mathematical definitions either.

Java Language Specification [12], §15.17.3

“The binary `%` operator is said to yield the remainder of its operands from an implied division [...] The remainder operation for operands that are integers after binary numeric promotion (§5.6.2) produces a result value such that $(a/b) * b + (a \% b)$ is equal to a .⁽¹⁾

This identity holds even in the special case that the dividend is the negative integer of largest possible magnitude for its type and the divisor is -1 (the remainder is 0).⁽²⁾

It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative, and can be positive only if the dividend is positive;⁽³⁾

moreover, the magnitude of the result is always less than the magnitude of the divisor.⁽⁴⁾

If the value of the divisor for an integer remainder operator is 0, then an `ArithmeticException` is thrown.⁽⁵⁾

Examples: `5%3` produces 2 (note that `5/3` produces 1)
`5%(-3)` produces 2 (note that `5/(-3)` produces -1)
`(-5)%3` produces -2 (note that `(-5)/3` produces -1)
`(-5)%(-3)` produces -2 (note that `(-5)/(-3)` produces 1)⁽⁶⁾”

When formalizing the remainder operator, we have to keep in mind the formalization of the division operator and the required equality (1). Therefore, the remainder operator `mod` is formalized as follows:

```
mod-def : (x::JavaInt) mod y ≡
  if ((0 < x ∧ y < 0) ∨ (x < 0 ∧ 0 < y))
    ∧ ¬ (∃ z. Rep-JavaInt x = z * Rep-JavaInt y) then
      Int-to-JavaInt( Rep-JavaInt x mod Rep-JavaInt y ) - y
  else
      Int-to-JavaInt( Rep-JavaInt x mod Rep-JavaInt y )
```

The formulations in the JLS give rise to the following lemmas:

- (1) **lemma** JavaInt-div-mod : $((a::\text{JavaInt}) \text{ div } b) * b + (a \text{ mod } b) = a$
- (2) **lemma** MinInt-mod-minusone: $\text{MinInt mod } -1 = 0$
lemma MinInt-minusone-div-mod-eq :
 $(\text{MinInt div } -1) * (-1) + (\text{MinInt mod } -1) = \text{MinInt}$
- (3) These phrases are not at all clear to us. We formalized them as follows, in the hope of meeting the intentions of the authors:
lemma neg-mod-sign : $(a::\text{JavaInt}) < 0 \wedge b \neq 0 \implies a \text{ mod } b \leq 0$
lemma pos-mod-sign : $0 \leq (a::\text{JavaInt}) \wedge b \neq 0 \wedge b \neq \text{MinInt} \implies 0 \leq a \text{ mod } b$
- (4) **lemma** JavaInt-mod-less : $b \neq 0 \wedge b \neq \text{MinInt} \implies \text{abs } ((a::\text{JavaInt}) \text{ mod } b) < \text{abs } b$
 It is not clear whether the “magnitude of the result” refers to the mathematical absolute value or to the Java method `Math.abs`. We decided to use the function `abs` on `JavaInt` which allows us to stay in the abstract model. This has the drawback that the lemma cannot be used for $b = \text{MinInt}$.
- (5) See the discussion for `div` above.
- (6) **lemma** div-mod-example1 : $(5::\text{JavaInt}) \text{ mod } 3 = 2$ etc.

Again, it is not made explicit in the JLS what happens if the dividend equals 0.

Java is not the only language whose definitions of `div` and `mod` do not resemble the mathematical definitions. The languages Fortran, Pascal and Ada define division in the same way as Java, and Fortran’s `MOD` and Ada’s `REM` operators are modelled in the same way as Java’s `%` operator. Goldberg [10, p. H-12] regrets this disagreement among programming languages and suggests the mathematical definition, some of whose advantages he points out.

5 Formalization With Bitstring Representation

5.1 Shift Operators

The shift operators are not properly described in the JLS (§15.19) either. It is especially unclear what happens if the right-hand-side operand of the shift operators is negative. Due to the space limitations of this paper, we have to refrain from presenting the full formalization of the shift operators here.

5.2 Relational Operators

As the relational operators (described in JLS §§15.20, 15.21) do not offer many surprises, we abstain from presenting their formalization here.

5.3 Integer Bitwise Operators $\&$, $\wedge$, and $\mid$

This section formalizes the bitwise AND, OR, and exclusive OR operators.

Java Language Specification [12], §15.22, 15.22.1

“The bitwise operators [...] include the AND operator $\&$, exclusive OR operator $\wedge$, and inclusive OR operator $\mid$.⁽¹⁾ [...] Each operator is commutative if the operand expressions have no side effects. Each operator is associative.⁽²⁾ [...] For $\&$, the result value is the bitwise AND of the operand values. For $\wedge$, the result value is the bitwise exclusive OR of the operand values. For $\mid$, the result value is the bitwise inclusive OR of the operand values. For example, the result of the expression $0\text{xff}00 \& 0\text{x}f0f0$ is $0\text{x}f000$. The result of $0\text{xff}00 \wedge 0\text{x}f0f0$ is $0\text{x}0ff0$. The result of $0\text{xff}00 \mid 0\text{x}f0f0$ is $0\text{xffff}0$.⁽³⁾”

- (1) These bitwise operators are formalized as follows:

constdefs

$\text{JavaInt-bitand} :: [\text{JavaInt}, \text{JavaInt}] \Rightarrow \text{JavaInt}$
 $x \& y \equiv \text{number-of}(\text{zip-bin}(\text{op} \& :: [\text{bool}, \text{bool}] \Rightarrow \text{bool})$
 $\quad (\text{bin-of } x) (\text{bin-of } y))$

where bin-of transforms a **JavaInt** into its bitstring representation, zip-bin merges two bitstrings into one by applying a function (which is passed as the first argument) to each bit pair in turn, and number-of turns the resulting bitstring back into a **JavaInt**. The other two bit operators are defined accordingly.

- (2) The commutativity and associativity of the three operators is proven by six lemmas, of which we present two here:

lemma bitand-commute: $a \& b = b \& a$

lemma bitand-assoc: $(a \& b) \& c = a \& (b \& c)$

- (3) We verify the results of the examples by proving the three lemmas

lemma bitand-example : $65280 \& 61680 = 61440$

lemma bitxor-example : $65280 \wedge 61680 = 4080$

lemma bitor-example : $65280 \mid 61680 = 65520$

In these lemmas we transformed the hexadecimal values into decimal values because Isabelle is currently not able to read hex values.

5.4 Further Features of the Model

The model of Java integers presented above forms a ring. This could easily be proved by using Isabelle/HOL’s Ring theory which only requires standard algebraic properties like associativity, commutativity and distributivity to be proven. The Ring theory makes dozens of ring theorems available for use in proofs. Our model also forms a linear ordering. To achieve this property, reflexivity, transitivity, antisymmetry and the fact that the $\leq$ operator imposes a total ordering had to be proven. This allows us to make use of Isabelle/HOL’s linorder theory. We get a two’s-complement representation by redefining (using our standard wrapper) the conversion function number-of-def

which is already provided for `int`. This representation is used for those operators that are defined bitwise. Altogether, the existing Isabelle theories make it relatively easy to achieve standard number-theoretic properties for types that are defined as a subset of the Isabelle/HOL integers.

5.5 Empirical Data: The Size of our Specification and Proofs

The formalization presented in the preceding sections consists of five theory files, the size of which is as follows:

Filename	Lines	Filename	Lines
JavaIntegersDef.thy	260	JavaIntegersAdd.thy	240
JavaIntegersTimes.thy	200	JavaIntegersRing.thy	500
JavaIntegersDiv.thy	1800	JavaIntegersBit.thy	350

It took about one week to specify the definitions and lemmas presented here and about eight to ten weeks to prove them, but the proof work was mainly performed by one of the authors (NR) who at the same time learned to use Isabelle, so an expert would be able to achieve these results much faster.

6 Formalizing the Exceptional Behavior Java Integers

The Java Language Specification introduces the concept of *exception* in expressions and statements of the language:

Java Language Specification [12], §11.3, §11.3.1

“The control transfer that occurs when an exception is thrown causes abrupt completion of expressions (§15.6) and statements (§14.1) until a catch clause is encountered that can handle the exception [...]
when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated.”

Thus, exceptions have two aspects in Java:

- they change the control flow of a program,
- they are a particular kind of side-effect (i.e. an exception object is created), and they prevent program parts from having side-effects.

While we deliberately neglect the latter aspect in our model (which can be handled in a Hoare Calculus on full Java, for example, when integrating our expression language into the statement language), we have to cope with the former aspect since it turns out to have dramatic consequences for the rules over Java expressions (these effects have not been made precise in the JLS).

So far, our normal behavior model is a completely denotational model; each expression is assigned a value by our semantic definitions. We maintain this

denotational view, with the consequence that we have to introduce *exceptional values* that are assigned to expressions that “may [not] appear to have been evaluated”. In the language fragment we are considering, only one kind of exception may occur:

Java Language Specification [12], §4.2.2

“The only numeric operators that can throw an exception (§11) are the integer divide operator `/` (§15.17.2) and the integer remainder operator `%` (§15.17.3), which throw an `ArithmeticException` if the right-hand operand is zero.”

In order to achieve a clean separation of concerns, we apply the technique developed in [7]. Conceptually, a theory morphism is used to convert a normal behavior model into a model enriched by exceptional behavior. Technically, the effect is achieved by redefining all operators such as `+`, `-`, `*` etc. using “semantical wrapper functions” and the normal behavior definitions given in the previous chapters. Two types of theory morphisms can be distinguished: One for a *one-exception world*, the other for a *multiple-exception world*. While the former is fully adequate for the arithmetic language fragment we are discussing throughout this paper, the latter is the basis for future extensions by e.g. array access constructs which may raise *out-of-bounds exceptions*. In the following, we therefore present the former in more detail and only outline the latter.

6.1 The One-Exception Theory Morphism

We begin with the introduction of a type constructor that disjointly adds to a type α a failure element such as $\perp$ (see e.g. [27], where the following construction is also called “lifting”). We declare a type class *bot* for all types containing a failure element $\perp$ and define as *semantical combinator*, i.e. as “wrapper function” of this theory morphism, the combinator *strictify* that turns a function into its *strict extension* wrt. the failure elements:

$$\begin{aligned} \text{strictify} &:: ((\alpha::\text{bot}) \Rightarrow (\beta::\text{bot})) \Rightarrow \alpha \Rightarrow \beta \\ \text{strictify } f \ x &\equiv \text{if } x=\perp \text{ then } \perp \text{ else } f \ x \end{aligned}$$

Moreover, we introduce the definedness predicate $\text{DEF} :: \alpha::\text{bot} \Rightarrow \text{bool}$ by $\text{DEF } x \equiv (x \neq \perp)$. Now we introduce a concrete type constructor that lifts any type α into the type class *bot*:

$$\mathbf{datatype} \quad \text{up}(\alpha) = \lfloor _ \rfloor \ \alpha \mid \perp$$

In the sequel, we write $t_\perp$ instead of $\text{up}(t)$. We define the inverse to the constructor $\lfloor _ \rfloor$ as $\lceil _ \rceil$. Based on this infrastructure, we can now define the type **JAVAINT** that includes a failure element:

$$\mathbf{types} \ \mathbf{JAVAINT} = \mathbf{Javalnt}_\perp$$

Furthermore, we can now define the operations on this enriched type; e.g. we convert the **Javalnt** unary minus operator into the related **JAVAINT** operator:

$$\begin{aligned} \mathbf{constdefs} \\ \text{uminus} &:: \mathbf{JAVAINT} \Rightarrow \mathbf{JAVAINT} \end{aligned}$$

$$\text{uminus} \quad \equiv \quad \text{strictify}(\lfloor _ \rfloor \circ \text{uminus} \circ \lceil _ \rceil)$$

As a canonical example for binary functions, we define the binary addition operator by (note that Isabelle supports overloading):

$$\begin{aligned} \text{op } +: [\text{JAVAINT}, \text{JAVAINT}] &\Rightarrow \text{JAVAINT} \\ \text{op } + &\equiv \text{strictify}(\lambda X. \text{strictify}(\lambda Y. \lfloor [X] \rfloor + \lceil [Y] \rceil)) \end{aligned}$$

All binary arithmetic operators that are strict extensions like $-$ or $*$ are constructed analogously; the equality and the logical operators like the strict logical AND $\&$ follow this scheme as well. For the division and modulo operators $/$ and $\%$, we add case distinctions whether the divisor is zero (yielding $\perp$). Java's non-strict logical AND $\&\&$ is defined in our framework by explicit case distinctions for $\perp$.

This adds new rules like $X + \perp = \perp$ and $\perp + X = \perp$. But what happens with the properties established for the normal behavior semantics? They can also be lifted, and this process can even be automated (see [7] for details). Thus, the commutativity and associativity laws for normal behavior, e.g. $(X :: \text{Javalnt}) + Y = Y + X$, can be lifted to $(X :: \text{JAVAINT}) + Y = Y + X$ by generic proof procedure establishing the case distinctions for failures. However, this works smoothly only if all variables occur on both sides of the equation; variables only occurring on one side have to be restricted to be defined. Consequently, the lifted version of the division theorem looks as follows:

$$\llbracket \text{DEF } Y; Y \neq 0 \rrbracket \implies ((X :: \text{JAVAINT}) / Y) * Y + (X \% Y) = X$$

6.2 The Multiple-Exception Theory Morphism

The picture changes a little if the semantics of more general expressions are to be modelled, including e.g. array access which can possibly lead to out-of-bounds exceptions. Such a change of the model can be achieved by exchanging the theory morphism, leaving the normal behavior model unchanged.

It suffices to present the differences to the previous theory morphism here. Instead of the class `bot` we introduce the class `exn` requiring a family of undefined values $\perp_e$. The according type constructor is defined as:

$$\text{datatype } \text{up}(\alpha) = \lfloor _ \rfloor \alpha \mid \perp \text{ exception}$$

and analogously to $\lfloor _ \rfloor$ we define $\text{exn-of}(\perp_e) = e$ as the inverse of the constructor $\perp$; exn-of is defined by an arbitrary but fixed HOL-value *arbitrary* for $\text{exn-of}(\lfloor _ \rfloor) = \text{arbitrary}$. Definedness is $\text{DEF}(x) = (\forall e. x \neq \perp_e)$.

The definition of operators is analogous to the previous section for the canonical cases; and the resulting lifting as well. Note, however, that the lifting of the commutativity laws fails and has to be restricted to the following:

$$\begin{aligned} \llbracket \text{DEF } X = \text{DEF } Y \wedge \text{exn-of } X = \text{exn-of } Y \rrbracket \\ \implies (X :: \text{JAVAINT}) + Y = Y + X \end{aligned}$$

These restrictions caused by the lifting reflect the fact that commutativity does not hold in a multi-exception world; if the left expression does not raise the same exception as the right, the expression order cannot be changed.

Hence, our proposed technique to use a theory morphism not only leads to a clear separation of concerns in the semantic description of Java, but also leads to the systematic introduction of the side-conditions of arithmetic laws in Java that are easily overlooked.

7 Conclusions and Future Work

In this paper we presented a formalization of Java’s two’s-complement integral types in Isabelle/HOL. Our formalization includes both normal and exceptional behavior. Such a formalization is a necessary prerequisite for the verification of efficient arithmetic Java programs such as encryption algorithms, in particular in tools like Jive [20] that generate verification conditions over arithmetic formulae from such programs.

Our formalization of the normal behavior is based on a direct analysis of the Java Language Specification [12] and led to the discovery of several underspecifications and ambiguities (see 3.1 (4), 4.1, 4.2, 5.1). These underspecifications are highly undesirable since even compliant Java compilers may interpret the same program differently, leading to unportable code. In the future, we strongly suggest to supplement informal language definitions by machine-checked specifications like the one we present in this paper as a part of the normative basis of a programming language.

We applied the technique of mechanized theory morphisms (developed in [7]) to our Java arithmetic model in order to achieve a clear separation of concerns between normal and exceptional behavior. Moreover, we showed that the concrete exceptional model can be exchanged — while controlling the exact side-conditions that are imposed by a concrete exceptional model. For the future, this leaves the option to use a lifting to the *exception state monad* [18] mapping the type `JAVAINT` to state $\Rightarrow(\text{Javalnt}_{\perp}, \text{state})$ in order to give semantics to expressions with side-effects like `i++ + i`.

Of course, more rules can be added to our theory in order to allow effective automatic computing of large (ground) expressions — this has not been in the focus of our interest so far. With respect to proof automation in `Javalnt`, it is an interesting question whether arithmetic decision procedures of most recent Isabelle versions (based on Cooper’s algorithm for Presburger Arithmetic) can be used to decide analogous formulas based on machine arithmetic. While an *adoption* of these procedures to Java arithmetic seems impossible (this would require cancellation rules such as $(a \leq b) = (k \times a \leq k \times b)$ for nonnegative k which do not hold in Java), it is possible to retranslate `Javalnt` formulas to standard integer formulas; remainder sub-expressions can be replaced via $P(a \bmod b) = \exists m. 0 \leq m < a \wedge (a - m) \mid b \wedge P(m)$, such that finally a Presburger formula results. Since a translation leads to an exponential blow-

up in the number of quantifiers (a critical feature for Cooper’s algorithm), it remains to be investigated to what extent this approach is feasible in practice.

References

- [1] M. D. Aagaard and C.-J. H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *Int. Conf. on Computer Aided Design*. IEEE Computer Society, 1995.
- [2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. CUP, 1996.
- [3] M. Balser et al. Formal system development with KIV. In *Fundamental Approaches to Software Engineering*, LNCS 1783, 2000.
- [4] B. Beckert and S. Schlager. Integer arithmetic in the specification and verification of Java programs. In *FM-TOOLS*, 2002.
- [5] J. v. d. Berg and B. Jacobs. The LOOP compiler for Java and JML. In *TACAS01*, LNCS 2031, 2001.
- [6] Dider Bondyfalat. Long integer division in Coq (algorithm divide and conquer). <http://www-sop.inria.fr/lemme/Didier.Bondyfalat/DIV/>.
- [7] A. D. Brucker and B. Wolff. Using theory morphisms for implementing formal methods tools. In *Types for Proof and Programs*, LNCS, 2003.
- [8] V. A. Carreño and P. S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *Higher Order Logic Theorem Proving and its Applications*, 1995.
- [9] A. C. J. Fox. An algebraic framework for modelling and verifying microprocessors using HOL. TR 512, University of Cambridge, 2001.
- [10] D. Goldberg. Computer arithmetic. In *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.
- [11] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
- [12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The JavaTM Language Specification – Second Edition*. Addison-Wesley, 2000.
- [13] J. Harrison. A machine-checked theory of floating point arithmetic. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS 1690, 1999.
- [14] Marc Herbstritt. e-mail communication, May 2003. Chair of Computer Architecture, Uni Freiburg.
- [15] S. Höreth and R. Drechsler. Formal verification of word-level specifications. In *IEEE Design, Automation and Test in Europe (DATE)*, 1999.
- [16] Bart Jacobs. Java’s integral types in PVS. *Submitted*, 2003.

- [17] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*. Kluwer, 1999.
- [18] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL'95: Principles of Programming Languages*, 1995.
- [19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.
- [20] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In *TACAS00*, LNCS 276, 2000.
- [21] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828. Springer, 1994.
- [22] C. Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, TU München, 1998.
- [23] D. Sannella and A. Tarlecki. Algebraic methods for specification and formal development of programs. *ACM Computing Surveys*, 31(3es), 1999.
- [24] C. Scholl, B. Becker, and T. Weis. On WLCDs and the complexity of word-level decision diagrams — a lower bound for division. *Formal Methods in System Design*, 20(3), 2002.
- [25] Sun Microsystems, Inc. *Java CardTM 2.1.1 Specifications – Release Notes*, 2000.
- [26] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.3*, 2002.
- [27] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.