JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics

Patrice Chalin

Computer Science Department, Concordia University
www.cs.concordia.ca/~faculty/chalin

1 Introduction

The Java Modeling Language, JML, is a notation for specifying and describing the detailed design and implementation of Java modules. It is a model-based specification language offering, in particular, method specification by pre- and post-condition, and class invariants to document required module behavior. It has recently been noted that the JML semantics of expressions over numeric types do not correspond to user expectations. As a result, an unusually high number of published JML specifications are invalid or inconsistent, including cases from the security critical area of smart card applications [Chalin03a]. In this extended abstract we briefly describe JML's ancestry and language design principles (Section 2). This will help to explain the origin of the "semantic gap" between user expectations and the current meaning given to JML numeric expressions. With the objective of better matching user expectations, we introduce JMLa, a variant of JML supporting *primitive arbitrary precision* numeric types as well as implicit promotion to these types (Section 3). This is done in a manner that is consistent with JML's language design goals and objectives [Chalin03a]. A preliminary formal semantics of JMLa expressions is given (Section 4) as well as an example of its application. Related and future work are described (Sections 5 and 6, respectively).

2 JML

2.1 Ancestry and language design principles

JML is a Behavioral Interface Specification Language (BISL). By definition, a BISL is tightly coupled to a particular programming language since its purpose is to allow developers to specify modules written in that programming language. A behavioral interface specification is a description of a module consisting of two main parts [Wing87]:

- an *interface*, that captures language specific elements that are exported by the module, such as field and method signatures;
- a *behavior*—as well as other properties and constraints—of the elements described in the interface.

Prior to JML, the main BISLs were members of the Larch family of languages of which two notable members are Larch/C++ [Leavens99] and LCL, the Larch/C interface specification language [GH93]. A key characteristic of Larch is its two-tiered approach. The *shared* tier contains specifications written in the *Larch Shared Language* (LSL). These shared tier specifications, called traits, define multisorted first-order theories. The *interface* tier contains specifications written in a Larch interface language. Each interface language is specialized for use with a particular programming language, but all interface languages make use of LSL to express module behavior [GH93]. In a departure from the Larch tradition, Leavens *et. al.* have defined JML as a single-tier BISL [LBR02]. Experience with Larch/C++ lead to the opinion that having to learn two—somewhat disparate—languages (C++ and LSL) in order to be able to read and write specifications, was too big a hurdle to overcome for most developers. The design intent has been to make JML a superset of (a significant subset of) Java. A key language design principle of JML has been to preserve the semantics of Java to the extent possible: that is, if a phrase is valid in Java and JML, then it should have the same meaning in both languages. Adherence to this principle greatly reduces the burden required to learn, understand and use JML.

```
/*@ public normal_behavior
  @   requires y >= 0;
  @   ensures Math.abs(\result) <= y
  @        && \result * \result <= y
  @        && y < (Math.abs(\result) + 1)
  @                * (Math.abs(\result) + 1);
  @*/
public static int isqrt(int y)
```

Figure 1. JML specification of `isqrt(int)`

```
/*@ spec_public */
    private short intPart, decPart;

/*@ normal_behavior
  @  modifiable intPart, decPart;
  @  ensures    intPart == -\old(intPart) &&
  @             decPart == -\old(decPart) ...;
  @*/
public Decimal oppose()
```

Figure 2. `Decimal` class specification excerpt

As is often the case for language design principles, its benefits come at a cost. Java was designed as a programming language, not a specification language. Although JML builds upon Java by adding language constructs for the purpose of expressing specifications, it remains that core Java phrase sets, like expressions, are (for the most part) shared by both languages. This renders expression semantics more complex than, for example, in Larch. Furthermore, as we shall see in the following section, developers are in a different mindset when reading or writing specifications, particularly when it comes to reasoning about integer arithmetic.

2.2 A semantic gap, motivating examples

Consider the specification in Figure 1 of an integer square root method, `isqrt`; it was excerpted from the June 2002 edition of the main JML reference document [LBR02]. The specification requires that a caller invoke the method with a nonnegative argument *y*, and in return, the method ensures that it will yield a resulting value, *r*, such that: $|r| \leq y \ \wedge \ r^2 \leq y < (|r|+1)^2$. The current definition of JML states that the expressions in the `requires` and `ensures` clauses of Figure 1 are to be interpreted using the semantics of Java. As a consequence (and a simple Java prototype will justify this claim), a valid implementation of `isqrt` would be permitted to return `Integer.MIN_VALUE` when *y* is 0. This unexpected situation arises because Java integral types have a fixed precision and because operators over these types obey rules similar to modular arithmetic—thus, for example `Integer.MIN_VALUE = Integer.MAX_VALUE + 1`. As another example consider the specification, given in Figure 2, that has been excerpt from a paper on the formal verification of an electronic purse applet [BvdBJ02]. In this case, the specification is inconsistent (it is unsatisfiable when, say, `intPart` is `Short.MIN_VALUE`).

What has gone wrong? These JML specifications (and others) demonstrate that specifiers most often ignore the finiteness of numeric types. Stated positively, specifiers generally *think* in terms of arbitrary precision arithmetic when they read and write specifications. A survey is given in [Chalin03a] of invalid and inconsistent JML specifications caused by this problem (including the two just described). Hence, there is a semantic gap between user expectations and the current language design and semantics of JML numeric types. Our attempts at mending this gap without changing the language definition of JML have been unfruitful [Chalin03b]. In particular, use of the `JMLInfiniteInteger` model class yields specifications that are overly verbose and hence difficult to read and understand. With the objective of closing the semantic gap, we propose in the next section a variant of JML named JMLa.

3 JMLa

Most JML specifiers think in terms of arbitrary precision arithmetic, yet the semantics of expressions in JML is such that fixed precision arithmetic is the *default* interpretation. We have defined a variant of JML called JMLa that offers support for the arbitrary precision numeric types `\bigint` and `\real` [Chalin03a]. The semantics of JMLa ensure that numeric operations that can cause arithmetic overflow are performed over *arbitrary precision types* by default. We will call the operators that can result in overflow *unsafe* operators; they are: unary `-`, binary `+`, `-`, * and `/`. Generally, in JMLa, unsafe operators will promote their integral operands to `\bigint`. As an exception to this rule, we preserve the semantics of Java for constant expressions whose evaluation would not result in an overflow. Examples are given in Figure 3. Notice how `-5` has type `int` whereas `-Integer.MIN_VALUE` has type `\bigint` because evaluation of the latter constant expression in Java would result in an overflow.

JMLa Expression	Equivalent JMLa Expression (all conversions made explicit)	Result Type	Same semantics as in Java?
`+i`	`+i`	`int`	YES, since unary + is safe.
`-i`	`-(\bigint)i`	`\bigint`	No, unary – is unsafe.
`-5`	`-5`	`int`	YES, since –5 is an `int` value.
`-Integer.MIN_VALUE`	`-(\bigint)Integer.MIN_VALUE`	`\bigint`	No, since the constant expression value is not in the range of `int`.
`i + j`	`(\bigint)i + (\bigint)j`	`\bigint`	No, since binary + is unsafe.
`Integer.MIN_VALUE - 1`	`(\bigint)Integer.MIN_VALUE - (\bigint)1`	`\bigint`	No, since the expression value is not in the range of `int`.
`2 + k`	`(\bigint)2 + k`	`\bigint`	No, since `\bigint` is not a Java type.
`2 + (int)k`	`(int)2 + (int)k`	`int`	Almost.
`i * f`	`(\real)i * (\real)f`	`\real`	No. Note that a floating-point type causes promotion to `\real`.
`3 * 5 – Short.MAX_VALUE`	`3 * 5 – (int)Short.MAX_VALUE`	`int`	YES (const. expr. value is an `int`).
`d / 0.5`	`(\real)d / (\real)0.5`	`\real`	No.
`(\bigint)d / 2`	`(\bigint)d / (\bigint)2`	`\bigint`	No.

Figure 3. Sample JMLa expressions (assume `int i,j; \bigint k; float f; double d`)

Narrowing casts of primitive numeric types are also given a special semantics in JMLa. For example, if `i` is an `int`, then `(byte)i` is interpreted as `narrowToByte(i)` where this method is defined as:

```
static byte narrowToByte(int i) {
  if((byte)i != i) {
    throw new ArithmeticException("Value out of range for type byte: " + i);
  }
  return (byte)i;  // no loss of precision.
}
```

On occasion, we need to write specifications in which some of the arithmetic operators are to be interpreted with their Java semantics. This occurs, for example, when documentation existing API classes (e.g. `java.lang.*`). For this purpose, we define the semantics of the JMLa expression `\nowarn(`*`E`*`)` to be the same as the Java semantics of *`E`*. Within `\nowarn(`*`E`*`)` one can make use of `\warn()` to re-enable JMLa semantics. As a variant, `\nowarn_op(`*`E`*`)` allows us to specify that only the outer-most operator in *`E`* should be interpreted with Java semantics. Similarly, `\warn_op(`*`E`*`)` only re-enables JMLa semantics for the outer-most operator in *`E`*. Hence, the JMLa expression `\nowarn_op((byte)`*`E`*`)` has the effect of interpreting *`E`* with JMLa semantics and then, like in Java, silently casting the result to `byte`. As another example we note that `\nowarn(Integer.MAX_VALUE + 1) == Integer.MIN_VALUE` would evaluate to `true`.

For the sake of conciseness in this extended abstract, we simply enumerate some of the key advantages of JMLa over JML—justifications of these claims as well as further details are provided in [Chalin03a]:

- JMLa semantics more closely match user expectations. We demonstrate in [Chalin03a] how all of the invalid or inconsistent JML specifications given in that report recover their validity and consistency when interpreted under JMLa with little or no changes to the specifications.
- JMLa can be used to write simpler, and clearer specifications (as compared, e.g., to use of `JMLInfiniteInteger`).
- The meaning of JMLa specifications can be independent of the particular choice of numeric types of fields and variables (as it should be since, e.g., method specifications are meant to express *essential* method behavior which often is independent of field and variable types).[1]
- ESC/Java [Flanagan+02] will be able to detect more errors under JMLa semantics than it currently can for JML.
- Verification proofs will be greatly simplified as we recover the familiar laws of arithmetic when operating over `\bigint` and `\real` (e.g. associativity, commutativity and closure of operators).

[1] This is not the case in Java and JML; e.g. "`\result == `*E*" may be unsatisfiable if `\result` is declared to be of type `short`.

$e \in$ EXPR ::= $c_\tau \mid \iota \mid op(e_1, \ldots, e_k) \mid (\tau)\, e \mid$ **\old**$(e) \mid e.\iota \mid e.\iota(e_1, \ldots, e_k) \mid (q\ tp\ \iota\ ;\ e)$
| **\nowarn_op**$(e) \mid$ **\warn_op**$(e) \mid$ **\nowarn**$(e) \mid$ **\warn**$(e) \mid \ldots$

$\tau \in$ TYPENM ::= $\iota \mid \ldots$

$op \in$ OPNM

$q \in$ QUANTNM ::= **\forall** | **\exists**

Figure 4. Abstract syntax of JMLa expressions

These points are particularly important as we witness the increased use of JML, especially in security critical areas like smart cards. Of course, these benefits come at the cost of a slightly more complex semantics and an increased departure from Java semantics. We believe though, that the benefits of JMLa outweigh its disadvantages.

4 JMLa Semantics

This section offers a glimpse at work in progress on the formalization of JMLa semantics by means of an embedding into the language of PVS [PVS]. PVS is a specification language integrated with a theorem prover. The specification language of PVS is based on classical, typed higher-order logic. For simplicity we can assume that each JMLa specification is translated into a PVS theory. JMLa expressions are translated into PVS expressions. In this extended abstract we focus on the semantics of expressions over primitive numeric types while ignoring the very important issue of abnormal termination in expressions.

4.1 Abstract syntax and semantic objects

The semantics of JMLa expressions is defined by means of an "inference system" in a style referred to as natural semantics [Winskel93]. The inference rules allow us to establish the validity of *elaboration predicates* of the form

$$\rho \vdash a \xrightarrow{A} x$$

where A is generally the name of an abstract syntax phrase class. Such a predicate asserts that the syntactic object a corresponds to the semantic object x under the context ρ; we will also say "a elaborates to x under ρ." For the cases covered here, the context will be an environment containing the declarations under which elaboration is to be performed. Furthermore, a will be a JMLa expression and x a PVS expression qualified with its type.

The abstract syntax for expressions relevant to our presentation is given in Figure 4. The cases defined are:

- An integral literal constant of type $\tau \in$ {`int`, `long`}.
- An identifier representing a logical variable (including `\result` and method parameters)[2].
- An operator applied to one or more arguments. Operators include those of Java (e.g. +, `-`, *) and JML (e.g. `==>`, `<==>`).
- A type cast expression.
- A pre-state expression which is of the form **\old**(e).
- A field access expression.
- A method invocation expression. (Recall that methods in JML expressions must be "pure" [LBR02].)
- A quantified expression.
- Warn/nowarn expressions.

JMLa expressions are translated into the "semantic objects" of PVS expressions, whose *annotated* abstract syntax is

$\varepsilon \in$ PVSEXPR ::= $c : \tau \mid op(\varepsilon_1, \ldots, \varepsilon_k) : \tau$

[2] Since our focus in this extended abstract is on the particularities of JMLa semantics of expressions over numeric types, we shall make the simplifying assumption that all class and instance members are expressed in the form $e.\iota$ so as to be distinguishable from the occurrence of a logical variable.

```
int: THEORY
 BEGIN
  ...
  twoPn: posint = 4294967296
  max  : nat    = 2147483647
  min  : negint = -2147483648

  int: TYPE+ = {i: integer | min <= i AND i <= max} CONTAINING 0;

  % narrowing primitive conversion to int
  narrow(i: integer): int =
      LET b:nat = mod(i, twoPn) IN
        IF b <= max THEN b ELSE b - twoPn ENDIF

  %-------------------------------------------------
  neg(i:int): int = narrow(-i)
  add(i,j:int): int = narrow(i + j)
  sub(i,j:int): int = narrow(i - j)
  mul(i,j:int): int = narrow(i * j)
  div(i:int, j:{j:int|j /= 0}): int = narrow(div.div(i,j))
  ...
  %-------------------------------------------------
  bit_neg(i:int): int = -i - 1
  ...
 END int
```

Figure 5. PVS theory for `int`

Each PVS expression is annotated with its type. This allows us to ensure that, in particular, overloaded operators can be disambiguated. Elaboration of expressions is done in the context of an environment, $\rho \in \text{ENV}$ that can be thought of as a mapping from identifiers[3] into their attributes. The updated environment denoted by $\rho \oplus \{ \iota \mapsto \alpha \}$ is the same as ρ except that it maps ι to α. Note that in the initial JMLa environment ρ_o, `\safeMath` is `true`.

4.2 Primitive numeric types in PVS

Before presenting the elaboration rules, let us explain how JMLa primitive numeric types are modeled in PVS. The JMLa arbitrary precision types `\bigint` and `\real` are modeled by the standard PVS types `integer` and `real`. For convenience, we have also defined a synonym for `integer` named `bigint`. We have created simple theories, all of the same form, for each of the bounded precision integral types. As an example, an excerpt of the theory for `int` is given in Figure 5. Notice how the `int` type is simply defined as the subtype of `integer` that contains values in the range `min` to `max` inclusive. A key function in this theory is `narrow` which effectively defines narrowing primitive conversion to `int`. All arithmetic operators are defined using their `integer` counterparts followed by an application of `narrow`. Thus, addition of `int`'s is defined as the addition of their values interpreted as `integer`'s followed by a narrowing of the result to `int`: i.e. `add(i,j) = narrow((i:int + j:int):integer):int`.

4.3 Elaboration rules

A simplified version of the elaboration rules for JMLa expressions is given in Figure 8 on page 9. For each syntactic case of EXPR we have combined the type and expression elaborations into a single rule. Aside from the "\" in JMLa type names `\bigint` and `\real`, JMLa and PVS type names coincide, hence we will make no distinction between them, using the same type name τ in both the abstract syntax and PVS expressions. This given simplified semantics does not cover rules for constant expressions, and (as was mentioned earlier) it ignores issues of abnormal termination.

The rules for literals [Literal] and logical variables [Logical Var] illustrate that they are translated almost literally. Thus `3` and `5L` elaborate to `3:int` and `5:long` respectively. Similarly, provided that a logical variable has been declared in scope, it elaborates to the same name qualified with its declared type[4].

[3] Including special identifiers like `\state`, denoting the default PVS state variable context, and `\safeMath`, `\result`, etc.
[4] Of course `\result` would have to be mapped to a special PVS name.

Operator(s) *op*	**Argument Type(s)** $ArgType_{op}$	**Argument Conversion** $conv_{op}$	**Result Type** $resultType_{op}$	**PVS Function(s)** ϕ_{op}
`+` (unary)	Numeric[5]	*unp* to τ	τ	`id`: τ → τ
`-` (unary)	Numeric	*unp* to τ	τ	`τ.neg`
`~`	Integral	*unp* to τ	τ	`τ.bit_neg`
`!`	`boolean`	none	`boolean`	`NOT`
`*, /, %`	Numeric, Numeric	*bnp* to τ	τ	`τ.mul`, `τ.div`, `τ.rem`
`+, -`	Numeric, Numeric	*bnp* to τ	τ	`τ.add`, `τ.sub`
`<<, >>, >>>`	Integral, Integral	arg1:*unp* to τ arg2: *unp*	τ	`τ.lshift`, `τ.rshift`, `τ.rshift_u`
`<, <=, >, >=`	Numeric, Numeric	*bnp*	`boolean`	`<, <=, >, >=`
`==, !=`	Numeric, Numeric	*bnp*	`boolean`	`==, /=`
	`boolean`, `boolean`	none	`boolean`	`==, /=`
`&, ^, |`	`boolean`, `boolean`	none	`boolean`	`AND`, `XOR`, `OR`
	Integral, Integral	*bnp* to τ	τ	`τ.bit_and`, `τ.bit_xor`, `τ.bit_or`
`&&, ||`	`boolean`, `boolean`	none	`boolean`	`AND`, `OR`
`==>`	`boolean`, `boolean`	none	`boolean`	`==>`
`<==>, <=!=>`	`boolean`, `boolean`	none	`boolean`	`==, /=`

Table 1. Semantics of selected JMLa operators

There are two elaboration rules for expressions involving operators. [Op_{safe}] applies to all unsafe operators (these are listed in Section 3) while in "safe math" mode. JMLa specifications are interpreted in safe math mode by default. This default can be changed by the use of operators `\nowarn`, `\warn`, etc., as can be seen in the Warn/Nowarn rules. The [Op] rule applies to operators that are not unsafe, or to any operator while in unsafe math mode. To apply either of these rules one must make use of the information provided in Table 1. Let *op* be an operator that appears in column 1 of the table, then the remaining columns define: the required argument type(s) ($ArgType_{op}$), the kind of argument conversion to be applied ($conv_{op}$), the resulting type of the expression ($resultType_{op}$) and finally, the PVS function corresponding to the operator *op* (ϕ_{op}). The argument conversions that can be applied correspond to unary numeric promotion or binary numeric promotion. These promotion functions come in two variants (see Figure 6): those corresponding to implicit promotion to arbitrary precision types (*JML*.*) and those that imitate the standard Java promotion rules (*Java*.*).

Type casts are processed in two cases depending on the nature of the cast: i.e. whether it corresponds to a widening [Widen] or narrowing primitive conversion [Narrow]. Type widening requires no special operator in PVS. On the other hand, narrowing to type τ requires the application of the `narrow` function defined in the theory of τ.

The remaining rules are further discussed in [Chalin03b].

[5] Recall that for JMLa, primitive numeric types include `\real` and `\bigint`.

JML.unp(τ) =
  if τ ∈ { `float`, `double`, `\real` }
  then `\real`
  else `\bigint` ***end***
 	
JML.bnp(τ_1, τ_2) =
  if τ_1 or τ_2 is one of { `float`, `double`, `\real` }
  then `\real`
  else `\bigint` ***end***
	Java.unp(τ) = ***if*** τ ∈ { `byte`, `short`, `char` }
	  then `int` ***else*** τ ***end***
	
	Java.bnp(τ_1, τ_2) = ***if*** `\real` ∈ {τ_1, τ_2} ***then*** `\real`
	  else-if `\bigint` ∈ {τ_1, τ_2} ***then*** `\bigint`
	  else-if `double` ∈ {τ_1, τ_2} ***then*** `double`
	  else-if `float` ∈ {τ_1, τ_2} ***then*** `float`
	  else-if `long` ∈ {τ_1, τ_2} ***then*** `long`
	  else `int` ***end***

Figure 6. JMLa type conversion functions

4.4 Example

As an example of the application of the elaboration rules we will use a slightly modified version of the specification of `isqrt` given in Figure 1 in which we replace the occurrences of `Math` by `JMLMath`. `JMLMath` is a model class that contains methods like `Math` but that are defined over JMLa arbitrary precision types [Chalin03a]. The result is shown in Figure 7. Notice how the PVS expressions closely resemble their JMLa counterparts. As a partial indication of the suitability of the semantic translation, we have been successful in proving the consistency of the `isqrt` specification.

5 Related work

Several computer languages and tools provide basic language support for arbitrary precision integers including: specification languages, such as B, OBJ, VDM, and Z [Bowen03]; BISLs such as Larch, and Extended ML (EML); Functional languages like ML, Haskell, and Lisp; proof tools (e.g. PVS) and symbolic mathematics systems such as Mathematica and Maple. Basic support for real numbers is most common in general design specification languages and proof tools and less common in other languages. Symbolic mathematics packages often provide arbitrary precision rational numbers.

The LOOP tool, being developed at the University of Nijmegen, translates JML specifications and Java code into the language of PVS [vdBJ01, JP01]. The current LOOP embedding defines a semantics that more closely resembles the proposed semantics for JMLa than that of JML. Unfortunately, the embedding also does not currently support expressions involving *both* arbitrary precision and bounded precision types.

The work described in this extended abstract has been inspired by our previous work on the semantics of LCL, which we also defined by means of an embedding into a logic [CGR96]. Our work on LCL semantics has resumed and we are currently extending and generalizing it [Chalin02]. In another report, we provide an exploration of language design alternatives for the JML support of arbitrary precision numeric types. In this same report, we also offer a more rigorous comparison of the formal semantics of JML, JMLa, Larch/C++ and LCL [Chalin03b].

6 Conclusions and future work

We have illustrated a semantic gap between user expectations of the meaning of expressions over numeric types and the current JML language definition. Due to this gap, several published JML specifications are

```
IMPORTING int, ...

isqrt_requires(y: int): bool =  y >= 0;
isqrt_ensures(y, result: int): bool =
          JMLMath.abs(result) <= y AND
          result * result <= y AND
          y < (JMLMath.abs(result) + 1) * (JMLMath.abs(result) + 1);

isqrt_consistent: LEMMA
      FORALL (y:int):
                EXISTS (result:int): isqrt_requires(y) => isqrt_ensures(y,result)
```

Figure 7. PVS definition of `isqrt`

invalid or inconsistent [Chalin03a]—we have presented two such problematic specifications. To better meet user expectations, we have defined a variant of JML called JMLa that has support for primitive arbitrary precision numeric types `\bigint` and `\real`. Furthermore, JMLa defines implicit operand promotion to these arbitrary precision types. Special JMLa operators such as `\nowarn()` allow us to recover Java expression semantics. A preliminary formal semantics of JMLa expressions is given and its application is illustrated by means of a simple example.

Work on the formal semantics of JMLa will be pursued so as to progressively include more language elements. Although PVS has been successfully used to establish proofs of consistency and validity of simple JMLa specifications, larger examples will need to be tackled. We will also continue our analysis of JML as other issues related to bounded vs. unbounded "data types" (such as arrays, sets and sequences) need to be examined. In collaboration with other JML project partners we will be implementing support for `\bigint` and `\real` in ESC/Java and the JML checker. This will allow us, in particular, to support our claim that ESC/Java will be able to detect more errors in JMLa vs. JML specifications. Preliminary implementation of JMLa in the JML checker has already allowed us to detect over two dozen inconsistent or erroneous JML specifications.

7 Acknowledgments

We thank the anonymous referees for their helpful comments on the paper, as well as members of the JML developer community for discussions that have contributed to the improvement of JMLa. Also thanks to Frederic Rioux for his contribution to the implementation of JMLa support in the JML checker.

8 References

[Bowen03] Jonathan Bowen, WWW Virtual Library: Formal Methods, http://www.afm.sbu.ac.uk. February 2003.

[BvdBJ02] C.-B. Breunesse, J. van den Berg, and B. Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In H. Kirchner and C. Ringeissen, editors, *AMAST'2002*, LNCS, pp. 304-318. Springer Verlag, 2002. Decimal class specification is available at www.cs.kun.nl/indexes/~ceesb/decimal/Decimal.java.

[CGR96] Patrice Chalin, Peter Grogono, and T. Radhakrishnan. "Identification of and solutions to shortcomings of LCL, a Larch/C interface specification language". In Marie-Claude Gaudel and James Woodcock, eds, *FME'96: Industrial Benefit and Advances in Formal Methods*, LNCS 1051, pp. 385–404. Formal Methods Europe, Springer, March 1996.

[Chalin02] Patrice Chalin. *Formal Semantics of LCL, Revised.* ENCS-CS Technical Note 2002-001.1, Concordia University, 2002.

[Chalin03a] Patrice Chalin. Improving JML: For a Safer and More Effective Language. In Stefania Gnesi, Keijiro Araki, and Dino Mandrioli (Eds.), *FME 2003, International Symposium of Formal Methods Europe*, Pisa, Italy, Sept. 8-14, 2003, Proceedings (to appear).

[Chalin03b] Patrice Chalin. *Back to Basics: Language Support and Semantics of Basic Infinite Integer Types in JML and Larch*. ENCS-CS TR 2002-003.4, Concordia University, October 2002; latest revision, July 2003.

[Flanagan+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In Cindy Norris and James B. Fenwick, editors, *Proceedings of Conference on Programming Language Design and Implementation (PLDI-02)*, volume 37, 5 of ACM SIGPLAN, pages 234–245, June 17–19 2002.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andr´es Modet, and Jeannette M. Wing.

[JP01] Bart Jacobs and Erik Poll. A Logic for the Java Modeling Language JML. In: H. Hussmann (ed.), *Fundamental Approaches to Software Engineering* (*FASE*), LNCS 2029 pages284-299. Springer-Verlag 2001.

[LBR02] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Computer Science Dept., Iowa State University, TR #98-06t, Dec. 2002.

[Leavens99] Gary T. Leavens. *Larch/C++ Reference Manual*, Iowa State University, Version 5.41, April 1999.

[PVS] The PVS Specification and Verification System. http://pvs.csl.sri.com.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In: T. Margaria and W. Yi editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS),* LNCS 2031, pp. 299-312. Springer, 2001.

[Wing87] Jeannette M. Wing. Writing Larch interface language specifications. ACM Transactions on Programming Languages and Systems, 9(1):1–24, January 1987.

[Winskel93] Glynn Winskel. The Formal Semantics of Programming Languages: An Introduction. Foundations of Computing Serices. MIT Press, 1993.

Literals and Logical Variables

$$\frac{}{\rho \vdash c_\tau \longrightarrow c : \tau}\ \text{Literal} \qquad \frac{\rho(\text{LVAR}\,\iota) = \tau}{\rho \vdash \iota \longrightarrow \iota : \tau}\ \text{Logical Var}$$

Operators

$$\frac{\begin{array}{c} op \in \mathit{unsafeOps} \wedge \rho(\texttt{\textbackslash safeMath}) = \texttt{true} \\ \rho \vdash e_i \longrightarrow \epsilon_i : \tau_i \quad \text{for } i = 1..k \\ (\tau_1, ..., \tau_k) \in \mathit{ArgType}_{op} \qquad \mathit{JML.conv}_{op}(\tau_1, ..., \tau_k) = (\tau'_1, ..., \tau'_k) \\ \mathit{resultType}_{op} = \tau \qquad (\tau'_1, ..., \tau'_k \to \tau) \in \rho(\text{OP}\,op).\mathit{types} \end{array}}{\rho \vdash op(e_1, ..., e_k) \longrightarrow \phi_{op}(\epsilon_1, ..., \epsilon_k) : \tau}\ \text{Op}_{\text{safe}}$$

$$\frac{\begin{array}{c} op \notin \mathit{unsafeOps} \vee \rho(\texttt{\textbackslash safeMath}) = \texttt{false} \\ \rho \vdash e_i \longrightarrow \epsilon_i : \tau_i \quad \text{for } i = 1..k \\ (\tau_1, ..., \tau_k) \in \mathit{ArgType}_{op} \qquad \mathit{Java.conv}_{op}(\tau_1, ..., \tau_k) = (\tau'_1, ..., \tau'_k) \\ \mathit{resultType}_{op} = \tau \qquad (\tau'_1, ..., \tau'_k \to \tau) \in \rho(\text{OP}\,op).\mathit{types} \end{array}}{\rho \vdash op(e_1, ..., e_k) \longrightarrow \phi_{op}(\epsilon_1, ..., \epsilon_k) : \tau}\ \text{Op}$$

Warn/Nowarn Operators

$$\frac{\rho \oplus \{\ \texttt{\textbackslash safeMath} \mapsto \texttt{false}\ \} \vdash e \longrightarrow \epsilon : \tau}{\rho \vdash \texttt{\textbackslash nowarn}(e) \longrightarrow \epsilon : \tau}\ \text{nowarn} \qquad \frac{\rho \oplus \{\ \texttt{\textbackslash safeMath} \mapsto \texttt{true}\ \} \vdash e \longrightarrow \epsilon : \tau}{\rho \vdash \texttt{\textbackslash warn}(e) \longrightarrow \epsilon : \tau}\ \text{warn}$$

$$\frac{\begin{array}{c} e'_i = \texttt{\textbackslash warn}(e_i) \quad \text{for } i = 1..k \\ \rho \vdash \texttt{\textbackslash nowarn}(op(e'_1, ..., e'_k)) \longrightarrow \epsilon : \tau \end{array}}{\rho \vdash \texttt{\textbackslash nowarn\_op}(op(e_1, ..., e_k)) \longrightarrow \epsilon : \tau}\ \text{nowarn\_op} \qquad \frac{\begin{array}{c} e'_i = \texttt{\textbackslash nowarn}(\epsilon_i) \quad \text{for } i = 1..k \\ \rho \vdash \texttt{\textbackslash warn}(op(e'_1, ..., e'_k)) \longrightarrow \epsilon : \tau \end{array}}{\rho \vdash \texttt{\textbackslash warn\_op}(op(e_1, ..., e_k)) \longrightarrow \epsilon : \tau}\ \text{warn\_op}$$

Casts

$$\frac{\rho \vdash e \longrightarrow \epsilon : \tau' \qquad \tau' \leq \tau}{\rho \vdash (\tau)e \longrightarrow \epsilon : \tau}\ \text{Widen} \qquad \frac{\rho \vdash e \longrightarrow \epsilon : \tau' \qquad \tau' > \tau}{\rho \vdash (\tau)e \longrightarrow \texttt{narrow(}\epsilon\texttt{)} : \tau}\ \text{Narrow}$$

Old & Quantifier Expressions

$$\frac{\rho \oplus \{\ \texttt{\textbackslash state} \mapsto \text{`}\texttt{pre}\text{'}\ \} \vdash e \longrightarrow \epsilon : \tau}{\rho \vdash \texttt{\textbackslash old}(e) \longrightarrow \epsilon : \tau}\ \text{old} \qquad \frac{\begin{array}{c} \rho \oplus \{\ \text{LVAR}\,\iota \mapsto \tau\ \} \vdash e \longrightarrow \epsilon : \texttt{boolean} \\ \tau \in \mathit{PrimitiveNumeric} \end{array}}{\rho \vdash (q\ \tau\ \iota\ e) \longrightarrow \texttt{(}q'\ \ \texttt{(}\iota : \tau\texttt{)}\texttt{:}\ \epsilon\texttt{)} : \texttt{boolean}}\ \text{Quant}$$

Static Field/Method Access

$$\frac{\begin{array}{c} (\text{CLASS}\,\iota') \in \operatorname{dom}\rho \\ \mathit{lookup}_{tp}(\rho, \iota', \iota) = \tau \\ \sigma = \rho(\texttt{\textbackslash state}) \end{array}}{\rho \vdash \iota'.\iota \longrightarrow \texttt{val(}\sigma\texttt{,}\ \iota'.\iota\texttt{)} : \tau}\ \text{Static Field} \qquad \frac{\begin{array}{c} \rho \vdash e_i \longrightarrow \epsilon_i : \tau_i \quad \text{for } i = 1..k \\ (\text{CLASS}\,\iota') \in \rho \\ \mathit{lookup}_{tp}(\rho, \iota', \iota) = [\tau_1, ..., \tau_k \to \tau'] \end{array}}{\rho \vdash \iota'.\iota(e_1, ..., e_k) \longrightarrow \iota'.\iota(\epsilon_1, ..., \epsilon_k) : \tau'}\ \text{Static Method}$$

Figure 8. JMLa expression semantics, selected rules