Original version of logic documentation (ESCJ 8) by K. Rustan M. Leino and Jim Saxe, 30 December 1997. Some additions by Cormac Flanagan, 24 April 1998. Modified by K. Rustan Leino, 13 September 1999 (making ESCJ 8a from ESCJ 8).

Original version of Java to Guarded Command Translation (ESCJ 16c) by K. Rustan M. Leino and Jim Sax, 26 August 1998. First draft by Rustan and Raymie Stata, 11 April 1997. Additional comments by Cormac Flanagan.

Modified by Joseph Kiniry, with initial input from Cesare Tinelli and input from Silvio Ranise, beginning on 21 June 2004.

Current editor Joseph Kiniry.

This document describes the logics of ESC/Java2. All of these logics represent partial semantics for Java and JML. Some are sets of axioms that are part of the Java logic (and thus are always included in the background predicate) and sets of axioms that are introduced to the background predicate when the Java program being checked contains various constructs. The strongest postcondition and weakest precondition calculi described herein describes the translation of Java into guarded commands. The assumptions and assertions produced by this translation are also discussed when this will help the reader understand the interaction between the wp-based translation and the logic.

This is edition $Revision$.

This document is a work in progress. Suggestions and input are always welcome.

The Logics and Calculi of ESC/Java2

Edition $Revision$, November 2004

This document describes the logics of ESC/Java2 version 2.0a8 and later.

Joseph R. Kiniry <kiniry@acm.org>

Table of Contents

This document describes the logics of ESC/Java2. Each logic is described by: (a) a set of axioms that constitute the logic and are thus are always included in the background predicate), and (b) the axioms that are introduced to the background predicate when the Java program being checked contains various constructs.

The strongest postcondition and weakest precondition calculi that are used to translate Java programs into verification conditions in a guarded command form is also discussed.

1 Introduction

When we introduce constants, predicates, and functions, we display an indented pseudo-declaration, which includes a signature.

When describing the untyped logic of SRC ESC/Java version 1, these signatures are just for our intuitive understanding, since the logic of SRC ESC/Java is based upon the logic of Simplify, which is untyped.

See Appendix A [Unsorted Construct Index], page 26 summarizes all such pseudo-declarations in the untyped logic, as documented in Chapter 2 [Unsorted Logic], page 3.

We display axioms in itemized lists with a prefix like this:

- axiom

The mere appearance of the pseudo-declaration of an operator does not implicitly give rise to any axiom. In the discussion of axioms, we indicate possible alternative axioms, or axioms that might be generated in illustrative examples, like this:

– alternative axiom

To prefix expressions that are produced as part of assumptions and assertions by the translation, we use an itemized list like this:

+ assumption
+ assertion

In this document, the use of footnotes indicates a discussion of open design issues.

This document was written under the heavy influence of Leino, Saxe, and Flanagan's *The Logic of ESC/Java*, which was in turn written under the heavy influence of Dave Detlefs's *Logic of ESC/Modula-3*. This document also includes the updated contents of *Java to Guarded Command Translation* by Leino, Saxe, Stata, and Flanagan.

2 An Unsorted Logic

2.1 Preliminaries

The ESC/Java tool attempts to find errors in Java programs by translating annotated Java programs into guarded commands, deriving strongest postconditions or weakest preconditions for those guarded commands, and testing those preconditions with a theorem prover. We have chosen to use Simplify as that theorem prover. Our design of the logic of ESC/Java is strongly influenced both by the underlying logic of Simplify (of which our logic is an extension) and by efficiency considerations specific to Simplify. In this section, we describe, more or less, what the reader needs to know about Simplify in order to understand the logic and the motivation for some of our design decisions.

2.1.1 Terms and predicates

Simplify's logic is untyped, but makes a strong distinction between terms and predicates. Terms are expressions that represent values in an underlying value space. Predicates are expressions that represent truth values.

A term in Simplify is a term constant, a variable, or an application of a function to terms. Simplify provides some built-in term constants, such as "0" and "6", and some built-in functions, such as "`+`". It also provides mechanisms by which users can implicitly declare constants, variables, and functions.

A predicate in Simplify is predicate constant (like `TRUE`), an application of a built-in predicate symbol to terms, an application of a boolean connective to predicates, or a quantified predicate. Simplify's built-in predicate symbols include "==", "!=", and "`<`"; its built-in boolean connectives include "&&", "`||`", and "!". (The actual symbols used by Simplify differ from those in this document, which uses a general mathematical syntax. For example, we write $x \neq y$ where Simplify expects `(NEQ x y)`. The different syntax should not create any confusion, except possibly for the built-in predicate symbol `EQ` and the built-in boolean connective `IFF`, both of which we write as ==. We hope the context of == will help disambiguate.) Throughout this document, the implication operator ==`>` binds more loosely than other logical connectives.

While Simplify does not allow a user to declare new predicate symbols, it allows the user to designate some function symbols to be usable where predicate symbols are expected. If `f` is such a function symbol, then whenever Simplify encounters an expression `f(...)` where a predicate is expected, it treats this expression, which would normally represent a term, as sugar for the predicate `f(...) == boolTrue`. (Note that `boolTrue` is a built-in term constant, not the built-in predicate constant `TRUE`.) We refer to such a function symbol `f` as a *user-defined predicate symbol* or, by even greater abuse of the language, a *predicate*. In this document, we write

- $foo : Predicate[bar \times gorp]$

to show that we intend to use the function symbol

- $foo : bar \times gorp \mapsto value$

as a user-defined predicate symbol.

2.1.2 Quantifiers and Triggering Patterns

We said above that Simplify has term constants and variables, but we weren't specific about what distinguishes them. Constants include not only numeric literals such as 6, but also symbolic constants that the uninitiated reader might naively perceive as variables. Symbol names are considered variables only when they are bound by a quantifier. As an example, consider the following axioms for group theory:

– $(\forall x :: times(e, x) == x)$

– $(\forall x :: times(inv(x), x) == e)$

- $(\forall x, y, z :: times(x, times(y, z)) == times(times(x, y), z))$

Here, x, y, and z are variables, but e is a constant. As another example, if Simplify is given the axioms

- $s < f(s)$
- $f(s) < t$

(from which it could successfully prove the conjecture s + $f(s)$ < 2 * t), then s and t are constants whose values are not known, other than that they satisfy the given axioms. The axiom s < $f(s)$, in which s is a constant, is entirely different from

- $(\forall s :: s < f(s))$

in which s is a variable.

A *ground term* is a term that contains no variables. The heart of Simplify's proving machinery is a set of procedures for testing the satisfiability of collections of equalities, distinctions (!=), and arithemetic inequalities of ground terms. To handle the boolean connectives, Simplify uses case analysis; to handle quantified expressions, Simplify uses Skolemization and matching as explained next.

When a quantified predicate is postulated to have a definite truth value–either directly by the user or as a result of case analysis–one of two things happens. If an existentially quantified predicate is postulated to be `TRUE`, Simplify introduces a Skolem constant for each of its variables, substitutes the Skolem constants for the variables in the body, and postulates the result. If a universally quantified predicate is postulated to be `TRUE`, Simplify produces a *matching rule*.

A matching rule represents a universally quantified predicate in a form that enables the prover to produce potentially relevant instantiations of its body in response to the detection of ground terms matching certain *triggering patterns*. For example, postulating the axiom

- $(\forall x :: times(e, x) == x)$

produces a matching rule with the triggering pattern $times(e,\ x)$. Whenever the prover finds a ground term of the form $times(e,\ T)$, it will instantiate the body of the axiom with $x := T$, that is, it will postulate $times(e,\ T) == T$.

The choice of triggering patterns for matching rules can impact both the completeness and the performance of the prover. Simplify has heuristics for automatically choosing triggering patterns, but allows a user to override the heuristics and specify the triggering patterns explicitly. In this document, we use underlining to indicate the triggering patterns of matching rules. For example, we would write the group theory identity axiom as

- $(\forall x :: \underline{times(e, x)} == x)$

to indicate that $times(e,\ x)$ is used as the triggering pattern of the resulting matching rule. In order to improve performance, we have attempted to write axioms and choose triggers in such a way as to reduce the cost of pattern matching and to reduce the likelihood that the prover will produce instantiations that lead to useless case splits. It might be tempting, in the quest for efficiency, to write axioms that are actually inconsistent and to depend on the choice of restrictive triggers to prevent the inconsistency from coming into play and causing bogus verifications to succeed; we have resisted this temptation.

Sometimes we must use a set of terms as a triggering pattern instead of a single term. For example, for a quantified predicate like

- $(\forall s, t, x :: member(x, s) \wedge subset(s, t) \longrightarrow member(x, t))$

no single term is an adequate trigger, since no single term contains all the quantified variables. An appropriate trigger is the set of terms $member(x,\ s)$, $subset(s,\ t)$:

- $(\forall s, t, x :: \underline{member(x, s)} \wedge \underline{subset(s, t)} \longrightarrow member(x, t))$

With this *multi-trigger*, the body will be instantiated upon the detection of a pair of ground terms matching $member(x, s)$ and $subset(s, t)$, with the same ground term matched to s. Although sometimes needed, multi-trigger matching is generally more expensive than single-trigger matching.

Note that triggering patterns are sets of terms, not predicates. Thus, it is not possible to specify the following trigger:

– $(\forall s, t, x :: \underline{member(x, s) \wedge subset(s, t)} \longrightarrow member(x, t))$

Neither is it possible to specify a trigger containing a built-in predicate symbol, such as `<` or ==.

2.1.3 Predicate Definitions

Simplify provides a mechanism by which a defining expression may be provided as part of the declaration of a user-defined predicate symbol P. Whenever an application of P is made equal to or distinct from `boolTrue`, the defining expression is instantiated with appropriate substitutions for the arguments and the resulting predicate or its negation, respectively, is postulated. By using this kind of definition, instead of separately introducing a universally quantified axiom, two sorts of efficiency improvements may result. First, we avoid invoking Simplify's general purpose pattern matching. Second, by instantiating the definition of a user-defined predicate only when an application's truth value becomes known, rather than when an application is introduced, we may avoid gratuitous case splitting. (Of course, there is a danger that we will sometimes postpone useful case splitting.)

When in this document we intend a given axiom $(\forall args :: P(args) == ...)$ to be the defining expression for a user-defined predicate P, we will use the notation

- Definition: $(\forall args :: P(args) == ...)$

2.1.4 The *as* Trick

In this section, we describe a technique, used in several of the axioms below, that allows us to choose triggering patterns that Simplify can match efficiently but that will not lead to extraneous matches.

In a world with types, a typical axiom might look like

– $(\forall x : X, y :: \underline{P(x, y)} \longrightarrow Q(x, y))$

where x is quantified over all values of type X and y is unconstrained. Since Simplify is type-free, so is our logic. The straightforward way of encoding the axiom above would be to introduce a predicate *isX* characterizing values of type X:

– $(\forall x, y :: isX(x) \wedge P(x, y) \longrightarrow Q(x, y))$

But what should be the triggering pattern of this axiom?

If we choose $P(x, y)$ as the triggering pattern, then Simplify is likely to instantiate the axiom with substitutions $x, y := t0, t1$ even where $t0$ is not known to satisfy *isX*. The result may be to cause the prover to do a useless case split with the cases $\neg isX(t0)$, $\neg P(t0, t1)$, and $Q(t0, t1)$. Even if $P(t0, t1)$ is known to hold, we can get a two-way case split.

Intuitively, we want to use the axiom only when x is already known to be of the correct type–this would be the common interpretation of the typed version of the axiom. If we can arrange for other mechanisms to postulate $isX(x)$ whenever we're possibly interested in instantiating the axiom, then we can use the terms $isX(x)$ and $P(x, y)$ together as a *multi-trigger*. This reduces the likelihood of producing useless instantiations of the axiom, without loss of completeness. If, further, we make sure that ground terms matching $isX(x)$ are introduced only when they are also equated to `boolTrue`, then the untyped axiom will be instantiated only as often as the typed version would have been in a typed prover.

A disadvantage of the approach just described is that Simplify's matching process for multi-triggers is generally more expensive than for ordinary triggering patterns.

Instead of introducing the predicate *isX*, the approach we actually take is to introduce a function *asX*. Intuitively, *asX* casts any value into a value of type *X*, and is the identity on values that are already of type *X*. When introducing a term *t0* of type *X*, instead of assuming

+ $isX(t0)$

we assume

+ $t0 == asX(t0)$

This allows us to write the axiom as

– $(\forall x, y :: \underline{P(asX(x), y)} \longrightarrow Q(asX(x), y))$

Here we have a single-term trigger, which should be efficient to match. Also, since we introduce *asX* only with arguments that are known to be of type *X*, we avoid producing irrelevant instantiations.

(We could introduce both *isX* and *asX*, in which case we could either define *isX*(x) by the axiom

– $(\forall x :: isX(x) == (x == asX(x)))$

or characterize *asX* by the axioms

– $(\forall x :: isX(x) \longrightarrow x == asX(x))$

– $(\forall y :: isX(asX(y)))$

However, once we have *asX*, introducing *isX* seems redundant.)

In the example above, we replaced a one-argument predicate *isX* with a one-argument function *asX*. We can apply a similar technique for predicates with more than one argument. For example, instead of writing an axiom of the form

– $(\forall x, y, z :: isXwrtZ(x, z) \wedge P(x, y, z) \longrightarrow Q(x, y, z))$

we may introduce a function *asXwrtZ*, assume x == *asXwrtZ*(x, *z*) when we would have assumed *isXwrtZ*(x, *z*), and write the axiom as

– $(\forall x, y, z :: P(asXwrtZ(x, z), y, z) \longrightarrow Q(asXwrtZ(x, z), y, z))$

Multi-argument predicates like *isXwrtZ* are used to express more intricate properties than types can.

2.1.5 Maps

ESC/Java uses *maps* to represent instance variables, arrays, and lock sets. A map is like a function, but is a first-order value in the logic. The logic includes the following functions on maps:

- . [.] : map \times value \mapsto value
- *store* : map \times value \times value \mapsto map

The [ ] function is sometimes called *select*. The semantics of [ ] and *store* are given by the following axioms:

- $(\forall m, i, x :: \underline{store(m, i, x)[i]} == x)$
- $(\forall m, i, j, x :: i \neq j \longrightarrow \underline{store(m, i, x)[j]} == m[j])$

ESC/Java uses Simplify's built-in *select* and *store* functions. The second of these axioms is treated specially by Simplify in that the case splits suggested by it are given some priority over case splits suggested by ordinary axioms.

2.2 Types and Subtypes

2.2.1 Types

Java types are ordinary values in the logic of ESC/Java. Although the logic is untyped, we informally think of these values as having type "type".

The built-in types in Java give rise to the following type constants:

- *boolean* : type
- *char* : type
- *byte* : type
- *short* : type
- *int* : type
- *long* : type
- *float* : type
- *double* : type

In addition, declarations of classes and interfaces give rise to type constants. Each class or interface declaration

```
class T ...
```

or

```
interface T ...
```

introduces a type identifier

- *T* : type

Here and throughout this document, we assume that identifiers denoting types, fields, and variables have been unique-ified. Throughout this document, when we refer to declarations, we include both user-provided declarations and built-in declarations, like the classes `String` and `Object`, the interface `Cloneable`.

All type constants appear together in an axiom that postulates them all to be different:

- DISTINCT(*Object*, *boolean*, *char*, *byte*, *short*, *int*, *long*, *float*, *double*, *Cloneable*, ..., *String*, ..., *T*, ...)

This axiom is called the *Distinct Types Axiom*.

2.2.2 The subtype Predicate

The logic includes a subtype predicate:

<: : Predicate[type \times type]

The predicate *t0* <: *t1* means that *t0* is a subtype of *t1*. The operator <: binds as tightly as arithmetic relations such as <.

The following axioms are sound and complete in the sense that for any named class or interface types A and B,

```
     |= A <: B    if and only
if   |- A <: B
```

where `|-` refers to provability based on these axioms, and `|=` refers to the model given by Java's semantics.

In some cases we also need to prove negative subtype statements such as

```
     |- not( A <: B )
```

To illustrate the need for proving such statements, see Section B.4 [Try-Catch Example], page 30. Our current axiomatization of negative subtype statements is quite incomplete – we currently only include the antisymmetric axiom. We plan to investigate this issue more thoroughly in the future. The subtype relation is reflexive and transitive:

- $(\forall t :: \underline{t <: t})$
- $(\forall t0, t1, t2 :: \underline{t0 <: t1} \wedge \underline{t1 <: t2} \longrightarrow t0 <: t2)$

The subtype relation is also antisymmetric.

- $(\forall t0, t1 :: \underline{t0 <: t1} \wedge \underline{t1 <: t0} \longrightarrow t0 == t1)$

[1]

A class or interface declaration gives rise to axioms about where the type introduced fits into the subtype ordering.

For each class declaration

```
class C extends D implements J, K, ...
```

(where the absence of an `extends` clause is taken as sugar for `extends Object`), we add the following axioms to the background predicate:

- $C <: D$
- $C <: J$
- $C <: K$
- ...

We could include an axiom that describes the supertypes of `C`

– $(\forall t :: \underline{C <: t} \longrightarrow t == C \vee D <: t \vee J <: t \vee K <: t \vee ...)$

For the built-in class `Object`, this would yield[2]

– $(\forall t :: \underline{Object <: t} \longrightarrow t == Object)$

For each interface declaration

```
interface I extends J, K, ...
```

we add the following axioms[3]:

- $I <: Object$
- $I <: J$
- $I <: K$
- ...

We could include an axiom describing the supertypes of I, as above, but see no immediate need for it.

For each final type T (that is, a final class or one of the primitive types boolean, char, byte, short, int, long, float, or double), we add the following axiom, which says that T has no proper subtypes:

- $(\forall t :: \underline{t <: T} == (t == T))$

To see why this axiom is useful, see Section B.2 [Final Type Axioms Example], page 29.

2.2.3 Disjointness of Incomparable

This section is not implemented. It may be useful for examples such as Section B.4 [Try-Catch Example], page 30[4]

---

[1] An alternative would be to experiment with Simplify's built-in ordering theory, but we have concerns about its reliability and its impact on performance.

[2] Do we need this?

[3] This is redundant, but probably not harmful, if the the interface declaration bears an explicit `extends` clause.

[4] The axioms in this section are similar to some axioms introduced in the logic of ESC/Modula-3 to address a problem that arose in a program verification. It is not clear whether the problem has since been addressed by other mechanisms. While we can contrive examples where these axioms would be necessary for ESC/Java verifications, we don't know if such examples will arise naturally. We may choose not to exclude the material in this section without impact on the rest of the logic. In particular, there are no uses the functions *classDown* and *asChild* other than those described in this section.

For any two classes, either one is a subtype of the other, or they have no subtypes in common. The most obvious ways of axiomatizing this fact seem likely to lead to poor prover performance, for reasons that we will not describe further. The ESC/Java logic includes weaker axioms implying that distinct explicitly declared subclasses of any class (including `Object`) have no subtypes in common.

To this end, the logic includes two functions:

- *classDown* : type \times type \mapsto type
- *asChild* : type \times type \mapsto type

Intuitively, if *t0* is a proper subclass of *t2*, then *classDown*(*t2*, *t0*) is the direct subclass of *t2* that is a superclass of *t0*. Consider a class `A` with distinct explicitly declared direct subclasses `B` and `C`, and suppose that `BB` is any subclass of `B` and `CC` is any subclass of `C`. Then, *classDown*(A, BB) is B and *classDown*(A, CC) is C. If Simplify ever explores a case in which BB and CC are equal, it will infer by congruence closure that *classDown*(A, BB) and *classDown*(A, CC) are equal, and thus that B and C are equal, in contradiction to the [distinctTypesAxiom], page 7. Of course, if `BB` and `CC` were explicitly declared classes, we could infer their distinction directly from the Distinct Types Axiom. However, `BB` and `CC` might be the unknown dynamic types of objects with declared types `B` and `C`, respectively.

We want to formalize the definition of *classDown*. To do so, we must first formalize the notion of being a direct subclass. We could introduce a predicate *isDirectSubclass*, characterize *classDown* by the axiom

– $(\forall t0, t1, t2 :: \underline{t0 <: t1} \wedge \underline{isDirectSubclass(t1, t2)} \longrightarrow classDown(t2, t0) == t1)$

and let each class declaration

```
class C extends D ...
```

give rise to the axiom

– *isDirectSubclass*(C, D)

Instead, we avoid use of a multi-trigger by employing Section 2.1.4 [The as Trick], page 5: We characterize *classDown* by the axiom

- $(\forall t0, t1, t2 :: \underline{t0 <: asChild(t1, t2)} \longrightarrow classDown(t2, t0) == asChild(t1, t2))$

and for each class declaration

```
class C extends D ...
```

we introduce the axiom

- $C == asChild(C, D)$

2.2.4 Array Types

Array types do not give rise to type constants. Instead, the logic includes a function to produce an array type from an element type.

array : type \mapsto type

If t represents a type T, then *array*(t) represents the array type *T[]*.

Sometimes in this document we make reference to an arbitrary type `T`, which may or may not be an array type. For simplicity, we will denote its type T, even though the type of `T` may in fact not be represented by a type constant, but by an expression *array*(...).

All array types are subtypes of `Cloneable`:

- $(\forall t :: \underline{array(t)} <: Cloneable)$

Note that since `Cloneable` is a subtype of `Object`, every array type is, by transitivity, also a subtype of `Object`. Conversely, `Object` and `Cloneable` are the only non-array supertypes of array types, so for each class declaration

```
class T ...
```

(except for the built-in class `Object`) or interface declaration

```
interface T ...
```

(except for the built-in interface `Cloneable`), we could add the axiom[5]

– $(\forall t :: \underline{\neg(array(t) <: T)})$

An alternative approach would be to generate the axiom

- $(\forall t0, t1 :: array(t0) <: t1 \longrightarrow t1 == array(elemType(t1)) \vee Cloneable <: t1)$

where *elemType* is defined below. Technically, this is more complete, but it seems more likely to lead to unfruitful case splits.

The function *array* has a left inverse:

- *elemType* : type \mapsto type

with axiom[6]

- $(\forall t :: \underline{elemType(array(t))} == t)$

Intuitively, a type t is an array type if and only if $t == array(elemType(t))$. We could introduce a predicate *isArrayType* with the axiom

– $(\forall t :: isArrayType(t) == (t == array(elemType(t))))$

Instead, we simply write $t == array(elemType(t))$ wherever we would have written $isArrayType(t)$[7].

As stated in the following axiom, the subtypes of an array type `T[]` are the array types whose element types are subtypes of `T`. A use of this axiom is described in Section B.2 [Final Type Axioms Example], page 29 and Section B.3 [Array Element Subtype Example], page 30.

- $(\forall t0, t1 :: \underline{t0 <: array(t1)} == (t0 == array(elemType(t0)) \wedge elemType(t0) <: t1))$

2.3 Types of Values

2.3.1 The is Predicate

To reason about the dynamic types of values, the logic includes the following predicate:

- $is : Predicate[value \times type]$

For each variable identifier (global variable, parameter, or result value) `v` of type `T`, we assume

+ $is(v, T)$

as part of the precondition of the method being checked, after each method call that modifies `v`, and as an invariant of each loop that modifies `v`.

---

[5] This is not implemented because it is not clear we need to reason about the not-subtype relation.

Technically, we need this axiom only for direct subclasses of `Object`, direct subinterfaces of `Cloneable`, and direct subinterfaces of `Object` other than `Cloneable`.

[6] Is this the right pattern?

[7] While `Object` and `Cloneable` are not array types, they are supertypes of all array types. The non-object primitive types `boolean`, `char`, etc. are not supertypes of any array types, but we have not given axioms to that effect, because we are not sure they are needed in practice. The axioms we just gave, however, may be useful, as Section B.1 [Array Type-Constant Axioms Example], page 28 show.

2.3.2 Casting

The logic contains a function that converts a value to a value of a specified type:

- *cast* : value \times type \mapsto value
- $(\forall x, t :: is(\underline{cast(x, t)}, t))$

If the value is already of the specified type, then casting leaves it unchanged:

- $(\forall x, t :: is(x, t) \longrightarrow \underline{cast(x, t)} == x)$

In cases where casting in Java can fail, the translation produces appropriate checks. These checks will be described in another document.

The axioms above do not completely capture the semantics of casting as specified by Java. For example, Java specifies that casting an `int` to a `short` preserves the value modulo 2^16. We propose to omit such additional axioms about casting until the need for them arises.

2.3.3 Types of Primitive Values

2.3.3.1 Booleans

For booleans, the logic contains two distinct constants:

- $boolFalse : value$
- $boolTrue : value$
- $boolFalse \neq boolTrue$

In fact, these are the only boolean values. We could express this fact with the axiom

- $(\forall x :: \underline{is(x, boolean)} == (x == boolFalse \vee x == boolTrue))$

Since this axiom has the potential to lead to useless case splits, we're reluctant to use it. In Section 2.6.3 [Reflections of Predicates into Term Space], page 21, we'll say more about our approach to handling booleans.

2.3.3.2 Integers

To reason about the ranges of integer values, the logic includes the constants:

- *longFirst* : value
- *intFirst* : value
- *intLast* : value
- *longLast* : value

and the following axioms:

- $(\forall x :: \underline{is(x, char)} == (0 \leq x \wedge x \leq 65535))$
- $(\forall x :: \underline{is(x, byte)} == (-128 \leq x \wedge x \leq 127))$
- $(\forall x :: \underline{is(x, short)} == (-32768 \leq x \wedge x \leq 32767))$
- $(\forall x :: \underline{is(x, int)} == (intFirst \leq x \wedge x \leq intLast))$
- $(\forall x :: \underline{is(x, long)} == (longFirst \leq x \wedge x \leq longLast))$

The reason for giving *longFirst*, *intFirst*, *intLast*, and *longLast* as symbolic constants instead of exact values is that we don't want to assume the underlying theorem prover to be capable of dealing properly with such large constants[8].

Digression. The axioms above may seem unsound given that not all numbers between, say, -32768 and 32767 are integers: If we translated the Java expression `2.0 < x && x < 3.0` (where `x` is a Java `float`) directly into $2 < x \wedge x < 3$ (where `<` is Simplify's built-in comparison operator), then

---

[8] Will the use of constants like 65535 and 127 cause performance problems because of Simplify's integer programming heuristic?

the axioms above would let us conclude $is(x, int)$, which would be bad. To avoid this problem, we considered introducing a predicate *isMathInt* to characterize the mathematical integers and writing the axioms above as:

- $(\forall x :: \underline{is(x, char)} == (0 \leq x \land x \leq 65535 \land isMathInt(x)))$
- ...

We have rejected this approach on account of an infelicity in the implementation of Simplify: Simplify's complete decision procedure for rational linear inequalities is extended by an incomplete heuristic for integer inequalities. Unfortunately, this heuristic is applied indiscriminately rather than only to terms that are somehow designated as integers. For example, Simplify will find the conjunction 2 < x && x < 3 to be inconsistent, even if x corresponds to a Java `float`. Consequently, translating Java's floating-point < to Simplify's built-in < is untenable even with *isMathInt*. We have chosen to give a quite weak axiomatization of Java's floating point operators (see Section 2.6.3.3 [Reflected Floating-Point Comparisons], page 22), and in particular to use Simplify's built-in comparison operators only for integers. Therefore, we see no need for *isMathInt*. We could, of course, include *isMathInt* anyhow, for aesthetics, but we would then need to include such axioms as

- $(\forall x, y :: isMathInt(x) \land isMathInt(y) \longrightarrow isMathInt(\underline{x + y}))$

and to generate the assumption *isMathInt*(c) for every integer literal c occurring in the program. (End of Digression.)

Complications arise when the Java program being checked contains explicit integer constants of large magnitude. Our plan for treating such constants is to replace all explicit constants whose magnitude exceeds some threshold (say, 1000000) with symbolic constants, and to add to the background predicate sufficient axioms to establish the ordering of those symbolic constants with respect to each other, the threshold and its negation, and the symbolic constants *longFirst*, *intFirst*, *intLast*, and *longLast*. For example, if the program contains the explicit constants

- `-12000000`
- `72000`
- `800000`
- `12000000`
- `123456789`
- `1234567890123456789L`

then, using 1000000 as a threshold, the constants `-12000000`, `12000000`, `123456789`, and `1234567890123456789L` will be replaced by the symbolic constants *neg12000000*, *pos12000000*, *pos123456789*, and *pos1234567890123456789*, and the following axioms will be added to the background predicate:

- *longFirst* < *intFirst*
- *intFirst* < *neg12000000*
- *neg12000000* < -1000000
- 1000000 < *pos12000000*
- *pos12000000* < *pos123456789*
- *pos123456789* < *intLast*
- *intLast* < *pos1234567890123456789*
- *pos1234567890123456789* < *longLast*

Note that in the absence of such large constants, we will have the following axioms:

- *longFirst* < *intFirst*
- *intFirst* < -1000000

- $1000000 < \mathit{intLast}$
- $\mathit{intLast} < \mathit{longLast}$

The axioms we have described for casts and integer values are sufficient to guarantee, for example, that casting a `short` to an `int` does not change its value. Also, when an `int` is in the range -32768..32767, casting it to a `short` does not change its value. For an `int` that is not already a `short`, the axioms guarantee that casting it to a `short` will yield a result in the range -32768..32767, but don't specify the exact result even though the Java specification does[9].

2.3.3.3 Floating Point Values

The ESC/Java logic is weak in its treatment of floating point values. The following are not implemented[10].

– $(\forall x :: is(x, int) \longrightarrow is(x, double))$

– $(\forall x :: is(x, float) \longrightarrow is(x, double))$

Note that, despite these properties, *int* and *float* are not subtypes of *double*. (If they were, *array*(*int*) and *array*(*float*) would be subtypes of *array*(*double*), according to the axioms about *array* in Section 2.2.4 [Array Types], page 9.)

2.3.4 Types of Objects

Every non-`null` object has a unique dynamic type, as determined by the *typeof* operator:

- *typeof* : value \mapsto type

A value is of a reference type `T` if the value is `null` or if its dynamic type is a subtype of `T`[11]:

- $(\forall x, t :: \underline{t <: Object} \longrightarrow \underline{is(x, t)} == (x == null \vee typeof(x) <: t)))$

We said in Section 2.3.1 [The is Predicate], page 10 that the translation will introduce a precondition assumption *is*(v, T) for any parameter `v` of type `T`. For the `this` parameter of an instance method of a class `C`, the translation introduces the following stronger precondition:

+ $this \neq null \wedge typeof(this) <: C$

2.3.4.1 Instantiable Types

The dynamic type of a non-`null` object must be an instantiable type. The logic could includes a predicate

- *instantiable* : Predicate[type]

and the axiom[12]

– $(\forall x :: \underline{instantiable(typeof(x))})$

For each interface declaration

```
interface T ...
```

or abstract class declaration

```
abstract class T ...
```

the background predicate contains the following axiom[13]:

– !*instantiable*(T)

---

[9] It remains to be seen if practice calls for more axioms.

[10] What are the triggers?

[11] Will this produce useless case splits?

[12] Is this the right trigger?

[13] Is *instantiable* useful in practice? We can omit the predicate *instantiable* and its associated axioms without impact on the remainder of the logic.

2.3.4.2 Types of Instance Variables

ESC/Java models instance variables (fields) as maps from objects to values. Where in Java one writes the r-value `x.f`, the translation writes $f[x]$.

To reason about the dynamic types of values of fields, the logic includes the following function:

- *asField* : map \times type \mapsto map

To encode that a field identifier `f` has range type `T`, the translation introduces the assumption

+ $f == \mathit{asField}(f,\ T)$

as part of the precondition of the method being checked, after each method call that modifies `f`, and as an invariant of each loop that modifies `f`. This is another application of the aforementioned Section 2.1.4 [The as Trick], page 5. The logic includes the axiom

- $(\forall f, t, x :: is(\underline{asField(f, t)[x]}, t))$

Notice that this axiom does not include an antecedent requiring that x be a non-`null` object of the class that declares `f`. We believe that this treatment of a fields as total maps with their declared range types is harmless to the soundness of the logic, and may be beneficial to prover efficiency.

2.3.4.3 Types of Array Elements

ESC/Java models the state of all arrays using a single global variable called *elems*. Where in Java one writes the r-value `a[i]`, the translation writes $\mathit{elems}[a][i]$. This uses the same *select* function as above for fields, twice[14].

To reason about the dynamic types of array elements, the logic includes the following function:

- *asElems* : map \mapsto map

Applying yet again the Section 2.1.4 [The as Trick], page 5, the translation introduces the assumption

+ $\mathit{elems} == \mathit{asElems}(\mathit{elems})$

as part of the precondition of the method being checked, after each method call that modifies *elems*, and as an invariant of each loop that modifies *elems*. This assumption is used to supply a trigger for the following axiom:

- $(\forall e, a, i :: is(\underline{asElems(e)[a][i]}, elemType(typeof(a))))$

Notice that this axiom does not include antecedents requiring that a be a non-`null` array object and that i be in bounds. We believe that this treatment is harmless to the soundness of the logic, and may be beneficial to prover efficiency.

2.4 Allocation

In this section, we introduce machinery for reasoning about the allocation of objects, and in particular for showing that a newly allocated object is distinct from any object reachable from program variables prior to its allocation. Although our motivating discussions are long, the resulting axioms are few and simple.

2.4.1 Allocation Times of Objects

Consider the following method:

```
void m(T x)
  T y = new T();
  /* assert x != y; */
```

---

[14] An alternative to using a single global variable *elems* would be to use a variable *objectElems* to model all arrays of objects and additional variables for each of the primitive types, with *intElems* modeling all arrays of `ints`, etc. Having separate variables may improve prover efficiency, but would complicate the translation into guarded commands (and the logic itself). We propose to keep things simple for the initial version of ESC/Java.

Intuitively, the reason that the assertion succeeds is that x is already allocated at the start of the method body, whereas the result of the constructor call `new T()` is an object not yet allocated before the call. To formalize this, we introduce a program variable *alloc*, which somehow models which objects have been allocated. As we shall see below, we actually model *alloc* as a time. We also introduce a predicate

- $isAllocated : Predicate[value \times time]$

where *isAllocated*(x, *aa*) means that object x has been allocated prior to time *aa*. For each variable identifier (global variable, parameter, or result value) `v` of an object type, the translation assumes

+ $isAllocated(v, alloc)$

as part of the precondition of the method being checked, after each method call that modifies `v`, and as an invariant of every loop that modifies `v`. Finally, the translation includes the following postcondition as part of the specification of `new T()`:

- $\neg isAllocated(result, alloc) \wedge isAllocated(result, alloc')$

where *alloc* and *alloc*' are the values of *alloc* before and after the call, respectively, and *result* is the value returned by the call. The guarded command translation of the method `m` is thus something like:

```
assume isAllocated(x, alloc) && ... ;
var y in
   assume y == null ;
   var result, alloc' in
     assume ! isAllocated(result,alloc) && isAllocated(result, alloc')&& ... ;
     alloc = alloc';
     y = result
   end ;
assert x != y
end
```

The verification condition for this piece of code is:

- $isAllocated(x, alloc) \wedge ... \longrightarrow (\forall y :: y == null \longrightarrow (\forall result, alloc' :: \neg isAllocated(result, alloc) \wedge ... \longrightarrow x! = result))$

so the verification succeeds.

Now, consider the following method:

```
void n(T x)
  p();
  T y = new T();
  /* assert x != y; */
```

where `p()` denotes a method call that modifies *alloc*. In order to verify the assertion, we must be able to infer that x is still allocated after the call to `p`. One possible approach would be for the translation to explicitly assume

+ $isAllocated(x, alloc)$

after the call [oh, how we wish we had hollow square bullets...]. We reject this approach, since it would require generating such an assumption for each variable in the program, instead of just those that are modified by the call. Another approach would be for the translation add to the following as a postcondition of every method that modifies *alloc*:

+ $(\forall v :: \underline{isAllocated(v, alloc)} \longrightarrow isAllocated(v, alloc'))$

We actually use yet a different approach, which we hope will achieve better efficiency by making use of Simplify's built-in Simplex algorithm. First, we let *alloc* denote a time. Second, we introduce a function from objects to their allocation times:

- $vAllocTime : value \mapsto time$

Third, we define *isAllocated* in terms of *vAllocTime* and Simplify's built-in `<` operator:

- Definition: $(\forall x, aa :: isAllocated(x, aa) == (vAllocTime(x) < aa))$

Fourth and finally, the translation assumes

+ $alloc0 \leq alloc$

after every method call that modifies *alloc* (where *alloc0* is the value of *alloc* before the call), and as an invariant of every loop that modifies *alloc* (where *alloc0* is the value of *alloc* before the loop)[15].

2.4.2 Closure of Allocatedness under Field Access

In Section 2.4.1 [Allocation Times of Objects], page 14, we introduced rules by which ESC/Java can verify that a newly allocated object is distinct from previous values of program variables. We may also need to verify that newly allocated objects are distinct from all objects accessible prior to allocation, as in the following example:

```
void m(U u)
  T y = new T();
  /* assert u.f != y; */
```

Indeed, it is an invariant of the language that fields of allocated objects are allocated. In this subsection, we show how ESC/Java formalizes this invariant.

One possible way to formalize the invariant would be to introduce a predicate *isFieldClosed*, characterized by the following axiom:

– $(\forall x, f, aa :: isFieldClosed(f, aa) \wedge isAllocated(x, aa) \longrightarrow isAllocated(f[x], aa))$

and to have the translation to assume, at appropriate points, *isFieldClosed*(*f*, *alloc*) for each field *f* whose range type is an object type.

The question now is: What are "appropriate points"? It would be nice not to have to re-assume *isFieldClosed*(*f*, *alloc*) after calls to a method `m` that does not modify *f*, even if `m` modifies *alloc*. Consider an object *x* such that *isAllocated*(*x*, *alloc*) holds after some call to `m`, and suppose we need to infer that *isAllocated*(*f*[x], *alloc*) holds. We proceed by case analysis: If *isAllocated*(*x*, *alloc0*) holds, where *alloc0* is the allocation time before the call to `m`, then by the axiom above *isAllocated*(*f*[x], *alloc0*) holds. From `m`'s postcondition *alloc0* `<=` *alloc*, the definition of *isAllocated*, and the transitivity of `<`, the desired inference is possible. Suppose, on the other hand, that $\neg isAllocated(x, alloc0)$. Then, $f[x] == null$, since f was not changed. Hence, we're done.

There are two problems with the approach just described. First, it may give rise to unnecessary case splits. Second, it inhibits an optimization that we'd like to do: If a method `m` modifies a field *f* only at newly allocated objects, we don't want to require that *f* be included in the **modifies** clause of `m`'s specification. Thus, we cannot assume, as we did in the informal proof above, that *f* is *null* at unallocated objects. Instead, the model we use is that, as seen by the caller, the method `m` allocates objects whose *f* fields already have the "right" values. Indeed, `m` might be seen as allocating a "pre-existing" cyclic structure of objects. Hence, what we would like to formalize is not merely the invariant that the current value of *alloc* is closed under the current value of *f*, but also that all future values of *alloc* are closed under the current value of *f*.

Because of the things we have just discussed, the logic includes the function

- $fClosedTime : map \mapsto time$

---

[15] Is it worth attempting to detect cases where method calls leave *alloc* unchanged as far as the caller is concerned?

where $fClosedTime(f)$ is a time beyond which all allocation times are closed under f:

- $(\forall x, f, aa :: fClosedTime(f) < aa \land isAllocated(x, aa) \longrightarrow \underline{isAllocated(f[x], aa)})$

Like the axioms about the types of fields and array elements, this axiom does not have an antecedent restricting the values at which maps are applied. For each field identifier `f`, the translation assumes

+ $fClosedTime(f)$ `<` *alloc*

as part of the precondition of the method being checked, after each method call that modifies `f`, and as an invariant of every loop that modifies `f`.

Note that we could instead have introduced the predicate *isFieldClosed* mentioned above, but with the axiom:

– $(\forall f, aa :: \underline{isFieldClosed(f, aa)} == fClosedTime(f) < aa)$

Then the solid-bulleted axiom and translation assumption above could have been written using *isFieldClosed*. Since we see no need to use *isFieldClosed* as a triggering pattern, it seems more straightforward to use `<` directly. (Note that the solid-bulleted axiom does use *isAllocated* in a trigger. This is why we include the function *isAllocated* in the logic, instead of replacing it every with its definition.)

2.4.3 Closure of Allocatedness under Array Access

The preceding subsection introduced machinery that formalizes the language invariant that $f[x]$ is allocated whenever x is allocated. We use similar machinery to formalize the invariant that *elems*[*a*][*i*] is allocated whenever *a* is allocated.

The logic includes the function

- $eClosedTime : map \mapsto time$

and the axiom

- $(\forall a, e, i, aa :: eClosedTime(e) < aa \land isAllocated(a, aa) \longrightarrow \underline{isAllocated(e[a][i], aa)})$

The translation assumes

+ $eClosedTime(elems) < alloc$

as part of the precondition of the method being checked, after each method call that modifies *elems*, and as an invariant of each loop that modifies *elems*.

2.5 Locking

ESC/Java checks for race conditions and deadlocks. The translation introduces a global map variable *LS*, called the *lock set*, that characterizes the set of locks held by the current thread; a lock *mu* is held whenever *LS*[*mu*] == `boolTrue`. (Recall that in Java, a lock is exactly the same thing as an object.) To check for race conditions, the programmer supplies annotations telling which shared variables are protected by which locks. Whenever a shared variable is accessed and it is necessary to check whether its lock *mu* is in the lock set, the translation generates the check

- $assertLS[mu] == boolTrue$

To check for deadlocks, the programmer supplies annotations defining a relation lockLess (written as `<` in annotations) on locks:

- $lockLess : Predicate[value \times value]$

This *lockLess* relation is transitively closed:

- $(\forall x0, x1, x2 :: \underline{(lockLess x0 x1)} \land \underline{(lockLess x1 x2)} => (lockLess x0 x2))$

Esc/Java verifies that locks are only acquired by any thread in ascending order. If the lockLess order is acyclic, then this guarantees absence of deadlock. (If the programmer erroneously specifies a cyclic ordering, then deadlock may result, but no other error-checking property of ESC/Java is affected.)

It is convenient to assume the invariant that LS has a maximal element. To see that this assumption is sound, note that the existence of a maximal element follows if LS is totally ordered, finite, and nonempty. Since LS can be extended only by the acquisition of a lock greater than all locks currently held, and since a method can acquire only one new lock at a time, it follows that if LS is initially totally ordered and finite, it will remain so throughout the execution of any ESC/Java-legal program. Finally, it is harmless to assume that LS initially contains a sentinel element smaller than any lock acquired during the execution.

To reason about which variables denote lock sets, the logic includes a function

- $asLockSet : map \mapsto map$

and the translation assumes as a precondition of the method being checked that LS is a valid lock set:

+ $LS == asLockSet(LS)$

In addition, the logic includes a function for extracting the maximum of a lock set:

- $lockSetMax : map \mapsto value$
- $(\forall S :: \underline{asLockSet(S)[lockSetMax(asLockSet(S))]} == boolTrue)$

The translation assumes as a precondition of the method being checked that every lock in the lock set is allocated[16]:

+ $(\forall mu :: LS[mu] \longrightarrow isAllocated(mu, alloc))$

Since there are no unmatched acquires or releases in Java, the value of LS is left unchanged by method calls and loops. Hence, there is no reason to repeat this assumption later in the translation of the method being checked.

The translation generates

+ $lockLess(lockSetMax(LS), this) \vee LS[this] == boolTrue$

as a precondition of every call to a synchronized non-static method, and generates

+ $lockLess(lockSetMax(LS), T) \vee LS[T] == boolTrue$

as a precondition of every call to a synchronized static method of a class `T`. If the method being checked is synchronized, then the translation assumes the precondition

+ $LS[this] == boolTrue$

if the method is non-static and

+ $LS[T] == boolTrue$

if the method is a static method of class `T`.

A synchronized block

```
synchronized (mu)  S
```

is translated into the guarded command

```
assert lockLess(lockSetMax(LS),mu) || LS[mu]== boolTrue ;
var oldLS in
    assume oldLS == LS ;
    var newLS in
        assume(lockLess(lockSetMax(LS), mu) && mu== lockSetMax(newLS)) ||
                   (LS[mu] == boolTrue && newLS == LS);
        assumenewLS == store(LS, mu,boolTrue) ;
       assume newLS == asLockSet(newLS);
        LS= newLS ;
        S  (* actually, the translation of S *)
```

---

[16] WHY???

```
        end ;
        LS = oldLS
    end
```

The assumption

- **assume**$(lockLess(lockSetMax(LS), mu) \wedge mu == lockSetMax(newLS)) \vee (LS[mu] == boolTrue \wedge newLS == LS)$

is used to check calls and synchronized blocks within `S`. The assumption

- **assume**$newLS == store(LS, mu, boolTrue)$

is used to check shared-variable accesses in `S`. The function *store* is explained in Section 2.1.5 [Maps], page 6.

2.6 Domain-specific Axioms

Pretty much every occurrence of a built-in operator of Java gives rise to an occurrence of a corresponding function in the translation. For many of these functions, there are no axioms specifying their semantics, at least in the initial version of ESC/Java. This section explains those functions that are given a semantics.

2.6.1 Properties of Arrays

A deference of the `length` field of an array is translated into an application of the function *arrayLength*:

- $arrayLength : value \mapsto value$

Every array length is a non-negative `int`:

- $(\forall a :: 0 \leq \underline{arrayLength(a)} \wedge is(arrayLength(a), int))$

The rest of this subsection describes four functions and one predicate used to simplify the translation of Java's `new` operator on array types, including multi-dimensional array types:

- $shapeOne : value \mapsto shape$
- $shapeMore : value \times shape \mapsto shape$
- $arrayParent : value \mapsto value$
- $arrayPosition : value \mapsto value$
- $arrayFresh : Predicate[value \times time \times time \times map \times shape \times type \times value]$

The functions *shapeOne* and *shapeMore* construct array shapes. Intuitively, a shape is a nonempty list of integers, representing the dimensions of a rectangular array. For example, *shapeOne*(6) would be the shape of a one-dimensional array of length 6, and *shapeMore*(12, *shapeOne*(7)) would be the shape of a two-dimensional array of length 12, each of whose elements is a one-dimensional array of length 7.

Execution of the Java construct `new T[E1][E2]...[En]` allocates $1 + 1 * E1 + 1 * E1 * E2 + ... + 1 * E1 * E2 * ... * E(n-1)$ distinct arrays. The functions *arrayParent* and *arrayPosition* are used to ensure that these arrays are in fact distinct, as described below.

The translation of the Java construct `new T[E1][E2]...[En]` includes an assumption like

+ $arrayFresh(a, alloc, alloc', elems, shapeMore(E1, shapeMore(E2, ...(shapeOne(En))...)), array(array(...($

where *a* is the newly allocated array, *alloc* and *alloc'* are the allocation times just before and after the allocation of *a*, *elems* is the global variable modeling the state of all arrays (see Section 2.3.4.3 [Types of Array Elements], page 14), and *zero* is the zero-equivalent value of type `T`.

Informally, the predicate `arrayFresh(a, aa, bb, e, s, T, v)` states that *a* is a non-`null` array allocated between the allocation times *aa* and *bb*, of type T and shape *s*, whose leaf elements in *e* are *v*. By "leaf elements in *e*", we mean values of the form $e[a][i]$ in case *s* is a one-dimensional shape, values of the form $e[e[a][i]][j]$ in case *s* is a two-dimensional shape, etc. Formally, *arrayFresh* is defined by the following axioms:

- $(\forall a, aa, bb, e, n, s, T, v :: \underline{arrayFresh(a, aa, bb, e, shapeMore(n, s), T, v)} == aa \leq vAllocTime(a) \land vAllocTime(a) < bb \land a! = null \land typeof(a) == T \land arrayLength(a) == n \land (\forall i :: arrayFresh(\underline{e[a][i]}, aa, bb, e, s, elemType(T), v) \land arrayParent(e[a][i]) == a \land arrayPosition(e[a][i]) == i))$
- $(\forall a, aa, bb, e, n, T, v :: \underline{arrayFresh(a, aa, bb, e, shapeOne(n), T, v)} == aa \leq vAllocTime(a) \land vAllocTime(a) < bb \land a! = null \land typeof(a) == T \land arrayLength(a) == n \land (\forall i :: \underline{e[a][i]} == v))$

Note that these axioms contain nested quantifications, which themselves have triggering patterns. Note also that the inner quantifications do not include antecedents requiring that i be in bounds. As we have remarked before, we believe that this treatment is harmless to the soundness of the logic, and may be beneficial to prover efficiency.

To see how the use of the functions *arrayParent* and *arrayPosition* ensure that the arrays allocated as part of a multi-dimensional array allocation are distinct, consider the following program fragment:

```
int[][][] a = new int[10][10][10];
/* assert a[3] != a[4]; */
/* assert a[3][7] != a[4][7]; */
/* assert a[3] != a[4][7]; */
```

The translation and the logic together ensure, after the allocation, that $arrayPosition(elems[a][3]) == 3$ and that $arrayPosition(elems[a][4]) == 4$, so when the prover considers the possibility that the first assertion fails (that is, that $elems[a][3] == elems[a][4]$, it will derive the contradiction $3 == 4$. The translation and logic also ensure that $arrayParent(elems[elems[a][3]][7]) == elems[a][3]$ and that $arrayParent(elems[elems[a][4]][7]) == elems[a][4]$, so when the prover considers the possibility that second assertion fails, it will derive $elems[a][3] == elems[a][4]$, which leads to the contradiction $3 == 4$ as just explained. Finally, the translation and logic ensure that $typeof(elems[a][3]) == array(array(int))$ and that $typeof(elems[emes[a][4]][7]) == array(int)$. As discussed in Section B.1 [Array Type-Constant Axioms Example], page 28, the axioms in Section 2.2.4 [Array Types], page 9 guarantee that the types $array(array(int))$ and $array(int)$ are distinct, so when the prover considers the possibility that the third assertion fails, it will derive a contradiction[17].

2.6.2 Arithmetic Functions on Integers

The Java `+`, `-`, `*`, `<`, `<=`, `==`, `!=`, `>=`, and `>` operators on integers are translated to the corresponding built-in operators of Simplify, which bring Simplify's equality and simplex decision procedures into play.

The Java `/` and `%` operators on integers are translated into the functions *integralDiv* and *integralMod*, respectively:

- $integralDiv : value \times value \mapsto value$
- $integralMod : value \times value \mapsto value$

The appropriate axioms are[18]:

- $(\forall i, j :: integralDiv(i, j) * j + \underline{integralMod(i, j)} == i)$
- $(\forall i, j :: 0 < j \longrightarrow 0 \leq \underline{integralMod(i, j)} \land integralMod(i, j) < j)$
- $(\forall i, j :: j < 0 \longrightarrow j < \underline{integralMod(i, j)} \land integralMod(i, j) \leq 0)$
- $(\forall i, j :: \underline{integralMod(i + j, j)} == integralMod(i, j))$
- $(\forall i, j :: \underline{integralMod(j + i, j)} == integralMod(i, j))$

---

[17] In Section 2.3.4.3 [Types of Array Elements], page 14, we discussed the possibility of splitting *elems* into multiple variables (*objectElems*, *intElems*, etc.). Such a change to the logic would complicate these axioms.

[18] Are these axioms and triggers well chosen?

2.6.3 Reflections of Predicates into Term Space

The next set of axioms we discuss relates to an issue that arises in the translation. The guarded command language makes a strong distinction between predicates and terms. A guard must be a predicate; the right-hand side of an assignment is a term. Simplify maintains a similar separation; it defines built-in predicates, and everything else is a term. Java, on the other hand, makes no such strong distinction. The condition of a conditional statement is just an expression of type `boolean`; the same expression could occur on the right-hand side of an assignment. Consequently, depending on the context in which a Java expression occurs, its translation produces either a predicate or a term. For example, the guard of the Java statement

```
if (x < y)  ...
```

can translate into the predicate $x < y$, while the right-hand side of the assignment statement

```
b = x < y;
```

must translate into a term $intLess(x, y)$. The function *intLess* (axiomatized below) is a reflection of `<` into the term space. The logic includes the following functions reflecting Java operators that produce booleans:

- $boolAnd : Predicate[value \times value]$
- $boolOr : Predicate[value \times value]$
- $boolNot : Predicate[value]$
- $boolEQ : Predicate[value \times value]$
- $floatingEQ : Predicate[value \times value]$
- $floatingLE : Predicate[value \times value]$
- $floatingLE : Predicate[value \times value]$

In this section, we discuss these functions and their axiomatizations.

2.6.3.1 Reflected Boolean Connectives

We start by describing a design decision related to the treatment of booleans. Recall that in Section 2.3.3.1 [booleanConstants], page 11 we remarked that we hesitated to include the axiom

– $(\forall x :: \underline{is(x, boolean)} == (x == boolFalse \vee x == boolTrue))$

for fear that it would lead to irrelevant case splits. Therefore, we take a different approach. Instead of assuming that there are only two values of type *boolean*, we axiomatize the reflected versions of the boolean connectives in such a way that the value `boolTrue` corresponds to the Java predicate `true`, and all values distinct from `boolTrue` correspond to the Java predicate `false`.

- Definition: $(\forall b, c :: boolAnd(b, c) == (b == boolTrue \wedge c == boolTrue))$
- Definition: $(\forall b, c :: boolOr(b, c) == (b == boolTrue \vee c == boolTrue))$
- Definition: $(\forall b :: boolNot(b) == (b \neq boolTrue))$
- Definition: $(\forall b, c :: boolEQ(b, c) == ((b == boolTrue) == (c == boolTrue)))$

(Recall that some occurrences of $==$ denote Simplify's built-in predicate symbol `EQ` and other denote Simplify's built-in boolean connective `IFF`. Also recall that Simplify allows applications of user-defined predicate symbols to be used syntactically either as terms or as predicates. In the case of the latter, Simplify implicitly compares them with `boolTrue`. For clarity, since our focus in this section is to describe reflections into term space, we use the functional form.)

2.6.3.2 Reflected Integer and Object Comparisons

To compare objects or integers for equality, the translator to Simplify generates the Simplify predicates EQ, `<` etc.

The Java operator `instanceof` is reflected by the user-defined predicate symbol *is*, which we have already described in Section 2.3.1 [The is Predicate], page 10.

2.6.3.3 Reflected Floating-Point Comparisons

Comparing floating-point values is not the same as comparing integers, for two reasons. For one thing, the Java expression `r == r`, where `r` is a Java `float` or `double`, sometimes doesn't evaluate to `true`, since `r` may be NaN (Not-a-Number). The other difference arises from an infelicitous feature in the implementation of Simplify, described in a digression in [isMathIntDigression], page 11. Thus, it is untenable to axiomatize *floatingEQ*, *floatingLE*, and *floatingLE* in the obvious way:

- Definition: $(\forall x, y :: floatingEQ(x, y) == (x == y))$
- Definition: $(\forall x, y :: floatingLE(x, y) == (x < y))$
- Definition: $(\forall x, y :: floatingLE(x, y) == (x \leq y))$

We could include such axioms as

- Definition: $(\forall x, y :: floatingEQ(x, y) == (\neg isNaN(x) \wedge \neg isNan(y) \wedge x == y))$

and axioms relating *floatingEQ*, *floatingLE*, and *floatingLE* to the floating-point arithmetic functions. However, we propose to omit all such axioms from the initial version of ESC/Java and to add them only as the need becomes evident.

2.6.3.4 Lifting Predicate Terms to Predicate Space

When a Java boolean variable `b` occurs in a context where a Java predicate is expected, as in the program fragment

```
if (b)  ...
```

the translation into guarded commands *lifts* the boolean term `b` into predicate space by comparing it to `boolTrue`:

- **if**$b == boolTrue \mapsto ...$

When a boolean expression occurs in such a context, we have a choice of how much of the "computation" to do in predicate space and how much to do in term space. For example, we might translate

```
if (b && x < y)  ...
```

in any of the following ways:

- **if** boolAnd(b, intLess(x, y)) == boolTrue \mapsto ...
- **if** b == boolTrue && intLess(x,y) == boolTrue \mapsto ...
- **if**$b == boolTrue \wedge x < y \mapsto ...$

A description of the exact translation algorithm, which also includes treatment of short-circuit boolean operators and expressions with side effects, is beyond the scope of this document.

In order to avoid the need to lower predicates into term space, users are not allowed to use genuine predicate expressions (namely, quantified expressions) as subexpressions of terms. For example, specifications cannot contain expressions like

- $store(myBooleanArray, i, (forall...))$

As it happens, we plan not to allow users to explicitly write *store* at all. However, see the discussion of the conditional operator in the next section.

2.6.3.5 Reflecting the Conditional Operator

Occurrences of the Java conditional operator `? :` in executable Java code pose no problems–the translation can handle these just as it handles short-circuit boolean operators and expressions with side effects. On the other hand, occurrences of the conditional operator in specifications will in general require a reflected operator.

- $termConditional : value \times value \times value \mapsto value$
- $(\forall x, y :: \underline{termConditional(boolTrue, x, y)} == x)$

- $(\forall b, x, y :: b \neq boolTrue \longrightarrow \underline{termConditional(b, x, y)} == y)$

An alternative would be to write the one axiom[19]

- $(\forall b, x, y :: (b == boolTrue \wedge \underline{termConditional(b, x, y)} == x) \vee (b! = boolTrue \wedge termConditional(b, x, y) == y))$

Since we are introducing the function *termConditional* for use in the translation of specifications, the translation of executable code may also benefit from using it.

Note that if the boolean expression `B` in the specification expression `B ? X : Y` is allowed to contain a quantified expression when the types of `X` and `Y` are not `boolean`, then the translation will be rather difficult since quantified expressions are fundamentally predicates and there is no direct mechanism for lowering predicates into term space. We therefore propose to restrict conditional expressions from containing such guards.

2.6.4 Other Domain-Specific Axioms

There is a host of standard Java library classes, such as `String`, `Thread`, and `Reflection`, whose specifications, one can imagine, would require extending the logic of ESC/Java with more functions and axioms. We don't know to what extent we will need to specify these classes in order to do useful extended static checking of their clients. For example, to prove that the program fragment

```
ch = "hello".toCharArray()[2];
```

doesn't cause an array index out-of-bounds error, we may need to introduce a function *stringLength* in order to specify the method `String.toCharArray` and also to provide special treatment for `String` literals in the translation of Java to guarded commands. Other examples may require an axiomatization of *stringLength* that says that all `String` lengths are non-negative. We propose to add such functions and axioms only as the need becomes evident.

---

[19] Which is best?

3 Many-Sorted Logics

Discussion of new logics.

4 Calculi

Discussion of ESC/Java2's strongest postcondition and weakest precondition calculi for translating Java into verification conditions via a guarded command language.

4.1 Strongest Postcondition Calculus

Discussion of ESC/Java2's strongest postcondtion calculus for translating Java into verification conditions via a guarded command language.

4.2 Weakest Precondition Calculus

Discussion of ESC/Java2's weakest precondition calculus for translating Java into verification conditions via a guarded command language.

Appendix A Index of Constructs in Unsorted Logic

From Section 2.1.5 [Maps], page 6:

- $.[.] : map \times value \mapsto value$
- $store : map \times value \times value \mapsto map$

From ⟨undefined⟩ [types], page ⟨undefined⟩:

- boolean : type
- char : type
- byte : type
- short : type
- int : type
- long : type
- float : type
- double : type

From Section 2.2.2 [The subtype Predicate], page 7:

- $<:: Predicate[type \times type]$

From Section 2.2.3 [Disjointness of Incomparable], page 8:

- $classDown : type \times type \mapsto type$
- $asChild : type \times type \mapsto type$

From Section 2.2.4 [Array Types], page 9:

- array : type \mapsto type
- elemType : type \mapsto type

From Section 2.3.1 [The is Predicate], page 10:

- $is : Predicate[value \times type]$

From Section 2.3.2 [Casting], page 11:

- $cast : value \times type \mapsto value$

From Section 2.3.3.1 [booleanConstants], page 11:

- boolFalse : value
- boolTrue : value

From Section 2.3.3.2 [integerConstants], page 11:

- longFirst : value
- intFirst : value
- intLast : value
- longLast : value

From Section 2.3.4 [Types of Objects], page 13:

- $typeof : value \mapsto type$
- instantiable : Predicate[type]

From Section 2.3.4.2 [Types of Instance Variables], page 14:

- $asField : map \times type \mapsto map$

From Section 2.3.4.3 [Types of Array Elements], page 14:

- $asElems : map \mapsto map$

From Section 2.4.1 [Allocation Times of Objects], page 14:

- $isAllocated : Predicate[value \times time]$
- $vAllocTime : value \mapsto time$

From Section 2.4.2 [Closure of Allocatedness under Field Access], page 16:

- $fClosedTime : map \mapsto time$

From Section 2.4.3 [Closure of Allocatedness under Array Access], page 17:

- $eClosedTime : map \mapsto time$

From Section 2.5 [Locking], page 17:

- $lockLess : Predicate[value \times value]$
- $asLockSet : map \mapsto map$
- $lockSetMax : map \mapsto value$

From Section 2.6.1 [Properties of Arrays], page 19:

- $arrayLength : value \mapsto value$
- $shapeOne : value \mapsto shape$
- $shapeMore : value \times shape \mapsto shape$
- $arrayParent : value \mapsto value$
- $arrayPosition : value \mapsto value$
- $arrayFresh : Predicate[value \times time \times time \times map \times shape \times type \times value]$

From Section 2.6.2 [Arithmetic Functions on Integers], page 20:

- $integralDiv : value \times value \mapsto value$
- $integralMod : value \times value \mapsto value$

From Section 2.6.3 [Reflections of Predicates into Term Space], page 21:

- $boolAnd : Predicate[value \times value]$
- $boolOr : Predicate[value \times value]$
- $boolNot : Predicate[value]$
- $boolEQ : Predicate[value \times value]$
- $floatingEQ : Predicate[value \times value]$
- $floatingLE : Predicate[value \times value]$
- $floatingLE : Predicate[value \times value]$

From Section 2.6.3.5 [Reflecting the Conditional Operator], page 22:

- $termConditional : value \times value \times value \mapsto value$

Appendix B Motivating Examples, Functions, and Constants

B.1 Array Type-Constant Axioms Example

This example is out-of-date, and may no longer be relevant.

We give an example to motivate the axioms in Section 2.2.4 [Array Types], page 9 that distinguish primitive types from array types, such as

- $int \neq array(elemType(int))$

Consider the program fragment

```
a[i][j] = 6;
/* assert a[i][j] == 6; */
```

where `a` is a variable of type `a[][]`. The translation turns this into a guarded command like

- elems = store(elems, elems[a][i], store(elems[elems[a][i]], j, 6)) ;
- **assert** elems[elems[a][i]][j] == 6

(For simplicity, we have left out the bounds checks.) The verification condition associated with this guarded command is:

```
store(elems,
      elems[a][i],
      store(elems[elems[a][i]], j, 6))
[store(elems,elems[a][i],store(elems[elems[a][i]],j, 6))[a][i]]
[j] == 6
```

Suppose we know $elems[a][i] \neq a$. Then, we can simplify the red underlined *select* expression to:

- elems[a]

so that the entire verification condition becomes:

$$\underline{store(elems, elems[a][i], store(elems[elems[a][i]], j,\ \ 6))[elems[a][i]][j]\ \ ==\ \ 6}$$

Since $elems[a][i] == elems[a][i]$, we can now simplify another *select* of *store* expression, reducing the verification condition to:

- store(elems[elems[a][i]],j, 6))
- [j] == 6

Since $j == j$, this reduces to:

- 6 == 6

which is true.

But we needed $elems[a][i] \neq a$. We have that $typeof(elems[a][i]) == array(int)$, whereas $typeof(a) == array(array(int))$. Hence, it suffices to know that these two types are different.

We end by showing how the axioms from Section 2.2.4 [Array Types], page 9 can help. Suppose that Simplify explores a potential satisfying assignment in which the two types are postulated to be equal:

0. $array(int) == array(array(int))$

By the Section 2.2.4 [Array Types], page 9 axiom

- 1. $(\forall t :: elemType(\underline{array(t)}) == t)$

we know that

- 2. elemType(array(int)) == int

and

- 3. elemType(array(array(int))) == array(int)

From 0, it follows by congruence closure that

- 4. elemType(array(int)) == elemType(array(array(int)))

and from 2, 3, and 4, it follows that

- 5. int == array(int)

From 2 and 5, we have

- 6. elemType(int) == int

From 5 and 6, we have

- 7. int == array(elemType(int))

contradicting the axiom

- 8. int != array(elemType(int))

given in Section 2.2.4 [Array Types], page 9.

Notice that to do the verification in this example, Simplify must consider and refute the case that $elems[a][i] \neq a$. The *select* of *store* axiom

- $(\forall m, i, j, x :: i \neq j \longrightarrow \underline{store(m, i, x)[j]} == m[j])$

will suggest the relevant case split, and give that case split a relatively high priority. However, we could avoid the case split altogether by changing the logic to split *elems* into multiple variables, as discussed in a remark in Section 2.3.4.3 [Types of Array Elements], page 14. If we did so, then the program fragment considered in this example would be translated into the guarded command

```
intElems = store(intElems,objectElems[a][i], store(intElems[objectElems[a][i]], j, 6)) ;
assert intElems[objectElems[a][i]][j] == 6
```

and the corresponding verification condition would be

```
store(intElems,
      objectElems[a][i],
      store(intElems[objectElems[a][i]],j, 6))
[objectElems[a][i]]
[j] == 6
```

and the verification can complete with no case splits and without the need for axiom 8. However, we would still need to perform a case split and to use axiom 8 for a similar example involving a 3-dimensional array.

B.2 Final Type Axioms Example

We give an example to motivate the final type axioms in Section 2.2.2 [The subtype Predicate], page 7. Consider the method:

```
void f(T[] a, T b) \
  a[0] = b;
\
```

Verifying this method requires ensuring that b is a subtype of the element type of a (which is non-trivial, since the element type of a may be a subtype of T). Simplify is given that:

```
typeof(a) <: array(T);
typeof(b) <: T;
```

and needs to prove that:

```
typeof(b) <: elemType(typeof(a));
```

From the array axiom triggered on the first antecedent, we have:

```
typeof(a) == array(elemType(typeof(a))) &&
elemType(typeof(a)) <: T;
```

If T is a final type, then the final type axiom is triggered, and yields that:

```
elemType(typeof(a)) == T;
```

and then the second antecedent yields the desired consequent.

B.3 Array Element Subtype Example

Considering the following variant of example 1:

```
void f(T[][] a, T[] b) \
  a[0] = b;
\
```

Verifying this method requires ensuring that b is a subtype of the element type of a (which is non-trivial, since the element type of a may be a subtype of T). Simplify is given that:

```
typeof(a) <: array(array(T));
typeof(b) <: array(T);
```

and needs to prove that:

```
typeof(b) <: elemType(typeof(a));
```

From the array axiom triggered on the first antecedent, we have:

```
typeof(a) == array(elemType(typeof(a))) &&
elemType(typeof(a)) <: array(T);
```

From the array axiom triggered on the last line, we have:

```
elemType(typeof(a)) == array(elemType(elemType(typeof(a)))) &&
elemType(elemType(typeof(a))) <: T;
```

If T is a final type, then the final type axiom is triggered, and yields that:

```
elemType(elemType(typeof(a))) == T;
elemType(typeof(a)) == array(T);
```

and hence the desired consequent holds. Note that one affect of the array axiom is to state that arrays of final classes are final.

B.4 A Try/Catch Example

Considering the following example (from test8/trycatch2.java)

```
class Try2 \
  void m1() throws Throwable \
    int x, y;
    Throwable t;
    try \
      x = 0;
      // assume typeof(t) == type(Throwable);
      // assume t != null;
      throw t;
    \ catch (RuntimeException t3) \
      x = 3;
    \
    // assert x == 0;
  \
\
```

To verify this class, Esc/Java needs to prove that

```
not (Throwable <: RuntimeException)
```

This motivates the need for the antisymmetry axiom.

Appendix C List of Possible Experiments

To be filled in.

Appendix D Logics

This appendix contains the raw source of every logic described in this document.

Section D.1 [Unsorted Logics], page 32 contains material relating to the original unsorted, Simplify-centric, object logic of SRC ESC/Java version 1. Section D.1.1 [Original Logic], page 32 contains the original unsorted logic. Section D.1.2 [Additional Axioms Introduced in Calvin and Houdini], page 43 contains all additional axioms that were added by various parties at SRC to the unsorted logics of the SRC tools Calvin and Houdini. Section D.1.3 [Additional Axioms Introduced in ESC/Java2], page 43 contains all additional axioms that have been added to ESC/Java2 by David Cok and Joe Kiniry.

Section D.2 [Many-Sorted Logics], page 46 contains a new many-sorted logics developed by Joe Kiniry and Cesare Tinelli. These logics are being developed for use with SMT-LIB based provers. They are not yet supported in ESC/Java2. These logics are being realized in the Maude logical framework and the PVS higher-order theorem prover.

D.1 Unsorted Logics

D.1.1 Original Logic

```
(PROMPT_OFF)
;----------------------------------------------------------------------
; "Universal", or class-independent part, of the background predicate

; === ESCJ 8: Section 0.4

(BG_PUSH (FORALL (m i x) (EQ (select (store m i x) i) x)))

(BG_PUSH (FORALL (m i j x)
 (IMPLIES (NEQ i j)
  (EQ (select (store m i x) j)
      (select m j)))))

; === ESCJ 8: Section 1.1

(DEFPRED (<: t0 t1))

; <: reflexive
(BG_PUSH
  (FORALL (t)
    (PATS (<: t t))
    (<: t t)))

; a special case, for which the above may not fire

(BG_PUSH (<: |T_java.lang.Object| |T_java.lang.Object|))

; <: transitive
(BG_PUSH
  (FORALL (t0 t1 t2)
    (PATS (MPAT (<: t0 t1) (<: t1 t2)))
    (IMPLIES (AND (<: t0 t1) (<: t1 t2))
      (<: t0 t2))))
```

```
;anti-symmetry
(BG_PUSH
 (FORALL
  (t0 t1)
  (PATS (MPAT (<: t0 t1) (<: t1 t0)))
  (IMPLIES (AND (<: t0 t1) (<: t1 t0)) (EQ t0 t1))))

; primitive types are final

(BG_PUSH (FORALL (t) (PATS (<: t |T_boolean|))
(IMPLIES (<: t |T_boolean|) (EQ t |T_boolean|))))
(BG_PUSH (FORALL (t) (PATS (<: t |T_char|))
(IMPLIES (<: t |T_char|) (EQ t |T_char|))))
(BG_PUSH (FORALL (t) (PATS (<: t |T_byte|))
(IMPLIES (<: t |T_byte|) (EQ t |T_byte|))))
(BG_PUSH (FORALL (t) (PATS (<: t |T_short|))
(IMPLIES (<: t |T_short|) (EQ t |T_short|))))
(BG_PUSH (FORALL (t) (PATS (<: t |T_int|))
(IMPLIES (<: t |T_int|) (EQ t |T_int|))))
(BG_PUSH (FORALL (t) (PATS (<: t |T_long|))
(IMPLIES (<: t |T_long|) (EQ t |T_long|))))
(BG_PUSH (FORALL (t) (PATS (<: t |T_float|))
(IMPLIES (<: t |T_float|) (EQ t |T_float|))))
(BG_PUSH (FORALL (t) (PATS (<: t |T_double|))
(IMPLIES (<: t |T_double|) (EQ t |T_double|))))

; (New as of 12 Dec 2000)
; primitive types have no proper supertypes

(BG_PUSH (FORALL (t) (PATS (<: |T_boolean| t))
(IMPLIES (<: |T_boolean| t) (EQ t |T_boolean|))))
(BG_PUSH (FORALL (t) (PATS (<: |T_char| t))
(IMPLIES (<: |T_char| t) (EQ t |T_char|))))
(BG_PUSH (FORALL (t) (PATS (<: |T_byte| t))
(IMPLIES (<: |T_byte| t) (EQ t |T_byte|))))
(BG_PUSH (FORALL (t) (PATS (<: |T_short| t))
(IMPLIES (<: |T_short| t) (EQ t |T_short|))))
(BG_PUSH (FORALL (t) (PATS (<: |T_int| t))
(IMPLIES (<: |T_int| t) (EQ t |T_int|))))
(BG_PUSH (FORALL (t) (PATS (<: |T_long| t))
(IMPLIES (<: |T_long| t) (EQ t |T_long|))))
(BG_PUSH (FORALL (t) (PATS (<: |T_float| t))
(IMPLIES (<: |T_float| t) (EQ t |T_float|))))
(BG_PUSH (FORALL (t) (PATS (<: |T_double| t))
(IMPLIES (<: |T_double| t) (EQ t |T_double|))))

; === ESCJ 8: Section 1.2

(BG_PUSH
  (FORALL (t0 t1 t2)
    (PATS (<: t0 (asChild t1 t2)))
    (IMPLIES
      (<: t0 (asChild t1 t2))
      (EQ (classDown t2 t0) (asChild t1 t2)))))

; === ESCJ 8: Section 1.3

; new

(BG_PUSH
  (<: |T_java.lang.Cloneable| |T_java.lang.Object|))

(BG_PUSH
  (FORALL (t)
    (PATS (array t))
    (<: (array t) |T_java.lang.Cloneable|)))

(BG_PUSH
  (FORALL (t)
    (PATS (elemtype (array t)))
    (EQ (elemtype (array t)) t)))

(BG_PUSH
  (FORALL (t0 t1)
    (PATS (<: t0 (array t1)))
    (IFF (<: t0 (array t1))
      (AND
(EQ t0 (array (elemtype t0)))
(<: (elemtype t0) t1)))))

; === ESCJ 8: Section 2.1

(DEFPRED (is x t))

(BG_PUSH
 (FORALL (x t)
 (PATS (cast x t))
 (is (cast x t) t)))

(BG_PUSH
 (FORALL (x t)
 (PATS (cast x t))
 (IMPLIES (is x t) (EQ (cast x t) x))))

; === ESCJ 8: Section 2.2

(BG_PUSH (DISTINCT |bool$false| |@true|))

; === ESCJ 8: Section 2.2.1

(BG_PUSH (FORALL (x)
   (PATS (is x |T_char|))
   (IFF (is x |T_char|) (AND (<= 0 x) (<= x 65535)))))
(BG_PUSH (FORALL (x)
   (PATS (is x |T_byte|))
   (IFF (is x |T_byte|) (AND (<= -128 x) (<= x 127)))))
(BG_PUSH (FORALL (x)
   (PATS (is x |T_short|))
   (IFF (is x |T_short|) (AND (<= -32768 x) (<= x 32767)))))
(BG_PUSH (FORALL (x)
   (PATS (is x |T_int|))
   (IFF (is x |T_int|) (AND (<= intFirst x) (<= x intLast)))))
(BG_PUSH (FORALL (x)
   (PATS (is x |T_long|))
   (IFF (is x |T_long|) (AND (<= longFirst x) (<= x longLast)))))

(BG_PUSH (< longFirst intFirst))
(BG_PUSH (< intFirst -1000000))
(BG_PUSH (< 1000000 intLast))
(BG_PUSH (< intLast longLast))

; === ESCJ 8: Section 2.3

(BG_PUSH
 (FORALL (x t)
 (PATS (MPAT (<: t |T_java.lang.Object|) (is x t)))
 (IMPLIES (<: t |T_java.lang.Object|)
  (IFF (is x t)
       (OR (EQ x null) (<: (typeof x) t))))))

; === ESCJ 8: Section 2.4

(BG_PUSH
 (FORALL (f t x) (PATS (select (asField f t) x))
 (is (select (asField f t) x) t)))

; === ESCJ 8: Section 2.5

(BG_PUSH
 (FORALL (e a i) (PATS (select (select (asElems e) a) i))
 (is (select (select (asElems e) a) i)
     (elemtype (typeof a)))))

; === ESCJ 8: Section 3.0

(DEFPRED (isAllocated x a0) (< (vAllocTime x) a0))

; === ESCJ 8: Section 3.1

(BG_PUSH
 (FORALL (x f a0) (PATS (isAllocated (select f x) a0))
 (IMPLIES (AND (< (fClosedTime f) a0)
       (isAllocated x a0))
  (isAllocated (select f x) a0))))

; === ESCJ 8: Section 3.2
```

```
(BG_PUSH
 (FORALL (a e i a0) (PATS (isAllocated (select (select e a) i) a0))
 (IMPLIES (AND (< (eClosedTime e) a0)
       (isAllocated a a0))
  (isAllocated (select (select e a) i) a0))))

; === ESCJ 8: Section 4

; max(lockset) is in lockset

(BG_PUSH
 (FORALL (S)
  (PATS (select (asLockSet S) (max (asLockSet S))))
  (EQ
   (select (asLockSet S) (max (asLockSet S)))
   |@true|)))

; null is in lockset (not in ESCJ 8)

(BG_PUSH
 (FORALL (S)
  (PATS (asLockSet S))
  (EQ (select (asLockSet S) null) |@true|)))

(DEFPRED (lockLE x y) (<= x y))

(DEFPRED (lockLT x y) (< x y))

; all locks in lockset are below max(lockset) (not in ESCJ 8)

(BG_PUSH
 (FORALL (S mu)
  (IMPLIES
   (EQ (select (asLockSet S) mu) |@true|)
   (lockLE mu (max (asLockSet S))))))

; null precedes all objects in locking order (not in ESCJ 8)

(BG_PUSH
  (FORALL (x)
    (PATS (lockLE null x) (lockLT null x) (lockLE x null) (lockLT x null))
    (IMPLIES
      (<: (typeof x) |T_java.lang.Object|)
      (lockLE null x))))


; === ESCJ 8: Section 5.0

(BG_PUSH
 (FORALL (a)
 (PATS (arrayLength a))
 (AND (<= 0 (arrayLength a))
      (is (arrayLength a) |T_int|))))

(DEFPRED (arrayFresh a a0 b0 e s T v))

(BG_PUSH
  (FORALL (a a0 b0 e n s T v)
    (PATS (arrayFresh a a0 b0 e (arrayShapeMore n s) T v))
    (IFF
      (arrayFresh a a0 b0 e (arrayShapeMore n s) T v)
      (AND
(<= a0 (vAllocTime a))
(isAllocated a b0)
(NEQ a null)
(EQ (typeof a) T)
(EQ (arrayLength a) n)
(FORALL (i)
  (PATS (select (select e a) i))
  (AND
    (arrayFresh (select (select e a) i) a0 b0 e s (elemtype T) v)
    (EQ (arrayParent (select (select e a) i)) a)
    (EQ (arrayPosition (select (select e a) i)) i)))))))

(BG_PUSH
  (FORALL (a a0 b0 e n T v)
    (PATS (arrayFresh a a0 b0 e (arrayShapeOne n) T v))
    (IFF
      (arrayFresh a a0 b0 e (arrayShapeOne n) T v)
      (AND
(<= a0 (vAllocTime a))
(isAllocated a b0)
(NEQ a null)
(EQ (typeof a) T)
(EQ (arrayLength a) n)
(FORALL (i)
  (PATS (select (select e a) i))
  (AND
    (EQ (select (select e a) i) v)))))))


; === code to ensure that (isNewArray x) ==> x has no invariants


; arrayType is distinct from all types with invariants (due to the
; generated type-distinctness axiom)

(BG_PUSH
  (EQ arrayType (asChild arrayType |T_java.lang.Object|)))

(BG_PUSH
   (FORALL (t)
      (PATS (array t))
      (<: (array t) arrayType)))

(BG_PUSH
  (FORALL (s)
  (PATS (isNewArray s))
  (IMPLIES (EQ |@true| (isNewArray s))
   (<: (typeof s) arrayType))))

; === ESCJ 8: Section 5.1

(BG_PUSH
 (FORALL (i j) (PATS (integralMod i j) (integralDiv i j))
 (EQ (+ (* (integralDiv i j) j) (integralMod i j))
     i)))

(BG_PUSH
 (FORALL (i j) (PATS (integralMod i j))
 (IMPLIES (< 0 j)
  (AND (<= 0 (integralMod i j))
       (< (integralMod i j) j)))))

(BG_PUSH
 (FORALL (i j) (PATS (integralMod i j))
 (IMPLIES (< j 0)
  (AND (< j (integralMod i j))
       (<= (integralMod i j) 0)))))

(BG_PUSH
 (FORALL (i j)
 (PATS (integralMod (+ i j) j))
 (EQ (integralMod (+ i j) j)
     (integralMod i j))))

(BG_PUSH
 (FORALL (i j)
 (PATS (integralMod (+ j i) j))
 (EQ (integralMod (+ j i) j)
     (integralMod i j))))

; to prevent a matching loop
(BG_PUSH
 (FORALL (x y)
  (PATS (* (integralDiv (* x y) y) y))
  (EQ (* (integralDiv (* x y) y) y) (* x y))))


; === ESCJ 8: Section 5.2

(DEFPRED (boolAnd a b)
  (AND
    (EQ a |@true|)
    (EQ b |@true|)))
```

```
(DEFPRED (boolEq a b)
  (IFF
    (EQ a |@true|)
    (EQ b |@true|)))

(DEFPRED (boolImplies a b)
  (IMPLIES
    (EQ a |@true|)
    (EQ b |@true|)))

(DEFPRED (boolNE a b)
  (NOT (IFF
 (EQ a |@true|)
 (EQ b |@true|))))

(DEFPRED (boolNot a)
  (NOT (EQ a |@true|)))

(DEFPRED (boolOr a b)
  (OR
    (EQ a |@true|)
    (EQ b |@true|)))

; Not in ESCJ8, but should be

(BG_PUSH
  (FORALL (x y)
    (PATS (integralEQ x y))
    (IFF
      (EQ (integralEQ x y) |@true|)
      (EQ x y))))

(BG_PUSH
  (FORALL (x y)
    (PATS (stringCat x y))
    (AND (NEQ (stringCat x y) null)
         (<: (typeof (stringCat x y)) |T_java.lang.String|))))

(BG_PUSH
  (FORALL (x y)
    (PATS (integralGE x y))
    (IFF
      (EQ (integralGE x y) |@true|)
      (>= x y))))

(BG_PUSH
  (FORALL (x y)
    (PATS (integralGT x y))
    (IFF
      (EQ (integralGT x y) |@true|)
      (> x y))))
```

```
(BG_PUSH
  (FORALL (x y)
    (PATS (integralLE x y))
    (IFF
      (EQ (integralLE x y) |@true|)
      (<= x y))))

(BG_PUSH
  (FORALL (x y)
    (PATS (integralLT x y))
    (IFF
      (EQ (integralLT x y) |@true|)
      (< x y))))

(BG_PUSH
  (FORALL (x y)
    (PATS (integralNE x y))
    (IFF
      (EQ (integralNE x y) |@true|)
      (NEQ x y))))

(BG_PUSH
  (FORALL (x y)
    (PATS (refEQ x y))
    (IFF
      (EQ (refEQ x y) |@true|)
      (EQ x y))))

(BG_PUSH
  (FORALL (x y)
    (PATS (refNE x y))
    (IFF
      (EQ (refNE x y) |@true|)
      (NEQ x y))))

; === ESCJ 8: Section 5.3

(BG_PUSH
 (FORALL (x y)
 (PATS (termConditional |@true| x y))
 (EQ (termConditional |@true| x y) x)))

(BG_PUSH
 (FORALL (b x y)
 (PATS (termConditional b x y))
 (IMPLIES (NEQ b |@true|)
  (EQ (termConditional b x y) y))))

; === Implementation of nonnullelements; not in ESCJ 8 (yet?):

(DEFPRED (nonnullelements x e)
   (AND (NEQ x null)
(FORALL (i)
   (IMPLIES (AND (<= 0 i)
 (< i (arrayLength x)))
    (NEQ (select (select e x) i) null)))))


; === Axioms about classLiteral; not in ESCJ 8 (yet?):

(BG_PUSH
 (FORALL (t)
 (PATS (classLiteral t))
 (AND (NEQ (classLiteral t) null)
      (is (classLiteral t) |T_java.lang.Class|)
              (isAllocated (classLiteral t) alloc))))


; === Axioms about properties of integral &, |, and /

(BG_PUSH
 (FORALL (x y)
  (PATS (integralAnd x y))
  (IMPLIES
   (OR (<= 0 x) (<= 0 y))
   (<= 0 (integralAnd x y)))))

(BG_PUSH
 (FORALL (x y)
  (PATS (integralAnd x y))
  (IMPLIES
   (<= 0 x)
   (<= (integralAnd x y) x))))

(BG_PUSH
 (FORALL (x y)
  (PATS (integralAnd x y))
  (IMPLIES
   (<= 0 y)
   (<= (integralAnd x y) y))))

(BG_PUSH
 (FORALL (x y)
  (PATS (integralOr x y))
  (IMPLIES
   (AND (<= 0 x) (<= 0 y))
   (AND (<= x (integralOr x y)) (<= y (integralOr x y))))))

(BG_PUSH
 (FORALL (x y)
  (PATS (integralDiv x y))
  (IMPLIES
   (AND (<= 0 x) (< 0 y))
```

```
        (AND (<= 0 (integralDiv x y)) (<= (integralDiv x y) x)))))

(BG_PUSH
 (FORALL (x y)
  (PATS (integralXor x y))
  (IMPLIES
   (AND (<= 0 x) (<= 0 y))
   (<= 0 (integralXor x y)))))

(BG_PUSH
 (FORALL (n)
  (PATS (intShiftL 1 n))
  (IMPLIES
   (AND (<= 0 n) (< n 31))
   (<= 1 (intShiftL 1 n)))))

(BG_PUSH
 (FORALL (n)
  (PATS (longShiftL 1 n))
  (IMPLIES
   (AND (<= 0 n) (< n 63))
   (<= 1 (longShiftL 1 n)))))

; === Temporary kludge to speed up distinguishing small integers:

(BG_PUSH
 (DISTINCT -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9
   10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
   30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
   50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
   70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
   90 91 92 93 94 95 96 97 98 99
   100 101 102 103 104 105 106 107 108 109
   110 111 112 113 114 115 116 117 118 119
   120 121 122 123 124 125 126 127 128 129
   130 131 132 133 134 135 136 137 138 139
   140 141 142 143 144 145 146 147 148 149
   150 151 152 153 154 155 156 157 158 159
   160 161 162 163 164 165 166 167 168 169
   170 171 172 173 174 175 176 177 178 179
   180 181 182 183 184 185 186 187 188 189
   190 191 192 193 194 195 196 197 198 199
   200 201 202 203 204 205 206 207 208 209
   210 211 212 213 214 215 216 217 218 219
   220 221 222 223 224 225 226 227 228 229
   230 231 232 233 234 235 236 237 238 239
   240 241 242 243 244 245 246 247 248 249
   250 251 252 253 254 255 256 257 258 259
   260 261 262 263 264 265 266 267 268 269
   270 271 272 273 274 275 276 277 278 279
   280 281 282 283 284 285 286 287 288 289
   290 291 292 293 294 295 296 297 298 299
```

```
     300 301 302 303 304 305 306 307 308 309
     310 311 312 313 314 315 316 317 318 319
     320 321 322 323 324 325 326 327 328 329
     330 331 332 333 334 335 336 337 338 339
     340 341 342 343 344 345 346 347 348 349
     350 351 352 353 354 355 356 357 358 359
     360 361 362 363 364 365 366 367 368 369
     370 371 372 373 374 375 376 377 378 379
     380 381 382 383 384 385 386 387 388 389
     390 391 392 393 394 395 396 397 398 399))

;----------------------------------------------------------------------
; End of Universal background predicate
;----------------------------------------------------------------------
(PROMPT_ON)
```

D.1.2 Additional Axioms Introduced in Calvin and Houdini

The following axioms were added to the object logic of Calvin. They have not yet been evaluated for inclusion in ESC/Java2.

```
; === Axiom for maps added by Shaz
(BG_PUSH
 (FORALL (f x t)
(PATS (MPAT (is f (map t)) (select f x)))
 (IMPLIES (is f (map t)) (is (select f x) t))))
; === End of Shaz's addition


; Fix suggested by Jim Saxe for the problem in DekkerLock where
; adding or removing asserts caused other asserts to break
(BG_PUSH
      (FORALL (a)
        (PATS (boolNot a))
        (OR (EQ a |@true|) (EQ (boolNot a) |@true|))))
; end of fix
```

Houdini introduced no new axioms.

D.1.3 Additional Axioms Introduced in ESC/Java2

The following axioms were introduced to the unsorted object logic to reason about new constructs introduced in ESC/Java2.

```
; === Define typeof for primitive types - DRCok

(BG_PUSH (FORALL (x)
   (PATS (typeof x))
   (IFF (is x |T_int|) (EQ (typeof x) |T_int|))))

(BG_PUSH (FORALL (x)
   (PATS (typeof x))
   ;(PATS (is x |T_short|))
   (IFF (is x |T_short|) (EQ (typeof x) |T_short|))))

(BG_PUSH (FORALL (x)
   (PATS (typeof x))
   ;(PATS (is x |T_long|))
   (IFF (is x |T_long|) (EQ (typeof x) |T_long|))))

(BG_PUSH (FORALL (x)
   (PATS (typeof x))
   ;(PATS (is x |T_byte|))
   (IFF (is x |T_byte|) (EQ (typeof x) |T_byte|))))

(BG_PUSH (FORALL (x)
   (PATS (typeof x))
   ;(PATS (is x |T_char|))
   (IFF (is x |T_char|) (EQ (typeof x) |T_char|))))

(BG_PUSH (FORALL (x)
   (PATS (typeof x))
   ;(PATS (is x |T_boolean|))
   (IFF (is x |T_boolean|) (EQ (typeof x) |T_boolean|))))

(BG_PUSH (FORALL (x)
   (PATS (typeof x))
   ;(PATS (is x |T_double|))
   (IFF (is x |T_double|) (EQ (typeof x) |T_double|))))

(BG_PUSH (FORALL (x)
   (PATS (typeof x))
   ;(PATS (is x |T_float|))
   (IFF (is x |T_float|) (EQ (typeof x) |T_float|))))

(BG_PUSH
  (FORALL (a0 b0 e s T v)
    (PATS (arrayMake a0 b0 s T v))
    (arrayFresh
(arrayMake a0 b0 s T v)
a0 b0 elems s T v)))

(BG_PUSH
  (FORALL (a0 b0 a1 b1 s1 s2 T1 T2 v)
    (PATS (MPAT (arrayMake a0 b0 s1 T1 v) (arrayMake a1 b1 s2 T2 v)))
    (IMPLIES
(EQ (arrayMake a0 b0 s1 T1 v) (arrayMake a1 b1 s2 T2 v))
(AND (EQ b0 b1) (EQ T1 T2) (EQ s1 s2)))))

(BG_PUSH
 (FORALL (i)
 (PATS (integralMod 0 i))
 (IMPLIES (NOT (EQ i 0))
  (EQ (integralMod 0 i) 0))))

(BG_PUSH
 (FORALL (t)
 (PATS (classLiteral t))
 (EQ (classLiteral t) t)))
```

```
; === A few floating point axioms - DRCok
;; FIXME - floatingLT etc are predicates here, but are functions in
;; ESC/java - is that a problem?

;; Order axioms
(BG_PUSH
 (FORALL (d dd)
  (AND
     (OR
(EQ |@true| (floatingLT d dd))
(EQ |@true| (floatingEQ d dd))
(EQ |@true| (floatingGT d dd)))
     (IFF (EQ |@true| (floatingLE d dd))
        (OR (EQ |@true| (floatingEQ d dd)) (EQ |@true| (floatingLT d dd))))
     (IFF (EQ |@true| (floatingGE d dd))
        (OR (EQ |@true| (floatingEQ d dd)) (EQ |@true| (floatingGT d dd))))
     (IFF (EQ |@true| (floatingLT d dd)) (EQ |@true| (floatingGT dd d)))
     (IFF (EQ |@true| (floatingLE d dd)) (EQ |@true| (floatingGE dd d)))
     (NOT (IFF (EQ |@true| (floatingLT d dd)) (EQ |@true| (floatingGE d dd))))
     (NOT (IFF (EQ |@true| (floatingGT d dd)) (EQ |@true| (floatingLE d dd)))))))


;; floatingNE
(BG_PUSH (FORALL (d dd) (IFF (EQ |@true| (floatingEQ d dd))
                             (NOT (EQ |@true| (floatingNE d dd))))))

;; floatADD
(BG_PUSH (FORALL (d dd)
  (IMPLIES (EQ |@true| (floatingGT d (floatingNEG dd)))
           (EQ |@true| (floatingGT (floatingADD d dd) |F_0.0|)))))

;; floatMUL (currently unused)
;;(BG_PUSH (FORALL (d dd) (AND
  ;;(IFF (OR (floatingEQ d |F_0.0|) (floatingEQ dd |F_0.0|))
  ;;     (floatingEQ (floatingMUL d dd) |F_0.0|))
  ;;(IMPLIES (AND (floatingGT d |F_0.0|) (floatingGT dd |F_0.0|))
  ;;         (floatingGT (floatingMUL d dd) |F_0.0|))
  ;;(IMPLIES (AND (floatingLT d |F_0.0|) (floatingLT dd |F_0.0|))
  ;;         (floatingGT (floatingMUL d dd) |F_0.0|))
  ;;(IMPLIES (AND (floatingLT d |F_0.0|) (floatingGT dd |F_0.0|))
  ;;         (floatingLT (floatingMUL d dd) |F_0.0|))
  ;;(IMPLIES (AND (floatingGT d |F_0.0|) (floatingLT dd |F_0.0|))
  ;;         (floatingLT (floatingMUL d dd) |F_0.0|)))))

;; floatingMOD
(BG_PUSH
 (FORALL (d dd)
  (IMPLIES (EQ |@true| (floatingNE dd |F_0.0|)) (AND
    (IMPLIES (EQ |@true| (floatingGE d |F_0.0|))
             (EQ |@true| (floatingGE (floatingMOD d dd) |F_0.0|)))
    (IMPLIES (EQ |@true| (floatingLE d |F_0.0|))
                  (EQ |@true| (floatingLE (floatingMOD d dd) |F_0.0|)))))))

(BG_PUSH (FORALL (d dd)
    (IMPLIES (EQ |@true| (floatingGT dd |F_0.0|)) (AND
        (EQ |@true| (floatingGT (floatingMOD d dd) (floatingNEG dd)))
        (EQ |@true| (floatingLT (floatingMOD d dd) dd))))))

(BG_PUSH (FORALL (d dd)
    (IMPLIES (EQ |@true| (floatingLT dd |F_0.0|))
             (EQ |@true| (floatingLT (floatingMOD d dd) (floatingNEG dd))))))

(BG_PUSH (FORALL (d dd)
    (IMPLIES (EQ |@true| (floatingLT dd |F_0.0|))
             (EQ |@true| (floatingGT (floatingMOD d dd) dd)))))
```

D.2 Many-Sorted Logics

This appendix section contains several different many-sorted object logics.

D.2.1 SMT-LIB Many-Sorted Logic

What follows is a many-sorted object logic for Java, written in SMT-LIB.

```
(theory escjava2-object-logic
  :notes "SMT-LIB realization of ESC/Java2's object logic.
          by Cesare Tinelli and Joe Kiniry
          Begun 24 June 2004
          $Revision: 1.3 $
          Based upon SRC ESC/Java object logic
            (design document ESCJ8a)"

  :sorts ( # sort that represents *values* of Java's boolean base type
           Boolean
           # sort that represents *values* of all Java's base types
           # but for Boolean
           Number
           # sort that represents all Java non-base types
           ReferenceType
           # ... represents object references
           Reference
           # ... represents object values
           Object
           # Boolean, Number, Object fields
           BooleanField
           NumberField
           ReferenceField
           # ... represents the heap
           Memory )

  # If we had subsorts, we would probably like to introduce the following:
  #   Time < Number
  #   Array < Object
  #   Boolean, Number, Reference < JavaType

  :funs ( # The heap.
          (HEAP    Memory)

          # Top of the object hierarchy.
          (java.lang.Object ReferenceType)

          (0       Number)
          (1       Number)

          (java.lang.Long.MAX_VALUE    Number)
          (java.lang.Long.MIN_VALUE    Number)
          (java.lang.Integer.MAX_VALUE Number)
          (java.lang.Integer.MIN_VALUE Number)
          (java.lang.Short.MAX_VALUE   Number)
          (java.lang.Short.MIN_VALUE   Number)
          (java.lang.Byte.MAX_VALUE    Number)
          (java.lang.Byte.MIN_VALUE    Number)
          (java.lang.Char.MAX_VALUE    Number)
          (java.lang.Char.MIN_VALUE    Number)

          (java.lang.Boolean.TRUE Boolean)
          (java.lang.Boolean.FALSE Boolean)

          # numeric downcasting (truncation)
          (narrowDouble2Float Number Number)
          (narrowDouble2Long  Number Number)
          (narrowDouble2Int   Number Number)

          (narrowFloat2Long  Number Number)
          (narrowFloat2Int   Number Number)

          (narrowLong2Int   Number Number)
          (narrowLong2Short Number Number)
          (narrowLong2Byte  Number Number)
          (narrowLong2Char  Number Number)

          (narrowInt2Short Number Number)
          (narrowInt2Byte  Number Number)
          (narrowInt2Char  Number Number)

          (narrowShort2Byte  Number Number)
          (narrowShort2Char  Number Number)

          (castByte2Char  Number Number)
          (castChar2Byte  Number Number)

          (NULL     Reference)
          (+        Number Number Number)
          (-        Number Number Number)
          ...etc...

          (!        Boolean Boolean)
          (&&       Boolean Boolean Boolean)
```

```
(||       Boolean Boolean Boolean)
...etc...

# Java ternary ?: operator
(?:Boolean     Boolean Boolean   Boolean)
(?:Number      Boolean Number    Number)
(?:Object      Boolean Reference Reference)

# array type constructors
(arrayBoolean   ReferenceType)
(arrayNumber    ReferenceType)
(arrayReference ReferenceType ReferenceType)

# \elementtype of object array
(elementReferenceType ReferenceType ReferenceType)

# dynamic type of a reference
(typeOf Reference ReferenceType)

# Java typecast
(cast   Reference ReferenceType Reference)

# TODO: perhaps rename getX and xGet

# Get/set a value from/at a specific index in an
# array of numbers.
(getNumber Object Number Number)
(setNumber Object Number Number Object)
# Get/set a value from/at a specific index in an
# array of booleans.
(getBoolean Object Number Boolean)
(setBoolean Object Number Boolean Object)
# Get/set a value from/at a specific index in an
# array of objects.
(getObject Object Number Reference)
(setObject Object Number Reference Object)

# Get/set objects in the memory (heap).
(memGet Memory Reference Object)
(memSet Memory Reference Object Memory)

# Get/set boolean fields of objects.
(boolSelect   BooleanField Object Boolean)
(boolStore    BooleanField Object Boolean BooleanField)
# Get/set number fields of objects.
(numberSelect NumberField Object Number)
(numberStore  NumberField Object Number NumberField)
# Get/set object fields of objects.
(referenceSelect ReferenceField Object Reference)
(referenceStore  ReferenceField Object Reference ReferenceField)

# Allocation time of a reference.
```

```
               (vAllocTime  Reference Number :injective)

               # Closure of allocation over objects and arrays.
               (fClosedTime ReferenceField Number)
               (eClosedTime Memory Number)

               # Array length as a first-order construct.
               (arrayLength Reference Number)
              )

       :preds ( # Java type predicates on numbers
                (isChar   Number)
                (isByte   Number)
                (isShort  Number)
                (isInt    Number)
                (isLong   Number)
                (isFloat  Number)
                (isDouble Number)

                # type predicate for mathematical integers
                (isZ      Number)

                # Java class and interface inheritance ("extends" and
                # "implements", but only smallest "implements" of a class,
                # not inherited implements)
                (extends ReferenceType ReferenceType)
                # Java subtyping
                (<:      ReferenceType ReferenceType :reflex :trans)

                # Cesare says these are built-in, since they are
                # not easily axiomatized.
                (<  Number Number)
                (<= Number Number)
                ...etc...

                # is-a predicate.  Corresponds to the static type of a
                # Java reference.
                (isa Reference ReferenceType)

                # abstract classes and interfaces are not instantiable
                # in Java
                (instantiable ReferenceType)

                # Is the referenced object allocated at the specified time?
                (isAllocated Reference Number)

                # Allocation of array elements.
                (arrayFresh Reference Memory Number )
          )

       :extensions ()
```

```
:definition "Unlike the logic in ESCJ8a, this is a many-sorted logic.
  Additionally, this is a prover-independent logic.  We only
  assume that the prover provides:
    o a theory of rational arithemetic
    o a theory of arrays
    o a theory of uninterpreted function symbols on which
      equality is defined"

:axioms ( # extends is antireflexive
          (forall ?x ReferenceType (not (extends ?x ?x)) )
          # subtype includes extends
          (forall ?x ReferenceType
             (forall ?y ReferenceType
                (implies (extends ?x ?y) (<: ?x ?y))))
          # subtype transitivity
          (forall ?x ReferenceType
             (forall ?y ReferenceType
                (forall ?z ReferenceType
                   (implies (and (<: ?x ?y)
                                 (<: ?y ?z))
                            (<: ?x ?z)))))
          # subtype reflexivity
          (forall ?x ReferenceType (<: ?x ?x))
          # subtype is the smallest relation that contains extends
          (forall ?x ReferenceType
             (forall ?y ReferenceType
                (implies (and (<: ?x ?y)
                              (not (= ?x ?y)))
                         (exists ?z ReferenceType
                            (and (extends ?x ?z)
                                 (<: ?z ?y))))))

          # <: is anti-symmetric
          (forall ?x ReferenceType
             (forall ?y ReferenceType
               (implies (and (<: ?x ?y) (<: ?y ?x))
                        (= (?x ?y)))))

          # java.lang.Object is top of subtype hierarchy
          (forall ?x ReferenceType (<: ?x java.lang.Object))

          # subtype rules for arrays
          (forall ?x ReferenceType
             (forall ?y ReferenceType
               (implies (<: ?x ?y)
                        (<: (array ?x) (array ?y)))))

          # left inverses of Object constructors
          (forall ?x ReferenceType
             (= (elementReferenceType (arrayObject ?x))
                ?x))
```

```
        # integral type boundaries
        (= minByte -128)
        (= maxByte 127)
        (forall ?x Number
           (iff (isByte ?x) (and (<= minByte ?x)
                                 (<= ?x maxByte))))
        ...

        # define an approximation of Z
        (isZ 0)
        (isZ 1)
        (forall ?x Number
          (iff (isZ ?x) (isZ (+ ?x 1))))
        (forall ?x Number
          (implies (isZ ?x)
            (forall ?y Number
              (implies (and (< ?x ?y)
                            (< ?y (+ ?x 1)))
                       (not (isZ ?y))))))

        # all Object Reference are NULL or have a unique
        # dynamic subtype
        (forall ?x Reference
          (forall ?t ReferenceType
            (iff (isa ?x ?t)
                 (or (= ?x NULL)
                     (typeOf ?x ?t)))))

        # definition of cast
        (forall ?x Reference
          (forall ?t ReferenceType
            (isa (cast ?x ?t) ?t)))

        # upcasting a value does not change it
        (forall ?x Reference
          (forall ?t ReferenceType
            (implies (isa ?x ?t)
                     (= (cast ?x ?t) ?x))))


        # TRUE and FALSE are distinct values
        (distinct TRUE FALSE)

# TRUE and FALSE are the only Boolean values
        (forall ?x Boolean (or (= ?x TRUE)
                               (= ?x FALSE)))

        # definition of Boolean operators
        ...

# definition of instantiable type
        (instantiable arrayBoolean)
```

```
(instantiable arrayNumber)
(instantiable java.lang.Object)
(forall ?x Reference (instantiable (typeOf ?x)))
(forall ?x ReferenceType (iff (instantiable ?x)
                           (instantiable (arrayObject ?x))))

# to be checked / modified by clement
(forall ?x ReferenceType
  (forall ?y ReferenceType
    (implies (and (subtype ?x ?y)
                  (instantiable ?y))
             (instantiable ?x))))


# for simplicity, get* and set* functions are defined
# on all Object values, whether they are arrays or not

(forall ?a object
  (for all ?i Number
    (forall ?x Number
       (= (getNumber (setNumber ?a ?i ?x) ?i)
          ?x))))
 (forall ?a Object
   (for all ?i Number
     (for all ?j Number
       (forall ?x Number
         (or (= ?i ?j)
             (= (getNumber (setNumber ?a ?i ?x) ?j)
                (getNumber ?a ?j)))))))

(forall ?a object
  (for all ?i Number
    (forall ?x Boolean
      (= (getBoolean (setBoolean ?a ?i ?x) ?i)
         ?x))))
(forall ?a Object
  (for all ?i Number
    (for all ?j Number
      (forall ?x Boolean
        (or (= ?i ?j)
            (= (getBoolean (setBoolean ?a ?i ?x) ?j)
               (getBoolean ?a ?j)))))))

(forall ?a object
  (for all ?i Number
    (forall ?x Reference
      (= (getObject (setObject ?a ?i ?x) ?i)
         ?x))))
(forall ?a Object
  (for all ?i Number
    (for all ?j Number
      (forall ?x Reference
        (or (= ?i ?j)
            (= (getObject (setObject ?a ?i ?x) ?j)
               (getObject ?a ?j)))))))


(forall ?m Memory
  (for all ?o Reference
    (forall ?x object
      (= (memGet (memSet ?m ?o ?x) ?o)
         ?x))))
(forall ?m Memory
  (for all ?o1 Reference
    (for all ?o2 Reference
      (forall ?x object
        (or (= ?o1 ?o2)
            (= (memGet (memSet ?m ?o1 ?x) ?o2)
               (memGet ?m ?o2)))))))

# axioms for select
...

# axioms for vAllocTime
# injectivity axiom of vAllocTime
(forall ?r1 Reference
   (forall ?r2 Reference
      (implies (= (vAllocTime ?r1) (vAllocTime ?r2))
               (= ?r1 ?r2))))

# definition of isAllocated in terms of vAllocTime
(forall ?r Reference
  (forall ?t Number
    (iff (isAllocated ?r ?t) (< (vAllocTime ?r ?t) ?t))))

# definition of fClosedTime.  Intuitively, what we are
# representing is that, if an object is allocated, then
# all of its fields are allocated.
(forall ?h Memory
  (forall ?r Reference
     (forall ?f ReferenceField
        (forall ?t Number
          (implies (and (< (fClosedTime ?f) ?t)
                        (isAllocated ?r ?t))
                   (isAllocated (referenceSelect ?f
                                  (memGet ?h ?r)) ?t))))))

# and the same axiom applies to arrays and their values
(forall ?h Memory
  (forall ?r Reference
     (forall ?i Number
        (forall ?t Number
          (implies (and (< (eClosedTime ?h) ?t)
                        (isAllocated ?r ?t))
```

```
                                      (isAllocated (getObject
                                                      (memGet ?h ?r) ?i) ?t))))))

                   # length of arrays
                   (forall ?r Reference
                      (let (length (arrayLength ?r))
                        (and (<= 0 length)
                              (isZ length))))


        )

     end)
```

D.2.2 Many-Sorted Logic with Subsorts

TBD

D.2.3 Maude Realization of Many-Sorted Logics

TBD

D.2.4 PVS Realization of Many-Sorted Logics

TBD

Appendix E Copying

Index