Preliminary Definition of Core JML

Gary T. Leavens David A. Naumann Stan Rosenberg

Stevens Institute of Technology, CS Report 2006-07

September 2006, revised June 2007 and December 2008

Abstract: The JML specification language has evolved over a number of years and several variations/subsets have been formalized, mainly in the context of prototype systems for runtime and static verification. This document records the preliminary definition of basic semantic concepts for a core fragment of JML. It is intended to facilitate investigation of new features and improvement in interoperability between tools. The formalization is based on a denotational semantics and has been encoded in the PVS theorem prover.

This material is based upon work supported by the
National Science Foundation under grants CCF-0429894, CCF-0429567, and
CNS-0627338.

Preliminary Definition of Core JML

Gary T. Leavens David A. Naumann Stan Rosenberg

December 15, 2008

Contents

1 Introduction

The JML specification language has evolved over a number of years. Several variations/subsets have been formalized, mostly as integral parts of prototype systems for runtime and static verification rather than as stand-alone mathematical definitions. This document records the preliminary definition of a core fragment, independent from any tool, to facilitate investigation of new features and improvement in interoperability between tools.

The formalization is based on a denotational semantics of Java. Verification tools typically implement some form of axiomatic semantics, often one which embodies various methodological assumptions such as behavioral subclassing.[1] Our aim is to provide foundations on which such semantics can be based. So we take pains to separate the semantics of the programming language from the semantics of specifications and other entities, such as ghost state, which are used in reasoning. Denotational semantics is more abstract in some sense than a small-step operational semantics that models an implementation. We claim that the denotational semantics is "obviously correct" in that the correspondence with operational semantics is too clear to merit formalization; on the other hand, because

[1] For example, the semantics of method invocation may be considered to be a nondeterministic choice between all known implementations, or to be given by the method specification associated with the static type of the target object.

the denotational semantics is compositional in terms of control constructs, it facilitates understandable proofs of interesting results.

This is a preliminary report that describes the project at its current state. The rest of this section is devoted to some design considerations.

The semantics of programs and specifications has been encoded in the PVS theorem prover and this report documents that encoding. A recent version of the PVS files can be found at `http://www.cs.stevens.edu/~naumann/corejmlPVS.tgz`.

Omitted features. The programming language does not include nested classes, threads, reflection.

Small-step semantics. A small-step semantics is of interest as further justification of the denotational semantics and more importantly for extensions to concurrency. The planned small-step semantics would use most of the semantic domains: Its configurations would involve program states as defined in the sequel and moreover the primitive commands like field update are atomic steps with essentially the same semantics as in the denotational one.

Sources. Formalization of syntax and semantics is extended and adapted from [BN05] (though terminology and naming conventions have been revised slightly) which in turn draws on [IPW01] for syntax. The formalization is also influenced by experience with a PVS encoding [Nau05].

Style of concrete syntax. The main aim of the work is a sound and robust semantical analysis of JML and more generally specifications for object-oriented programs. We follow Reynolds' admonition that programs are mathematical objects and should be typeset as such. We use a traditional Algol-like journal-paper notation, with method signatures as in CLU [LG86]; this should be transparent for a wide readership.

Expressions versus commands. Since we are considering exceptions, expressions aren't completely pure. Moreover, to avoid folding the semantics of assignment into the semantics of **new** and method call, we allow invocations and allocations in expressions, so an expression can have an effect on the heap. There are still differences between expressions and commands. A command can also have an effect on the store (i.e., assign to local variables and parameters), which an expression cannot. An expression has a result value, which a command does not.

Constructors. In this preliminary version, constructors are completely omitted. They can be treated as syntactic sugar for *ctor*() methods that are called exactly, and only, immediately following allocations as we note in Sect. 2.2.

For some purposes we will want to assume the presence of constructors, e.g., to deal with initialization of object invariants. To this end, it suffices to assume each class has exactly one constructor and it is public. In most respects a constructor is like any other method: it has parameters and its body is an arbitrary command, which may call other methods. Rules for reasoning can be simplified somewhat if constructors are not allowed to call methods that access the object under construction, which is then not fully initialized. But the language here has no such restrictions, since the aim is to use it to justify restrictions for reasoning.

Product types and list notation. A method has a list of parameters; in this document, we write, e.g., $\overline{x:T}$ for a list of variables $x_0, x_1 \ldots$ each assigned a type $x_i : T_i$. Here and throughout we use over-lines to indicate sequences, possibly empty.

It would be slightly annoying to have to work with lists in the PVS encoding, so the current PVS version adds Cartesian products to the type system and methods have a single parameter. (This is still annoying in that some results have to go by boring inductions on the structure of types.)

In detail: Where $\overline{T}$ is used in this document, the PVS formalization uses an explicit, binary Cartesian product. The expression language includes a pairing operator as well as left and right projection functions. Pairs are immutable; there is no update operator. Product types may appear as results from methods and in various other places where they are not possible in Java; there seems to be no need to prevent this.

2 Programming language syntax

2.1 Grammar

The grammar is based on some given sets of names, using the following nomenclature for typical elements:

$C \in \mathit{ClassName}$	class names
$I \in \mathit{InterfaceName}$	interface names
x, y, f	variable names (for parameters, fields, and local variables)
m	method names

It is convenient to define

$$\mathit{RefType} = \mathit{ClassName} \cup \mathit{InterfaceName}$$

We assume there are two distinguished variable names, self and res, which have special uses in the semantics (the target object and method result, respectively). We assume there are four distinguished class names Object, Thrwbl, NullDeref and ClassCast. Method declarations do not list the checked exceptions; as if every method declared "throws Thrwbl".

An interface may have only ghost fields. These are considered to be instance fields (which is explicitly marked in JML since JML also has static fields in interfaces).

Grammar of classes and methods

Class and interface declarations have the following forms:

$$\textbf{class}\ \ C\ \ \textbf{ext}\ \ C\ \ \textbf{impl}\ \ \overline{I}\ \{\ \overline{\mathit{vis}\ f:T}\quad \overline{\mathit{vis}\ \mathit{mdec}}\ \}$$
$$\textbf{interface}\ \ I\ \ \textbf{ext}\ \ \overline{I}\ \{\ \overline{\textbf{gho}\ f:T}\quad \overline{\textbf{pub}\ \mathit{msig}}\ \}$$

The remaining syntactic categories are as follows. (Bold keywords and punctuation marks including "{" and "}" are terminal symbols.)

T	::=	$C \mid I \mid \mathbf{bool} \mid \mathbf{int} \mid \mathbf{unit}$	data type
vis	::=	$\mathbf{priv} \mid \mathbf{prot} \mid \mathbf{pub} \mid \mathbf{gho}$	visibility modifier
$msig$	::=	$m(\overline{x}:\overline{T}):T$	method signature
$mdec$	::=	$\mathbf{meth}\ msig\ \{\ S\ \}$	method declaration
S	::=	$x := E \mid x.f := y$	assign to variable, to field
	$\mid$	$\mathbf{var}\ x:T\ \mathbf{in}\ S$	local variable block
	$\mid$	$S_1;\ S_2 \mid \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$	sequence, conditional
	$\mid$	$\mathbf{throw}\ x$	throw exception
	$\mid$	$\mathbf{try}\ S_1\ \mathbf{catch}(x:T)\ S_2$	try-catch
	$\mid$	$\mathbf{try}\ S_1\ \mathbf{finally}\ S_2$	try-finally
E	::=	$x \mid \mathbf{null} \mid \mathbf{true} \mid \mathbf{false} \mid 0 \mid 1 \mid 2\ldots$	variable, constant
	$\mid$	$x.f \mid x = y$	field access, equality test
	$\mid$	$x\ \mathbf{is}\ T \mid (T)\ x$	type test, cast
	$\mid$	$\mathbf{new}\ C()$	object construction
	$\mid$	$x.m(\overline{y})$	method call
	$\mid$	$\mathbf{let}\ x\ \mathbf{be}\ E_1\ \mathbf{in}\ E_2$	sequenced local binding

We omit arithmetic and boolean operators as their treatment is straightforward, essentially like equality test.

Equality test, field update, and method call are restricted to act on variables rather than general expressions; this helps simplify the semantic definitions. To avoid loss of expressiveness, we add the expression form **let** x **be** E **in** E. Its use in desugaring is discussed later. Note that the identifiers x and f both range over variable names, though for perspicuity we tend to use f in contexts that call for field names.

Variable self corresponds to *this* in Java; it represents the receiver object of the current method invocation and cannot be the target of assignment. Variable res can be assigned; it provides the result value for a method and is automatically initialized to the default value for its type. There is no **return** statement; in effect we confine attention to method bodies that declare a local variable and return it as their last statement.

A complete program is a collection of class and interface declarations. Formally, we consider a *class table* CT which is a mapping sending each class name C to its declaration $CT(C)$ and each interface name I to its declaration $CT(I)$.

We eschew the term "statement" but use identifier S for commands since C is mnemonic for class names.

To save clutter, the grammar for commands is ambiguous; in examples we use braces to delimit scope of local variables and for grouping of constituent commands in control structures.

2.2 Syntactic sugars

A simple syntactic sugar is that, in a method signature, the result type can be omitted, meaning that it is **unit**.

Constructors. The general form of object allocation includes arguments to be passed to a constructor method: **new** $C(\overline{E})$. In essence, a constructor is a method, with return type **unit**, invoked only immediately following a primitive allocation.

Thus we can apply the following desugaring transformation, which uses dummy variable u in a **let** for sequencing of evaluation:

$$\textbf{new } C(\overline{E}) \quad = \quad \textbf{let } o \textbf{ be new } C() \textbf{ in let } u \textbf{ be } o.ctor(\overline{E}) \textbf{ in } o$$

For exact correspondence with Java, require that method *ctor* is called nowhere else.

General expressions. The syntax is in something like "A-normal form" [SF93], i.e., subexpressions in various constructs are restricted to be variables.

Since expressions have effects on the heap and in the form of exceptions, the semantic definitions must thread these effects through the evaluation of subexpressions. Our semantics follows the Java Language Specification (JLS) but we have restricted the syntax of expressions, and added the form **let** x **be** E **in** E, to simplify the semantic definitions. We do not aim for the maximal possible desugaring, just some convenient simplifications to avoid mixing the propagation of exceptions with throwing of new ones. The desugaring was motivated primarily by complications that appear in machine checking the definitions.

One would expect that the grammar for expressions allows equality tests of the form $E = E$, method calls of the form $E.m(E)$, and so forth. These can all be desugared to our syntax by a simple translation. Let us write E^o for a translated expression and S^o for a translated command. For the constructs of interest, the translation is defined as follows, where in each case x, y are two fresh variables.

$$\begin{aligned}
(E_1 = E_2)^o &= \textbf{let } x \textbf{ be } E_1^o \textbf{ in let } y \textbf{ be } E_2^o \textbf{ in } x = y \\
(E_1.m(E_2))^o &= \textbf{let } x \textbf{ be } E_1^o \textbf{ in let } y \textbf{ be } E_2^o \textbf{ in } x.m(y) \\
(E.f)^o &= \textbf{let } x \textbf{ be } E^o \textbf{ in } x.f \\
((T)\ E)^o &= \textbf{let } x \textbf{ be } E^o \textbf{ in } (T)\ x \\
(E_1.f := E_2)^o &= \textbf{var } x : T_1 \textbf{ in var } y : T_2 \textbf{ in } x := E_1^o;\ y := E_2^o;\ x.f := y \\
(\textbf{throw } E_1)^o &= \textbf{var } x : T_1 \textbf{ in } x := E_1;\ \textbf{throw } x \\
(\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2)^o &= \textbf{var } x : \textbf{bool in } x := E^o;\ \textbf{if } x \textbf{ then } S_1^o \textbf{ else } S_2^o
\end{aligned}$$

We assume that E_1 (resp. E_2) in the original code is typed as T_1 (resp. T_2). The translation distributes over all other constructs.

This transformation preserves the order in which exceptions occur. For example, according to the JLS, $E_1.f := E_2$ is executed by first evaluating E_1, then E_2, and finally E_1 tested for nullity if both expressions are non-exceptional.

Desugaring try/catch/finally. The syntax has separate catch and finally, and a single catch type. These suffice to desugar the general form found in Java. The details are slightly intricate though straightforward; see for example [RH04, Sec. 2.7.2].

2.3 Informal semantics

We idealize from heap and stack bounds as well as arithmetic errors. Thus, as we consider only well-formed programs, there are no runtime errors to consider. Evaluation of an expression can have an effect on the heap, owing to **new** and method call. Evaluation of an expression can have the following outcomes:

- normal termination with a result value (and updated heap)
- exception thrown (and updated heap)
- divergence (due to a divergent method call)

Execution of a command can result in

- normal termination with updated heap and store
- exception thrown (and updated heap and store)
- divergence

Exceptions are thrown by the **throw** command, of course. In addition, NullDeref exceptions are thrown by field access, field update, and method call; ClassCast is thrown by casts.

2.4 Program typing

The typing rules are syntax directed. This makes it straightforward to make definitions and proofs by induction on typing derivations. One would expect the standard subsumption rule would be admissible; subsumptions are reflected in subtyping conditions in the typing rules, which reflect various subsumption properties that the semantics enjoys. However, it is slightly simpler for the rules to give in some sense the most precise type, e.g., the type of a variable is its declared type. This does not restrict which expressions are typable, only the types they are given. In this regard our typing rules are similar to those for Featherweight Java [IPW01].

The typing rules for commands and expressions are expressed using judgments in which the variable context is explicit, giving names and types of local variables and parameters that are in scope (namely, the method parameters, any locals in scope, and the special variables self, res). The typing rules also depend on the complete class table, as is needed to deal with recursive class declarations. This is not made explicit in the judgments, as we consider only one class table at a time; instead, various auxiliary notations are used that depend on the class table, again without explicit indication.

Recall that the syntax is defined relative to a fixed set of class names C. Also recall that a class table CT is a function that assigns a class declaration to every C in *ClassName* and an interface declaration to every I in *InterfaceName*. Suppose

$$CT(C) = \textbf{class}\ \ C\ \ \textbf{ext}\ \ D\ \ \textbf{impl}\ \ \overline{I}\ \{\overline{vis\ f:T};\ \overline{mdec}\}$$

Let *super* $C = D$ and *superinterfaces* $C = \overline{I}$. Similarly, for interfaces, suppose

$$CT(I) = \textbf{interface}\ \ I\ \ \textbf{ext}\ \ \overline{I}\ \{\ \overline{\textbf{gho}\ f:T}\quad \overline{\textbf{pub}\ msig}\ \}$$

Let *superinterfaces* $I = \overline{I}$.

Let $mdec$ be in the list $\overline{mdec}$ of method declarations, so $mdec$ has the form

$$\textbf{meth}\ m(\overline{x}:\overline{T}):T_1\ \{S\}$$

We record the type and parameter names[2] by defining $mtype(C,m) = \overline{x}:\overline{T}\to T_1$. If m is inherited in C from D (i.e., is defined in D but not declared in C) then $mtype(C,m)$ is

[2]Inclusion of the parameter names in the type of a method is unusual. It is done for convenience in the semantic definitions: arguments are passed in the form of a "store" which maps parameter names to their values. This avoids the need to create a tuple of values and then defining a store —as needed for the initial state of the method body— based on the tuple. The consequence for syntax is that method overrides are required to use the same parameter names, even though in other respects method parameters have scope only over a particular method implementation.

defined to be $mtype(D, m)$. Thus $mtype(C, m)$ is defined iff m is declared or inherited in C. Similarly for interfaces: if I extends some I' in $superinterfaces\ I$, then $mtype(I, m)$ is the same as for classes.

For declared fields we define $dfields\, C = \overline{vis\ f : T}$ and similarly $dfields\, I$ are the fields declared by interface I.

To include inherited fields we define

$$fields\, C = fields\, D \cup dfields\, C \cup (\cup I \in superinterfaces\, C \bullet fields\, I)$$

In a well-formed class table this union will be disjoint.

Define $fields\, I = dfields\, I \cup (\cup J \in superinterfaces\, I \bullet fields\, J)$. Again, in a well-formed class table this union will be disjoint.

Definition 1 *The subtype relation $\leq$ is defined inductively by*

- $C \leq D$ *if* $super\ C = D$
- $T \leq I$ *if* $I \in superinterfaces\ T$ *(in which case T is a ref type)*
- $I \leq \mathsf{Object}$ *if* $I \in dom(CT)$ *is an interface*
- $\leq$ *is reflexive and transitive*

A consequence is that, for primitive types, $T \leq U$ holds just if T is U (in a well-formed class table). Moreover $T \leq \mathsf{Object}$ for all ref types T.

Well-formed class table

A class table CT is *well-formed* provided it satisfies the following.

1. The subtype ordering $\leq$ is acyclic
2. Any ref type that appears as a field type, superclass, local variable type, cast etc. is declared in CT.
3. Field names are not shadowed; that is, for each ref type T, if $vis\ f : U$ is in $dfields\, T$, then (a) f is not in $dom(fields(super\, T))$, and (b) f is not in $dom(fields\, I)$, for all $I \in superinterfaces\, T$.
4. Every field declaration in an interface has visibility **gho** . (As is already enforced by the context-free syntax.)
5. Inherited field names have consistent types and visibilities, that is, if T has (transitive) superinterfaces I_1, I_2 with **gho** $f : T_1$ in $dfields\, I_1$ and **gho** $f : T_2$ in $dfields\, I_2$ then $T_1 = T_2$.

   In this situation, we consider that there is a single field f.
6. Method types —including parameter names— are invariant, that is, if $mtype(U, m)$ is defined and $T \leq U$ then $mtype(T, m) = mtype(U, m)$.
7. For any C, any $I \in superinterfaces\, C$, and any method signature $m(\overline{x} : \overline{T}) : T$ declared or inherited in I, there is a declared or inherited method in C with the same signature.
8. Variable exc does not occur anywhere. (It is used in specifications to refer to the exceptional result, if any, and it is used in the semantics.)

9. For any C, every method declaration $m(\overline{x}:\overline{T}):T\ \{S\}$ in $CT(C)$ is typable in the sense that $\Gamma \vdash S$ where $\Gamma = [\mathsf{self}:C, \mathsf{res}:T, \overline{x}:\overline{T}]$. Rules that define $\Gamma \vdash S$ appear just below. The rules refer to partial function *mtype*, which is well defined given the preceding conditions.

Here and throughout the paper we let Γ range over typing contexts, i.e., finite maps from variables to types.

Because *ClassName* and *InterfaceName* are considered to be the set of all declared class and interface names, we have that in a well-formed class table every ref type that appears as a field type, superclass, local variable type, cast etc. is declared in CT.

Ghost fields are auxiliary state used for reasoning. To streamline the definitions we refrain from distinguishing them from ordinary fields wherever possible. Some distinction is necessary, however, in order to ensure that they have no influence on the value of non-ghost fields or variables. Owing to the desugared syntax, it sufficies to require that no ghost field appear in the expression part of any variable assignment $x := E$.

In fact there is no need for these restrictions in the syntax and semantics, since we do not use JML's syntax to distinguish between ghost and ordinary assignments. So the restrictions are not imposed until we come to specifications and reasoning.

The typing rules are syntax-directed, to cater for definitions by induction on syntax.

Typing rules for expressions

$$\Gamma \vdash x : \Gamma x \qquad \Gamma \vdash \mathbf{true} : \mathbf{bool} \qquad \Gamma \vdash \mathbf{false} : \mathbf{bool}$$

$$\Gamma \vdash 0 : \mathbf{int} \qquad \Gamma \vdash 1 : \mathbf{int} \qquad \ldots \qquad \frac{T \in \mathit{RefType}}{\Gamma \vdash \mathbf{null} : T}$$

$$\frac{\Gamma \vdash x : T_1 \qquad \Gamma \vdash y : T_2}{\Gamma \vdash x = y : \mathbf{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{new}\ C() : C}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : U \qquad (\mathit{vis}\ f : T) \in \mathit{dfields}\, V \\ U \leq V \qquad (\mathit{vis} = \mathbf{priv} \Rightarrow \Gamma\,\mathsf{self} = V) \qquad (\mathit{vis} = \mathbf{prot} \Rightarrow \Gamma\,\mathsf{self} \leq V)\end{array}}{\Gamma \vdash x.f : T}$$

$$\frac{\Gamma \vdash x : T \qquad U \leq T \qquad T \in \mathit{RefType}}{\Gamma \vdash (U)\ x : U} \qquad \frac{\Gamma \vdash x : T \qquad U \leq T \qquad T \in \mathit{RefType}}{\Gamma \vdash x\ \mathbf{is}\ U : \mathbf{bool}}$$

$$\frac{\Gamma \vdash x : T \qquad T \in \mathit{RefType} \qquad \mathit{mtype}(T, m) = \overline{z}:\overline{T} \to U \qquad \Gamma \vdash \overline{y} : \overline{V} \qquad \overline{V} \leq \overline{T}}{\Gamma \vdash x.m(\overline{y}) : U}$$

$$\frac{\Gamma \vdash E : T \qquad [\Gamma, x : T] \vdash E1 : U}{\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ E\ \mathbf{in}\ E1 : U}$$

Typing rules for commands

$$\frac{\Gamma \vdash E : T \qquad T \leq \Gamma\, x \qquad x \neq \mathsf{self}}{\Gamma \vdash x := E}$$

$$\frac{U \leq V \qquad \Gamma\, y \leq T \qquad \begin{array}{c}\Gamma \vdash x : U \qquad (vis\ f : T) \in \mathit{dfields}\, V \\ (vis = \mathbf{priv} \Rightarrow \Gamma\, \mathsf{self} = V) \qquad (vis = \mathbf{prot} \Rightarrow \Gamma\, \mathsf{self} \leq V)\end{array}}{\Gamma \vdash x.f := y}$$

$$\frac{\Gamma \vdash x : \mathbf{bool} \qquad \Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2} \qquad \frac{[\Gamma, x : T] \vdash S}{\Gamma \vdash \mathbf{var}\ x : T\ \mathbf{in}\ S}$$

$$\frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash S_1;\ S_2} \qquad \frac{\Gamma \vdash x : T \qquad T \leq \mathsf{Thrwbl}}{\Gamma \vdash \mathbf{throw}\ x}$$

$$\frac{\Gamma \vdash S_1 \qquad \Gamma, x : T \vdash S_2 \qquad T \leq \mathsf{Thrwbl}}{\Gamma \vdash \mathbf{try}\ S_1\ \mathbf{catch}(x : T)\ S_2} \qquad \frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash \mathbf{try}\ S_1\ \mathbf{finally}\ S_2}$$

2.5 PVS formalization

The context free rules for data types are in theory DTY and the rules for expressions and commands are in theory LANG. As discussed in Sect. 1, the data types include cartesian products. Because try-finally can be desugared using try-catch and assignments to temporary variables, it is omitted.

Typing rules for expressions and commands are in theory TYPING. Theory CLASSTABLESIG declares the signature of a class table, which consists of the names of declared classes and interfaces, the declared subclassing relations, the auxiliary functions *mtype*, *fields*, etc. Most of the well-formedness conditions for a class table are also imposed in CLASSTABLESIG; the formalization of subtyping is somewhat different from that above, but the effect is the same. Theory WELLFORMEDCT describes declaration and type correctness for method implementations.

The syntax omits visibility declarations (Java "access modifiers") and thus for inheritance the typing rules are based on function *fields* rather than *dfields*.

To cater for methodologies like Boogie that require update to ghost fields, even update of ghost fields for unboundedly many objects, the PVS syntax includes primitive command names and the semantics is given with respect to an arbitrary interpretation of those names as state transformers. This lets us avoid the nondeterminacy required to interpret general "assume" statements while still modeling reasoning based on ghost state.

A very minor detail: Fields are not allowed to have type **unit** (which is useless anyway) because we chose to encode *fields* as a total function $RefType \times FieldName \to Types$ with $\mathit{dfields}(T, f) = \mathbf{unit}$ representing that f is not a declared field of T.

3 Program semantics

We assume a given set *Ref* of *references* —abstract addresses. A *ref context* is a finite partial function ρ that maps references to class names. The idea is that if $o \in \mathit{dom}\, \rho$

then o is allocated and moreover o points to an object of type $\rho\, o$. We define the set $\mathit{RefCtx} = \mathit{Ref} \rightharpoonup \mathit{ClassName}$, where $\rightharpoonup$ denotes finite partial functions. Note that for ρ and ρ' in RefCtx, we can write $\rho \subseteq \rho'$ to express that the domain of ρ' includes at least the objects in ρ and for objects allocated in ρ the types are the same in ρ'.

The term "location" is used for possible assignment targets, which are pairs (o, f) of a reference and a field name.

Semantic domains. Semantic domains are the sets of possible meanings for various kinds of phrases such as data types, expressions, and method bodies. The definitions capture important invariants about the semantics:

- there are no dangling references;
- the value of a field or variable of a given static type is a value of that type;
- values of type T, for ref type T, are references to objects of all classes C, $C \leq T$.

Not all invariants of a Java-like language are modeled, e.g., self is not null and cannot be updated but our results do not depend on this.

Most of the semantic domains are defined in terms of a given ref context. For example, $Val(T, \rho)$ is the set of values of type T in a state where ρ is the ref context.[3] In case T is a primitive type, the definition of $Val(T, \rho)$ is independent from ρ. But if T is a class type then $Val(T, \rho)$ is some set containing *null* and some references —all the allocated objects of some type U such that $U \leq T$.

Here is a guide to the domains. Those marked with $*$ are just special cases of the others as explained below. The PVS version currently uses some different names, as noted, with R in place of ρ and V in place of Γ.

domain	description	meta-vars.	PVS notation
$Val(T, \rho)$	values of type T	o, v	ValOfType(T,R)
$Store(\Gamma, \rho)$	stores for Γ	r	Store(V,R)
$*\ FieldRcrd(C, \rho)$	fields of C-objects		ObjState(C,R)
$Heap(\rho)$	heaps	h	Heap(R)
$State(\Gamma)$	states for Γ	s, t	ProgState(V)
$STrans(\Gamma_1, \Gamma_2)$	state transformers	σ, τ	STrans(V1,V2)
$*\ SemExpr(\Gamma, T)$	semantic expression		SemExpr(V,T)
$*\ SemCommand(\Gamma)$	semantic command		SemCmd(V)
$*\ SemMeth(C, m)$	semantics of method m in C		SemMeth(C,m)
$Menv$	method environment	μ	MethEnv

For data types T the domain of values is defined is by cases on T:

$$\begin{array}{lcl} Val(\mathbf{bool}, \rho) & = & \{\mathit{true}, \mathit{false}\} \\ Val(\mathbf{int}, \rho) & = & \mathbb{Z} \\ Val(\mathbf{unit}, \rho) & = & \{\mathit{it}\} \\ Val(C, \rho) & = & \{\mathit{null}\} \cup \{o \mid o \in \mathit{dom}\, \rho \wedge \rho\, o \leq C\} \\ Val(I, \rho) & = & \{o \mid \exists C \bullet C \leq I \wedge o \in Val(C, \rho)\} \end{array}$$

Note that, for any ρ, we have

$$T \leq U \text{ implies } Val(T, \rho) \subseteq Val(U, \rho) \tag{1}$$

[3] Banerjee and Naumann's papers use notation like $[\![T]\!]$ for domains, and define syntactic categories θ so every domain has a name of the form $[\![\theta]\!]$.

The next definitions involve dependent function spaces or dependent pairs, for which we use the following.

Notation for dependent types in metatheory

Suppose X is a set and for every $x \in X$ a set Z_x is given. Then the dependent product $(y : X) \times Z_y$ is the set of pairs (x, z) such that $x \in X$ and $z \in Z_x$. The dependent function space $(y : X) \to Z_y$ is the set of functions f from X to $\cup_{x \in X} Z_x$ such that $f\,x \in Z_x$ for all $x \in X$. Note that y is a bound variable in these notations.[4]

The first use is in the definition

$$Store(\Gamma, \rho) = (x : dom\,\Gamma) \to Val(\Gamma\,x, \rho)$$

What this means is that $Store(\Gamma, \rho)$ is a set of functions; and for any r in $Store(\Gamma, \rho)$, the domain of r is $dom\,\Gamma$ and $r\,x$ is an element of $Val(\Gamma\,x, \rho)$ for each $x \in dom\,\Gamma$.

The primary use for a variable typing Γ is for the variables that are in scope in some method body —its locals, parameters, and self. A state for Γ is thus local in a sense, although it contains the global heap. To adapt our results to a language with global variables, i.e., static class fields, these could be added as a separate component of the state or included in every store of interest.

Next we build up to program states. Define $obcontext\,C$ to be the variable context obtained by removing visibility markers from $fields\,C$.

$$\begin{array}{lcl} FieldRcrd(C, \rho) & = & Store(obcontext\,C, \rho) \\ Heap(\rho) & = & (o : dom\,\rho) \to FieldRcrd(\rho\,o, \rho) \\ State(\Gamma) & = & (\rho : RefCtx) \times Heap(\rho) \times Store(\Gamma, \rho) \end{array}$$

A heap h is map sending each allocated reference o to a record, $h\,o$, of the object's current field values and $h\,o\,f$ is the value of field f.

The most important domain is state transformers:

$$STrans(\Gamma, \Gamma') = (s : State(\Gamma)) \to \{\bot\} \cup \{s' | s' \in State(\Gamma') \wedge extState(s, s')\}$$

Relation $extState$ is defined to say that one state's ref context extends the other's:[5]

$$extState((\rho, h, r), (\rho', h', r')) \iff \rho \subseteq \rho'$$

Elements of $STrans(\Gamma_1, \Gamma_2)$ are functions that map a state in $State(\Gamma_1)$ to either $\bot$ or a state in $State(\Gamma_2)$. The domain of state transformers subsumes meanings for methods, expressions and commands:

$$\begin{array}{lcl} SemExpr(\Gamma, T) & = & STrans(\Gamma,\ [\mathsf{res} : T, \mathsf{exc} : \mathsf{Thrwbl}]) \\ SemCommand(\Gamma) & = & STrans(\Gamma,\ [\Gamma,\ \mathsf{exc} : \mathsf{Thrwbl}]) \\ SemMeth(T, m) & = & STrans([\mathsf{self} : T, \overline{x} : \overline{T}],\ [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thrwbl}]) \\ & & \text{where } mtype(T, m) = \overline{x} : \overline{T} \to U \end{array}$$

We sometimes use the term *state transformer type* and notation $\Gamma \rightsquigarrow \Gamma'$ in connection with elements of $STrans(\Gamma, \Gamma')$, for any Γ, Γ'. Thus a command in context

[4] More standard notations are $\Sigma\,y : X.Z_y$ for $(y : X) \times Z_y$ and $\Pi\,y : X.Z_y$ for $(y : X) \to Z_y$. Ours are similar to the PVS prover's.

[5] Since ρ and ρ' are partial functions which we treat as sets of pairs, $\rho \subseteq \rho'$ says that ρ' has at least the domain of ρ and they agree on their common domain.

Γ denotes a state transformer of type $\Gamma \rightsquigarrow [\Gamma,\ \mathsf{exc} : \mathsf{Thrwbl}]$. An expression of type T in context Γ denotes a state transformer of type $\Gamma \rightsquigarrow [\mathsf{res} : T, \mathsf{exc} : \mathsf{Thrwbl}]$. And a method declaration with $\mathit{mtype}(T, m) = \overline{x} : \overline{T} \to U$ denotes a state transformer of type $[\mathsf{self} : T, \overline{x} : \overline{T}] \rightsquigarrow [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thrwbl}]$.

A *method environment* is defined to be a table of meanings for all methods in all classes:

$$\mathit{Menv} = (C : \mathit{ClassName}) \times (m : \mathit{Meths}\, C) \to \mathit{SemMeth}(C, m)$$

A complete program CT denotes a method environment.

The idea is that a method environment μ is defined for pairs (C, m) where C is a class with method m and moreover $\mu(C, m)$ is a state transformer suitable to be the meaning of a method of type $\mathit{mtype}(C, m)$. In case m is inherited in C from B, $\mu(C, m)$ will be the restriction of $\mu(B, m)$ to receiver objects of type C.

3.1 Semantics of expressions and commands

Some notations

For typing contexts we write $x : T$ to express that x is mapped to T, and $[\Gamma, x : T]$ indicates the extension of Γ with a binding for x which must not occur in the domain of Γ. In the semantics we use a similar notation, e.g., $[r,\ \mathsf{exc} : v]$ extends store r to map exc to v, where exc is not in $\mathit{dom}\, r$. We write $[r \mid \mathsf{exc} : v]$ to update r in case exc is in $\mathit{dom}\, r$. Note that extension of a map is distinguished from update by use of comma versus $|$.

For field updates, we write $[h \mid o.f : v]$ to abbreviate the nested update $[h \mid o : [h\, o \mid f : v]]$.

Application binds most tightly, e.g., $[r,\ \mathsf{exc} : r_1\, \mathsf{exc}]$ updates r to map exc to the value of $r_1\, \mathsf{exc}$.

To remove an element from the domain of a function we use the minus sign, e.g., if r is a store then $r - \mathsf{exc}$ is the same store but with exc removed from its domain.

Let-expressions in the metalanguage are $\bot$-strict, i.e., if α is $\bot$ then $\mathsf{let}\ x = \alpha\ \mathsf{in}\ E$ is $\bot$ and otherwise it is obtained from E under the binding of x.

To define $\mathit{defaultFieldRcrd}\, C$, which provides the initial state of an object of type C, we define *default* for each type: $\mathit{default}\,\mathbf{int} = 0$, $\mathit{default}\,\mathbf{bool} = \mathit{false}$, and $\mathit{default}\, T = \mathit{null}$ for reference type T. Then $\mathit{defaultFieldRcrd}\, C$ is just the mapping of $\mathit{fields}\, C$ to the default values for their types, so that $\mathit{defaultFieldRcrd}\, C$ is in $\mathit{FieldRcrd}(C, \rho)$ for all ρ.

To streamline the semantics of expressions, we define a helping function to create exceptional result states. Given ref context ρ, heap $h \in \mathit{Heap}(\rho)$, classname $C \leq \mathsf{Thrwbl}$, and any type T we define $\mathit{except}(\rho, h, T, C)$ to be an element of $\mathit{State}([\mathsf{res} : T, \mathsf{exc} : \mathsf{Thrwbl}])$ as follows.

$$\begin{array}{lcl} \mathit{except}(\rho, h, T, C) & = & \mathsf{let}\ o = \mathit{fresh}\, \rho\ \mathsf{in} \\ & & \mathsf{let}\ \rho_1 = [\rho,\ o : C]\ \mathsf{in} \\ & & \mathsf{let}\ h_1 = [h,\ o : \mathit{defaultFieldRcrd}\, C]\ \mathsf{in}\ (\rho_1, h_1, [\mathsf{res} : \mathit{default}\, T, \mathsf{exc} : o]) \end{array}$$

This is similar to the semantics of **new** $C()$, but the new object is assigned to exc rather than to res.

A similar helping function is used in the semantics of commands. Given (ρ, h, r) in $\mathit{State}(\Gamma)$ and classname $C \leq \mathsf{Thrwbl}$ we define $\mathit{except}(\rho, h, r, C)$ to be an element of

$State([\Gamma,\ \mathsf{exc}:\mathsf{Thrwbl}])$ as follows.

$$\begin{array}{lcl} except(\rho,h,r,C) & = & \mathsf{let}\ o = fresh\,\rho\ \mathsf{in} \\ & & \mathsf{let}\ \rho_1 = [\rho,\ o:C]\ \mathsf{in} \\ & & \mathsf{let}\ h_1 = [h,\ o:defaultFieldRcrd\,C]\ \mathsf{in}\ (\rho_1, h_1, [r,\ \mathsf{exc}:o]) \end{array}$$

With these ingredients we are finally ready to define the semantics of expressions and commands. In brief, for a typable expression $\Gamma \vdash E:T$, the semantics $[\![\Gamma \vdash E:T]\!]$ is an element of

$$Menv \to SemExpr(\Gamma, T)$$

That is, the semantics $[\![\Gamma \vdash E:T]\!]$ gets applied to a method environment μ to yield a state transformer $[\![\Gamma \vdash E:T]\!](\mu)$ that in turn is applied to a state (ρ,h,r) in $State(\Gamma)$. Finally,$[\![\Gamma \vdash E:T]\!](\mu)(\rho,h,r)$ yields either $\perp$ or an element of $State([\mathsf{res}:T, \mathsf{exc}:\mathsf{Thrwbl}])$.

The definition is by recursion on the structure of E. In case E has subexpressions, the definition is formulated using nomenclature from the corresponding typing rule.[6]

Semantics of expressions

[6]To be very precise, the recursion is on typing derivations. But they are unique except for **null** and the semantics for **null** does is the same for any typing derivation. So it is not worth the trouble to formulate and prove a coherence theorem.

$$[\![\Gamma \vdash x : T]\!](\mu)(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : r\,x, \mathsf{exc} : \mathit{null}]$$
$$[\![\Gamma \vdash \mathbf{true} : \mathbf{bool}]\!](\mu)(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : \mathit{true}, \mathsf{exc} : \mathit{null}])$$
$$[\![\Gamma \vdash \mathbf{false} : \mathbf{bool}]\!](\mu)(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : \mathit{false}, \mathsf{exc} : \mathit{null}])$$
$$[\![\Gamma \vdash 0 : \mathbf{int}]\!](\mu)(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : 0, \mathsf{exc} : \mathit{null}])$$
$$[\![\Gamma \vdash 1 : \mathbf{int}]\!](\mu)(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : 1, \mathsf{exc} : \mathit{null}])$$
$$\ldots$$
$$[\![\Gamma \vdash \mathbf{null} : T]\!](\mu)(\rho, h, r) \;=\; (\rho, h, [\mathsf{res} : \mathit{null}, \mathsf{exc} : \mathit{null}])$$
$$[\![\Gamma \vdash x = y : \mathbf{bool}]\!](\mu)(\rho, h, r) \;=$$
$$\quad \mathsf{let}\ v = (\mathsf{if}\ (r\,x = r\,y)\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ \mathit{false})\ \mathsf{in}\ (\rho, h, [\mathsf{res} : v, \mathsf{exc} : \mathit{null}])$$
$$[\![\Gamma \vdash \mathbf{new}\ C() : C]\!](\mu)(\rho, h, r) \;=$$
$$\quad \mathsf{let}\ o = \mathit{fresh}\,\rho\ \mathsf{in}\ \mathsf{let}\ \rho_0 = [\rho,\ o : C]\ \mathsf{in}\ \mathsf{let}\ h_0 = [h,\ o : \mathit{defaultFieldRcrd}\,C]\ \mathsf{in}$$
$$\quad (\rho_0, h_0, [\mathsf{res} : o, \mathsf{exc} : \mathit{null}])$$
$$[\![\Gamma \vdash x.f : T]\!](\mu)(\rho, h, r) \;=$$
$$\quad \mathsf{if}\ r\,x \neq \mathit{null}\ \mathsf{then}\ (\rho, h, [\mathsf{res} : h(r\,x).f, \mathsf{exc} : \mathit{null}])\ \mathsf{else}\ \mathit{except}(\rho, h, T, \mathsf{NullDeref})$$
$$[\![\Gamma \vdash (U)\ x : U]\!](\mu)(\rho, h, r) \;=$$
$$\quad \mathsf{if}\ r\,x = \mathit{null} \vee \rho(r\,x) \leq U\ \mathsf{then}\ (\rho, h, [\mathsf{res} : r\,x, \mathsf{exc} : \mathit{null}])\ \mathsf{else}\ \mathit{except}(\rho, h, U, \mathsf{ClassCast})$$
$$[\![\Gamma \vdash x\ \mathbf{is}\ U : \mathbf{bool}]\!](\mu)(\rho, h, r) \;=$$
$$\quad \mathsf{let}\ v = (\mathsf{if}\ r\,x \neq \mathit{null} \wedge \rho(r\,x) \leq U\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ \mathit{false})\ \mathsf{in}\ (\rho, h, [\mathsf{res} : v, \mathsf{exc} : \mathit{null}])$$
$$[\![\Gamma \vdash \mathbf{let}\ x\ \mathbf{be}\ E\ \mathbf{in}\ E1 : U]\!](\mu)(\rho, h, r) \;=$$
$$\quad \mathsf{let}\ (\rho_0, h_0, r_0) = [\![\Gamma \vdash E : T]\!](\mu)(\rho, h, r)\ \mathsf{in}$$
$$\quad \mathsf{if}\ r_0\,\mathsf{exc} \neq \mathit{null}\ \mathsf{then}\ (\rho_0, h_0, [\mathsf{res} : \mathit{default}\,U, \mathsf{exc} : r_0\,\mathsf{exc}])$$
$$\quad \mathsf{else}\ \mathsf{let}\ r_1 = [r,\ x : r_0\,\mathsf{res}]\ \mathsf{in}\ [\![\Gamma \vdash E1 : U]\!](\mu)(\rho_0, h_0, r_1)$$
$$[\![\Gamma \vdash x.m(\overline{y}) : U]\!](\mu)(\rho, h, r) \;=$$
$$\quad \mathsf{if}\ r\,x = \mathit{null}\ \mathsf{then}\ \mathit{except}(\rho, h, U, \mathsf{NullDeref})$$
$$\quad \mathsf{else}\ \mathsf{let}\ \overline{z} : \overline{T} \to U = \mathit{mtype}(T, m)\ \mathsf{in}\ \mathsf{let}\ r_1 = [\mathsf{self} : r\,x, \overline{z} : r\,\overline{y}]\ \mathsf{in}\ \mu(\rho(r\,x), m)(\rho, h, r_1)$$

For cast and type test, the typing rule ensures that type T (and hence type U) is a class or interface; this justifies application of ρ in the semantics.

For method call, the target object is $r\,x$, so $\rho(r\,x)$ is the dynamic type of the object; thus to look up in method environment μ the meaning of the dynamically dispatched method we write $\mu(\rho(r\,x), m)$. Since the argument expressions $\overline{y}$ are variables we can write $r\,\overline{y}$ for their values.

The semantics of commands is also defined by recursion on structure and using nomenclature from the typing rules. For any derivable typing $\Gamma \vdash S$ and method environment μ, we define $[\![\Gamma \vdash S]\!](\mu)$ so it is an element of $\mathit{SemCommand}(\Gamma)$.

Semantics of commands

$[\![\Gamma \vdash x := E]\!](\mu)(\rho,h,r) \;=$
  let $(\rho_1,h_1,r_1) = [\![\Gamma \vdash E : T]\!](\mu)(\rho,h,r)$ in
  if $r_1\,\mathsf{exc} = \mathit{null}$ then $(\rho_1, h_1, [[r \mid\ x : r_1\,\mathsf{res}],\ \mathsf{exc} : \mathit{null}])$ else $(\rho_1, h_1, [r,\ \mathsf{exc} : r_1\,\mathsf{exc}])$
$[\![\Gamma \vdash x.f := y]\!](\mu)(\rho,h,r) \;=$
  if $r\,x \neq \mathit{null}$ then $(\rho, [h,\ r\,x.f : r\,y], [r,\ \mathsf{exc} : \mathit{null}])$ else $\mathit{except}(\rho,h,r,\mathsf{NullDeref})$
$[\![\Gamma \vdash \textbf{if } x \textbf{ then } S_1 \textbf{ else } S_2]\!](\mu)(\rho,h,r) \;=$
  if $r\,x = \mathit{true}$ then $[\![\Gamma \vdash S_1]\!](\mu)(\rho,h,r)$ else $[\![\Gamma \vdash S_2]\!](\mu)(\rho,h,r)$
$[\![\Gamma \vdash \textbf{var } x : T \textbf{ in } S]\!](\,) =$
  let $(\rho_1,h_1,r_1) = [\![\Gamma, x : T \vdash S]\!](\mu)(\rho,h,[r,\ x : \mathit{default}T])$ in $(\rho_1,h_1,r_1 - x)$
$[\![\Gamma \vdash S_1;\ S_2]\!](\mu)(\rho,h,r) \;=$
  let $(\rho_1,h_1,r_1) = [\![\Gamma \vdash S_1]\!](\mu)(\rho,h,r)$ in
  if $r_1\,\mathsf{exc} = \mathit{null}$ then $[\![\Gamma \vdash S_2]\!](\mu)(\rho_1,h_1,r_1 - \mathsf{exc})$ else (ρ_1,h_1,r_1)
$[\![\Gamma \vdash \textbf{throw } x]\!](\mu)(\rho,h,r) \;=$
  if $r\,x \neq \mathit{null}$ then $(\rho,h,[r,\ \mathsf{exc} : r\,x])$ else $\mathit{except}(\rho,h,r,\mathsf{NullDeref})$
$[\![\Gamma \vdash \textbf{try } S_1 \textbf{ catch}(x:T)\ S_2\ ]\!](\mu)(\rho,h,r) \;=$
  let $(\rho_1,h_1,r_1) = [\![\Gamma \vdash S_1]\!](\mu)(\rho,h,r)$ in
  if $r_1\,\mathsf{exc} = \mathit{null} \vee \rho(r_1\,\mathsf{exc}) \not\leq T$ then (ρ_1,h_1,r_1)
  else let $r_3 = [r_1,\ x : r_1\,\mathsf{res}] - \mathsf{exc}$ in
    let $(\rho_2,h_2,r_2) = [\![\Gamma, x : T \vdash S_2]\!](\mu)(\rho_1,h_1,r_3)$ in $(\rho_2,h_2,r_2 - x)$
$[\![\Gamma \vdash \textbf{try } S_1 \textbf{ finally } S_2\ ]\!](\mu)(\rho,h,r) \;=$
  let $(\rho_1,h_1,r_1) = [\![\Gamma \vdash S_1]\!](\mu)(\rho,h,r)$ in
  let $v = r_1\,\mathsf{exc}$ in
  let $(\rho_2,h_2,r_2) = [\![\Gamma \vdash S_2]\!](\mu)(\rho_1,h_1,r_1 - \mathsf{exc})$ in
  if $r_2\,\mathsf{exc} = \mathit{null}$ then $(\rho_2,h_2,[r_2,\ \mathsf{exc} : v])$ else (ρ_2,h_2,r_2)

In the semantics of field update we write $[h \mid\ r\,x.f : r\,y]$ to abbreviate the nested update $[h \mid\ r\,x : [h(r\,x) \mid\ f : r\,y]]$ that changes only field f of object $r\,x$.

The definitions are somewhat intricate in part owing to propagation of exceptions. For example, the clause for $x := E$ checks whether E yielded an exception. If not, the program store r is updated to map x to the result value of E, i.e., $r_1\,\mathsf{res}$, and extended to map exc to null. In case of an exception, r is extended to map exc to the exception reference, $r_1\,\mathsf{exc}$, from the expression semantics.

3.2 Domain Orderings

A class table denotes a method environment obtained as the least upper bound of a chain of approximations. For this purpose each semantic domain is given a partial ordering. The sets $Val(T,\rho)$, $Store(\Gamma,\rho)$, $Heap(\rho)$, and $State(\Gamma)$ are ordered by equality. For σ,τ in $STrans(\Gamma_1,\Gamma_2)$ define $\sigma \leq \tau$ iff for all states s, if $\sigma s \neq \bot$ then $\sigma s = \tau s$. Finally, for μ,μ' in $Menv$, define $\mu \leq \mu'$ iff $\mu(C,m) \leq \mu'(C,m)$ for all C,m.

Note that $STrans(\Gamma_1, \Gamma_2)$ carries the usual pointwise order on functions, where the domain $State(\Gamma_1)$ is ordered discretely but the range, $lift(State(\Gamma_2))$, has the flat order $s \leq t$ iff $s = \bot \vee s = t$.

Each of these sets is a partial order, complete in the sense that every ascending chain has a least upper bound. The everywhere-$\bot$ function is the least element in $STrans(\Gamma_1, \Gamma_2)$. The least method environment, μ_0, is defined so that $\mu_0(C, m)$ is the everywhere-$\bot$ function for all C, m.

The least upper bound of method environments has a simple characterization.

Lemma 2 (characterization of least upper bounds for method environments) *Suppose μ is a function from naturals to method environments such that $i \leq j \Rightarrow \mu_i \leq \mu_j$. Then for any C, m and state s there is some j such that $(lub\, \mu)(C, m)(s) = \mu_k(C, m)(s)$ for all $k \geq j$.*

This follows from a similar property for ascending chains of state transformers (of a fixed type). Using these characterizations it is straightforward to show that the semantics of commands and expressions is continuous in the method environment and in the semantics of constituent commands and expressions.[7]

3.3 Semantics of method declarations and class table

To define the semantics of a class table we need to obtain the semantics of each method declaration from the semantics of its body. Suppose class C has declaration

$$\textbf{meth}\ m(\overline{x} : \overline{T}) : U\ \{S\}$$

Its meaning is a state transformer of type $[\mathsf{self} : C, \overline{x} : \overline{T}] \rightsquigarrow [\mathsf{res} : U, \mathsf{exc} : \mathsf{Thrwbl}]$, and the meaning of S is a state transformer of type $\Gamma \rightsquigarrow [\Gamma, \mathsf{exc} : \mathsf{Thrwbl}]$ where $\Gamma = [\mathsf{self} : C, \overline{x} : \overline{T}, \mathsf{res} : U]$. Basically, the method meaning is obtained from the denotation of S by initializing res and then projecting out of the final state.

Formally, suppose μ is a method environment. Then $methSem(C, m)(\mu)$ is the element of $SemMeth(C, m)$ defined by

$$\begin{aligned} methSem(C, m)(\mu)(\rho, h, r) = \ & \mathsf{let}\ r_0 = [r,\ \mathsf{res} : defaultT]\ \mathsf{in} \\ & \mathsf{let}\ (\rho_1, h_1, r_1) = [\![\overline{x} : \overline{T}, \mathsf{self} : C, \mathsf{res} : T \vdash S]\!](\mu)(\rho, h, r_0)\ \mathsf{in} \\ & (\rho_1, h_1, [\mathsf{res} : (r_1\ \mathsf{res}), \mathsf{exc} : (r_1,\ \mathsf{exc})]) \end{aligned}$$

Function $methSem$ is used only in the following.

Definition 3 (semantics of complete program) *The semantics of a class table CT is the least upper bound of the ascending chain μ defined as follows.*

$$\begin{aligned} & \mu_0(C, m) = \lambda s \bullet \bot \\ & \mu_{j+1}(C, m) = methSem(C, m)(\mu_j) && \text{if } m \text{ is declared as } M \text{ in } C \\ & \mu_{j+1}(C, m) = \mu_{j+1}(B, m) && \text{if } m \text{ is inherited from } B \text{ in } C \end{aligned}$$

To be very precise for an inherited method, if $mtype(C, m) = \overline{x} : \overline{T} \rightarrow T$ then $\mu_{j+1}(C, m)$ should apply to stores for $\overline{x} : \overline{T}, \mathsf{self} : C$ whereas $\mu_{j+1}(B, m)$ applies to stores for $\overline{x} : \overline{T}, \mathsf{self} : B$. But the latter contains the former, owing to Equation (1). This does not obtrude in the sequel.

[7] In fact we don't need continuity to prove well definedness of the semantics because the program semantics is at the level of method environments. To prove that the approximation chain in Def. 3 is ascending, we do need that command semantics is monotonic in the method environment. For details, see the PVS formalization.

3.4 PVS Formalization

The semantic domains are defined in theory SEMANTICDOMAINS. Theory STATETRANSFORMERS defines some basic state transformers for use in the semantics. The semantics of expressions and of commands is in theory EXPSEMANTICS and COMSEMANTICS. Theory SEMCT gives the semantics of a class table; it relies on theory ORDERDOMAINS which defines partial orderings on the domains and on COMSEMANTICSPROPS which has monotonicity results needed to prove that the semantics is well defined.

For semantics of a complete program, in the case of m inherited from B (see Def. 3), the PVS definition has the form $\mu_{j+1}(C,m) = \mathit{restrict}(\mu_{j+1}(B,m))$ using an explicit restriction of function $\mu_{j+1}(B,m)$ to the smaller domain where self is an object of type $\leq C$.

4 Specifications

From now on it is assumed that CT is some well-formed class table. This section formalizes method specifications and satisfaction by state transformers (in the sense of total correctness). On this basis we define specification tables and satisfaction for them as well as the induced refinement relation between specifications. In this version of the formalization, specifications are treated semantically.

In practice, most specifications are written using *two-state* postconditions over program state, with special syntax like "old(x)" to refer to the initial state. A specification of this kind can be desugared into one where the pre- and post-conditions are one-state predicates, using auxiliary variables universally quantified over the Hoare triple [Apt81] as is made explicit in JML's notation for auxiliary variables. Care must be taken in reasoning with such specifications to avoid soundness pitfalls (cf. [AdB90]). For our purposes it is convenient to distinguish auxiliaries from ordinary program variables by considering indexed families of pre/post predicates on program state.[8]

Definition 4 (state transformer specification) *A* simple specification of type $\Gamma \rightsquigarrow \Gamma'$ *is a pair* $(pre, post)$ *such that pre is a subset of* $State(\Gamma)$ *and post is a subset of* $State(\Gamma')$

A general specification of type $\Gamma \rightsquigarrow \Gamma'$ *is a triple* $(J, pre, post)$ *consisting of a set* J *and indexed families* $pre \in J \to \mathbb{P}(State(\Gamma))$ *and* $post \in J \to \mathbb{P}(State(\Gamma'))$.

A method specification of type (T,m) *is one of type* $[\mathsf{self}\colon T, \overline{x}\colon \overline{T}] \rightsquigarrow [\mathsf{res}\colon U, \mathsf{exc}\colon \mathsf{Thrwbl}]$ *where* $mtype(m,T) = \overline{x}\colon \overline{T} \to U$. *And a* Γ-specification *is one of type* $\Gamma \rightsquigarrow [\Gamma, \mathsf{exc}\colon \mathsf{Thrwbl}]$.

Unqualified, "specification" means general specification unless the context makes it obvious that simple specifications are considered.

Definition 5 (interpretation of two-state postconditions) *Given a two-state postcondition* $Q \subseteq State(\Gamma) \times State(\Gamma')$ *and* $P \subseteq State(\Gamma)$ *one obtains a general specification* $(J, pre, post)$ *by taking* J *to be* $State(\Gamma)$ *and defining* pre_s *and* $post_s$, *for* $s \in State(\Gamma)$, *by* $pre_s = \{t \mid t = s \wedge s \in P\}$ *and* $post_s = \{t \mid (s,t) \in Q\}$.

Definition 6 (satisfaction by state transformer, $\models$) *Let* $(pre, post)$ *be a specification of type* $\Gamma \rightsquigarrow \Gamma'$ *and* σ *be in* $STrans(\Gamma, \Gamma')$. *Then* σ satisfies $(pre, post)$, *written* $\sigma \models (pre, post)$, *iff*

$$\forall s \bullet s \in pre \;\Rightarrow\; \sigma\, s \in post$$

For general $(J, pre, post)$, *let* $\sigma \models (J, pre, post)$ *iff* $\sigma \models (pre_i, post_i)$ *for all* $i \in J$.

[8] This is also convenient for the generalization to relational logics [Ben04, Yan04] for programs acting on the heap, see [Nau06].

A *specification table*, ST, contains a method specification $ST(T, m)$ of type $mtype(T, m)$ for each ref type T and each $m \in Meths\, T$. It models what might be called the "effective specification", which is typically obtained from declared specifications by means of context-dependent interpretation of modifies clauses [LN02, Mül02], specification inheritance [DL96, Lea06, Wil92, Win83], invariant disciplines [BDF+04, LM04, MPHL06, NB04], etc.

Definition 7 (satisfaction by method environment) *A method environment μ* satisfies *ST, written $\mu \models ST$, iff $\mu(C, m) \models ST(C, m)$ for all classes C and $m \in Meths\, C$.*

Note this does not require $\mu(C, m)$ to satisfy specifications of the interfaces implemented by C, nor of its superclass. The idea is that $ST(C, m)$ is the entire proof obligation imposed on an implementation of m in class C —so $\mu(C, m)$ will satisfy specifications of m in super-classes and super-interfaces provided ST has behavioral subtyping.

4.1 Ordering specifications and satisfaction at a type.

The behavioral subtyping property is expressed in terms of a refinement ordering on specifications: it says that if T is a subtype of U then $ST(T, m)$ is a stronger specification than $ST(U, m)$ in the sense that any method satisfying $ST(T, m)$ also satisfies $ST(U, m)$. This intrinsic ordering on specifications is determined by the nature of command denotations and the definition of satisfaction. Some care needs to be taken with the details, because if T is a class, a method in class T is defined to act on receiver objects of type T whereas a specification of type (U, m) imposes a requirement on state transformers for target type U. Owing to the semantics of dynamic dispatch, however, it is sound for a method in class T to assume a strengthened precondition saying that the receiver object has type T. (This is explicit in the proof obligation for method bodies in proof systems, e.g. [Mül02, PdB05].)

For a method m of class U with $mtype(U, m) = \overline{x} : \overline{T} \to V$, the relevant state transformers are in $SemMeth(U, m)$, i.e., of type $[\mathsf{self} : U, \overline{x} : \overline{T}] \rightsquigarrow [\mathsf{res} : V, \mathsf{exc} : \mathsf{Thrwbl}]$. For T, a method meaning will have type $[\mathsf{self} : T, \overline{x} : \overline{T}] \rightsquigarrow [\mathsf{res} : V, \mathsf{exc} : \mathsf{Thrwbl}]$ —only the type of self varies.

Definition 8 (satisfaction at a type, $\lfloor$, $\Vdash^T$) *Let $(pre, post)$ be a specification of type $\Gamma \rightsquigarrow \Gamma'$ and let $T \leq \Gamma\, \mathsf{self}$. Define $(pre, post)\lfloor T$ to be the specification $(pre', post)$, of type $[\Gamma \mid \mathsf{self} : T] \rightsquigarrow \Gamma'$, where pre' is defined by $(\rho, h, r) \in pre' \iff r\, \mathsf{self} = T \wedge (\rho, h, r) \in pre$. For a general specification, $(J, pre, post)\lfloor T$ is $(J, pre', post)$ where $pre'_i = pre_i \lfloor T$ for all $i \in J$.*

An element $\sigma \in STrans([\Gamma \mid \mathsf{self} : T], \Gamma')$ satisfies *$(pre, post)$ at T, written $\sigma \Vdash^T (pre, post)$, iff $\sigma \models (pre, post)\lfloor T$. For a general specification, the restriction is applied at each index.*

The reader must keep in mind that "$= \bot$" is not the same as undefined. Note that σ is not even defined outside $State([\Gamma \mid \mathsf{self} : T])$ —it has no use in our theory. On the other hand, for $(\rho, h, r) \in State([\Gamma \mid \mathsf{self} : T])$ the outcome of σ on this state, i.e., $\sigma(\rho, h, r)$ may be $\bot$ or a state.

Definition 9 (specification refinement, $\sqsupseteq^T$) *Let $spec_0$ be a specification of type $\Gamma \rightsquigarrow \Gamma'$ and $spec_1$ be of type $[\Gamma \mid \mathsf{self} : T] \rightsquigarrow \Gamma'$ where $T \leq \Gamma\, \mathsf{self}$. Then $spec_1$* refines *$spec_0$* with respect to *T, written $spec_1 \sqsupseteq^T spec_0$, iff $\sigma \models spec_1 \Rightarrow \sigma \Vdash^T spec_0$ for all $\sigma \in STrans([\Gamma \mid \mathsf{self} : T], \Gamma')$.*

This applies in particular to general specifications. Note that σ ranges over the smaller set of transformers and only weakly satisfies $spec_0$. In case $T = \Gamma\,\mathsf{self}$, however, $\sigma \models spec_0$ is the same as $\sigma \models^T spec_0$. We may omit the superscript on $\sqsupseteq$ just in this case.

Lemma 10 (weak transitivity) *Suppose $spec_0$ is a specification of type $\Gamma \rightsquigarrow \Gamma'$, $spec_1$ is of type $[\Gamma \mid \mathsf{self}\!:\!T] \rightsquigarrow \Gamma'$ where $T \leq \Gamma\,\mathsf{self}$, and $spec_2$ is of type $[\Gamma \mid \mathsf{self}\!:\!U] \rightsquigarrow \Gamma'$ with $U \leq T$. If $spec_2 \sqsupseteq^U spec_1$ and $spec_1 \sqsupseteq^T spec_0$ then $spec_2 \sqsupseteq^U spec_0$, provided $spec_1$ is satisfiable.*[9]

The preceding notions are used to formulate specification refinement, behavioral subtyping, and other notions that are beyond the scope of this document [LN06b, LN06a].

4.2 PVS Formalization

Theory PREDICATES defines typed predicates and SPECS defines method specifications, satisfaction, and refinement.

The current distribution is likely to include additional theories that pertain to behavioral subtyping.

5 Modular satisfaction and predicate transformer semantics

The ultimate aim of verification is usually that a main program, linked to supporting libraries, is correct. In practice, most tools perform modular reasoning: the libraries are verified against specifications and the main program is shown to be correct for any correct libraries. This is formalized as follows.

Definition 11 (modular satisfaction) *For command $\Gamma \vdash S$ and Γ-specification spec, we say S* modularly satisfies *spec* under *ST, and write*

$$ST, (\Gamma \vdash S) \models spec$$

if and only if

$$\forall \eta \in Menv \bullet \eta \models ST \;\Rightarrow\; [\![\Gamma \vdash S]\!](\eta) \models spec \tag{2}$$

For expression $\Gamma \vdash E\!:\!T$ and spec of type $\Gamma \rightsquigarrow [\mathsf{res}\!:\!T, \mathsf{exc}\!:\!\mathsf{Thrwbl}]$ we define $ST, (\Gamma \vdash E\!:\!T) \models spec$ iff $\forall \eta \in Menv \bullet \eta \models ST \;\Rightarrow\; [\![\Gamma \vdash E\!:\!T]\!](\eta) \models spec$

To capture supertype abstraction, this quantification over environments is not sufficient. Instead, we need a single environment containing, for each method, the least refined meaning that satisfies the method's specification. To this end we formalize a predicate transformer semantics.

[[[Predicate transformer semantics has been written up in the behavioral subtyping paper, including its derivation from the state transformer semantics. The writeup needs to be adapted to this report and the PVS encoding described.]]]

[9] To see that the satisfiability condition is necessary, let $spec_1$ be the simple specification $(pre_1, post_1)$ where $pre_1 = true$ and $post_1$ says that self **is** U. No element of $STrans([\Gamma \mid \mathsf{self}\!:\!T], \Gamma')$ satisfies $spec_1$. Owing to unsatisfiability we have $spec_1 \sqsupseteq^T spec_0$ for any $spec_0$. Define $spec_2$ to have $pre_2 = true = post_2$. Then because $\sqsupseteq^U$ restricts the initial state we get $spec_2 \sqsupseteq^U spec_1$. But it is easy to choose $spec_0$ so that $spec_2 \not\sqsupseteq^U spec_0$.

References

[AdB90] Pierre America and Frank de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84(2):129–164, 1990.

[Apt81] K.R. Apt. Ten years of Hoare's logic, a survey, part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[BDF+04] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.

[Ben04] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 14–25, 2004.

[BN02] Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 166–177, 2002.

[BN05] Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, November 2005. Extended version of [BN02].

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996.

[IPW01] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–459, May 2001.

[Lea06] Gary T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Zhiming Liu and He Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *LNCS*, pages 2–34, New York, NY, November 2006. Springer-Verlag.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[LM04] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming*, pages 491–516, 2004.

[LN02] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, 2002.

[LN06a] Gary T. Leavens and David A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 2006.

[LN06b] Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 2006.

[MPHL06] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, October 2006.

[Mül02] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[Nau05] David A. Naumann. Verifying a secure information flow analyzer. In Joe Hurd and Tom Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics TPHOLS*, volume 3603 of *LNCS*, pages 211–226, 2005.

[Nau06] David A. Naumann. From coupling relations to mated invariants for secure information flow. In *European Symposium on Research in Computer Security (ESORICS)*, number 4189 in LNCS, pages 279–296, 2006.

[NB04] David A. Naumann and Mike Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *IEEE Symp. on Logic in Computer Science (LICS)*, pages 313–323, 2004.

[PdB05] Cees Pierik and Frank S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 2005. to appear.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[SF93] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[Wil92] Alan Wills. Specification in Fresco. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[Win83] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, MIT Lab for Computer Science, 1983.

[Yan04] Hongseok Yang. Relational separation logic. *Theoretical Computer Science*, 2004. To appear.