Formal Techniques for Java Programs 2000

Proceedings, Sophia Antipolis, France
June 12, 2000

Sophia Drossopoulou, Susan Eisenbach, Bart Jacobs, Gary T. Leavens,
Peter Müller and Arnd Poetzsch-Heffter (editors)

Contents

Preface

This is the proceedings of the second workshop on *Formal Techniques for Java Programs*, June 12, 2000, held in Sophia Antipolis, France. The workshop is affiliated with the *14th European Conference on Object-Oriented Programming*, ECOOP 2000. Papers in the proceedings are included here based on the reviews of the workshop organizers. This proceedings will also be available from

`www.informatik.fernuni-hagen.de/import/pi5/publications.html`

The objective of the workshop is to bring together people developing formal techniques and tool support for Java. Formal techniques can help to analyze programs, to precisely describe program behavior, and to verify program properties. Applying such techniques to object-oriented technology is especially interesting because:

1. The OO-paradigm forms the basis for the software component industry with their need for certification techniques.

2. It is widely used for distributed and network programming.

3. The potential for reuse in OO-programming carries over to reusing specifications and proofs.

Such formal techniques are sound, only if based on a formalization of the language itself.

Java is a good platform to bridge the gap between formal techniques and practical program development. It plays an important role in these areas and is on the way to becoming a de facto standard because of its reasonably clear semantics and its standardized library.

Sophia Drossopoulou
Susan Eisenbach
Bart Jacobs
Gary T. Leavens
Peter Müller
Arnd Poetzsch-Heffter

From Executable Formal Specification to Java Property Verification

I. Attali, D. Caromel, H. Nilsson, M. Russo

From Executable Formal Specification to Java Property Verification

Extended Abstract

I. Attali, D. Caromel, H. Nilsson and M. Russo
INRIA Sophia Antipolis,
CNRS - I3S - Univ. Nice Sophia Antipolis,
BP 93, 06902 Sophia Antipolis Cedex - France
tel: 33 4 92 38 79 10
fax: 33 4 92 38 76 33
First.Last@sophia.inria.fr
http://www.inria.fr/oasis/java

1 Position Statement

To be sure of the meaning of a programming language, we need to have its formal semantics. But semantic specifications are hard to write, and it is difficult to be convinced that they are correct. Having an executable semantics helps, since this allows the semantics to be tested on real programs which tends to expose many mistakes. But ultimately the correctness of the semantics has to be proved and then preferably by means of a computerized proof assistant to avoid mistakes in the proof. This is why we are working towards machine proofs with executable formal specifications. The extended abstract that follows is a step towards that goal.

2 Introduction

In this article, we are defining a dynamic semantics of a large, concurrent subset of Java. We are not concerned with typing (we assume our programs are typed correctly). This specification is executable, but it is also used as the basis for formal verification of Java semantical properties. This executable specification enables us to derive an environment which includes visualization of programs execution and tests our semantics.

In Section 3 we present related work in the domain of formal semantics and property verification. Section 4 describes our formal specification of Java (syntax and semantics), and the graphical interactive execution environment for Java we derive from this executable semantics. Section 5 presents Java property verification in the context of multithreaded applications.. Section 6 concludes the paper.

3 Related Work

The formal specification of the Java semantics is an active research area. There are two main approaches. The first possibility is to directly specify the semantics of Java source code. This has been done by e.g. Drossopoulou and Eisenbach [7], Syme [17], Nipkow and Oheimb [14]. Another approach is to work at the JVM byte-code level. Qian [16] has specified a subset of the JVM instructions for objects, methods and subroutines. Börger and Schulte [2, 1] define the JVM in order to prove the correctness of Java compilation. Jensen, Le Metayer and Thorn [10] formalize dynamic class loading mechanisms in the JVM and study some security properties of Java.

One of the most common approaches to verification is to translate the program source code into an analysis formalism by using some abstraction techniques. Correctness properties are expressed in the same formalism and a specific mechanism is applied on the program translation to check this correctness property. This approach has been applied to prove the absence of deadlocks in multi-threaded Java programs using model-checking (as in [4], and in [9, 5] with the Spin system), and the absence of livelocks thanks to data and control-flow analysis [12].

Another important research topic is to prove properties of the language itself. Type-soundness (*Well typed programs cannot go wrong*) is an important language property studied in [7, 13, 17, 15]. In [15], Müller and Poetzsch-Heffter present a Hoare-style [11] programming logic for a sequential kernel of Java. They also specify a formal operational semantics of this subset and prove the soundness of this semantics by formalizing it and the typing rules in higher-order logic.the

Our work is original in that the same formal specifications are used both to derive an environment and as a basis for proving properties.

4 Formal Specifications For Debugging

This section presents our formal specification of a large, concurrent subset of Java. We also explain how we derive a visualizing execution environment from the formal specification.

These specifications have been written within the Centaur system [3], which is a generic programming environment: from the syntax and the semantics specifications of a given language, one can automatically produce a structure editor and semantics-based tools (such as type checkers and interpreters) for this language.

4.1 Java formal specification

Every structured object of our semantics is represented within the system as an abstract syntax tree. The abstract syntax so defined includes 128 operators and 56 types related to Java constructors (as defined in [8]). In our semantics specification (about 800 Typol rules), objects, threads, and configurations (a pair of objects and static variables) are modeled as semantic structures. Typol [6] (a formalism which is an implementation of the Natural Semantics) rules are organized in modules which enhances the design, improves the readability, and facilitates debugging of the specification. Some modules are expressed in a Natural Semantics style (especially object-oriented features). Some others are expressed in a Structural Operational Semantics style (especially concurrent features). Our operational semantics simulates concurrency with a deterministic interleaving of threads.

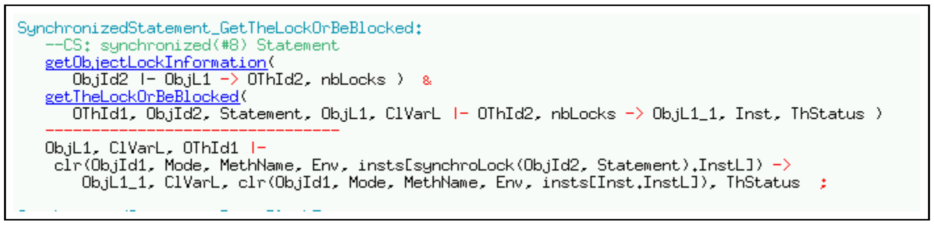
In Java, the only way for a thread to acquire a lock on a given object is to either enter a *synchronized* block or to call a *synchronized* method. A *synchronized* method is equivalent to a "normal" method whose body is enclosed in a *synchronized* block. The behavior of the *synchronized* statement is the following:

1. Compute the reference to the concerned object .
2. Perform a lock action on that object: the thread may become (temporarily) *blocked* if the object is already locked by another thread).
3. Execute the *synchronized* block body.
4. Perform an unlock action on the object.

Our semantics reflects this behavior through five Typol rules which correspond to each step of the above informal specification. Two Typol rules are corresponding to the first step of the behavior: one concerns the evaluation of the expression in order to get the reference of the object to lock, whereas the second one enables the transition between the first step and the second one of the synchronization specification. Figure 1 presents the Typol rule which correspond to the second step of the *synchronized* statement behavior.

4.2 Visualization environment

A visualization environment (see *http://www.inria/oasis/java/* for snap shots of the environment) is derived from these formal specifications. It includes both textual and graphical visualization of Java programs interpretation. This environment shows the list of objects using two visualization engines, both of them based on the semantic structure modeling the object list. The textual



```
SynchronizedStatement_GetTheLockOrBeBlocked:
   --CS: synchronized(#8) Statement
   getObjectLockInformation(
      ObjId2 |- ObjL1 -> OThId2, nbLocks )  &
   getTheLockOrBeBlocked(
      OThId1, ObjId2, Statement, ObjL1, ClVarL |- OThId2, nbLocks -> ObjL1_1, Inst, ThStatus )
   --------------------------------
   ObjL1, ClVarL, OThId1 |-
    clr(ObjId1, Mode, MethName, Env, insts[synchroLock(ObjId2, Statement).InstL]) ->
       ObjL1_1, ClVarL, clr(ObjId1, Mode, MethName, Env, insts[Inst.InstL]), ThStatus  ;
```

Figure 1: Lock the object or block the currently *executing* thread.

view is a direct pretty-printing of the list of objects and threads. The graphical view shows the complete topology of the object graph.

In order to provide graphical animation *during* program execution, the semantics interpreter notifies the two textual and graphical visualization engines of important events (state change). This is done by annotating the semantics. The annotations are effectively procedure calls for sending information to and synchronize with the visualization engines.

5 Formal Specifications For Proofs

This section presents work in progress: the verification of important properties of our semantics. We are particularly interested in synchronization properties as they are quite complex and difficult to test.

If two distinct threads can access some specific object, this object is shared by these two threads. This can potentially lead to a protection problem of the shared data;

Here we aim to present the skeleton of the proof of the following property: `Two threads cannot hold a lock on a same object at the same time` (as stated in [8], page 399). To prove this, we first observe the following:

1. When the program interpretation begins, there is only one thread: the *main* thread, that is to say a thread which has not been created explicitly by the programmer but implicitly, to execute the *main* method.

2. Synchronization needs appear only when there are at least two threads.

From point 1, we can deduce that at the execution beginning, the property holds.

The main point is to prove that if this property holds at a given step of the execution, then it still holds at the next step. Indeed, the property holds as long as no thread wants to acquire a lock on a given object. The step interesting to study is in fact when an object is locked by a given thread, and when a second thread wants to acquire a lock on this object (if the object is not already locked, there is no problem). Our semantics simulates concurrency through the introduction of interleaving between the different threads which have been created by a given program. At each execution step there is therefore only one *executing* thread. Moreover, a Typol rule is atomic, which means that nothing can happen between the execution of the two premises of the rule presented in Figure 1. The information acquired by the first premise about the absence or presence of lock on the considered object is therefore always right when the second premise is executed.

There is one special case to study. A thread (thread 1) which calls its *wait* method on a specific object, releases all the locks it owned on this object and becomes *dormant*. When another thread (thread 2) calls the *notify* or *notifyAll* method on this object, thread 1 becomes *blocked*, waiting for a lock on this object. Once thread 2 has released and thread 1 acquired the lock, thread 1 becomes executing again.

6 Conclusion

In this extended abstract, we briefly presented our semantic definition of a large subset of Java. The main point of our work is that the same formal specifications are used to derive an environment and also to prove properties.

The semantic specification, using both a small-step and a big-step style (thanks to the Typol logical framework which makes it possible to mix these two styles), includes primitives types, classes, inheritance, instance variables and methods, class variables and methods, interfaces, overloading, shadowing, dynamic method binding, object and thread creation, concurrency, arrays and exceptions. From this specification we derive a graphical programming environment. This environment is animated and interactive, it includes visualization of the object topology during program execution.

The last section presented in this abstract is a work in progress. We have to formalize the proof skeleton given in Section 5. We also intend to study other thread-specific properties. For example, if the execution of a *synchronized* method is ever completed, either normally or abruptly (with an exception), an *unlock* action is always performed on that same lock ([8], page 416).

References

[1] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *23rd International Symposium on Mathematical Foundations of Computer Science*, LNCS. Springer-Verlag, 1998. to appear.

[2] E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In *Formal Syntax and Semantics of Java*. Springer-Verlag, LNCS 1523, 1998.

[3] P. Borras and et al. Centaur: the System. In *SIGSOFT'88 Third Annual Symposium on Software Development Environments*, Boston, 1988.

[4] J. C. Corbett. Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs. Technical report, October 1998. submitted for publication.

[5] C. Demartini, R. Iosif, and R. Sisto. Modeling and Validation of Java Multithreaded Applications using Spin. In *Proc. of the 4th SPIN Workshop, Paris, France*, November 1998.

[6] T. Despeyroux. Typol: A Formalism to Implement Natural Semantics. Research Report 94, INRIA, 1988.

[7] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. *Lecture Notes in Computer Science*, 1523, 1999.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. In *International Journal on Software Tools for Technology Transfer (STTT)*, 1998. To appear.

[10] T. Jensen, D. Le Métayer, and T. Thorn. Security and Dynamic Class Loading in Java: a Formal isation. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 4–15, May 1998.

[11] L. Lamport. The 'hoare logic' of concurrent programs. In Springer-Verlag, editor, *Acta Informatica 14*, pages 21–37, 1980.

[12] G. Naumovich, S. Avrunin, and L. A. Clarke. Data Flow Analysis for Checking Properties of Concurrent Java Programs. In *Proc. of the International Conference on Software Engineering*, May 1999.

[13] T. Nipkow. Invited talk: Embedding programming languages in theorem provers. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *LNAI*, page 398, Berlin, July 7–10, 1999. Springer-Verlag.

[14] T. Nipkow and D. von Oheimb. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119+37. SV, 1998.

[15] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[16] Z. Qian. A formal specification of Java[TM] virtual machine instructions for objects, methods and subroutines. *Lecture Notes in Computer Science*, 1523, 1999.

[17] D. Syme. Proving Java type soundness. *Lecture Notes in Computer Science*, 1523, 1999.

Event Structures for Java

P. Cenciarelli

8

Event Structures for Java

Pietro Cenciarelli

University of Rome, “La Sapienza”
`cenciarelli@dsi.uniroma1.it`

Abstract A semantics based on Winskel's *event structures* is proposed for a meaningful subset of Java, including threads and synchronization. An “adequacy” result relates denotational and operational semantics. Program equivalences are discussed.

1 Introduction

Mathematical models of programming languages provide foundations for formal program development and verification, correctness proofs of compilers and of optimization techniques, and in general for all applications of formal methods. While the use of formal methods is well established for languages like “C”, Pascal or ML [14,13], it is not yet so for concurrent object-oriented languages like Java [7]: the systems being currently developed for this language [10,9] capture important aspects concerning types, subclassing and iheritance, but not concurrency.

A denotational description of concurrent objects capturing the Java model of concurrency is not yet available in the literature. To cite a few examples, neither the *CuPer* model of Abadi and Cardelli [1] nor the one by Jacobs [8] handle concurrency. The difficulty may derive from the fact that, unlike in process calculi (where the parallel composition of two processes is a process), two objects running in parallel do not form an object but a “computation” (thus contravening to Cardelli's slogan that “everything is an object” [3]).

On the other end, several operational descriptions of Java capturing concurrency have been proposed in the literature [2,5,6], but here the natural notion of contextual equivalence which comes with operational semantics is hardly usable since no “context lemma” is available for Java-like languages, as it is for PCF [12] or for functional languages with effects [11].

In the present paper we propose a semantics of a meaningful subset of Java, including threads and synchronization, based on Winskel's *event structures* [16]. There is a rich theory developed for this fundamental model of concurrency [15], which we so make available for studying notions of equivalence for Java programs. However, we show that the obvious notion which only takes as equivalent programs which denote the same event structure, is not enough to prove the *commutativity of assignment*, a rather surprising property of Java, stating that one thread cannot observe the order in which another thread performs assignments to (distinct) *shared* variables.

2 Operational semantics

In [5] a *structural* operational semantics is proposed, which captures a significant portion of Java, including the running and stopping of threads, thread interaction via shared memory, synchronization by monitoring and notification, and sequential control mechanisms such as exception handling and return statements. The operational semantics is parametric in the notion of "event space" [4], which formalizes the rules that threads and memory must obey in their interaction.

Each thread of control has, in Java, a private *working memory* in which it keeps its own working copy of variables that it must use or assign. As the thread executes a program, it operates on these working copies. The main memory contains the master copy of each variable. There are rules about when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy or vice versa. The process of copying is *asynchronous.* There are also rules which regulate the *locking* and *unlocking* of objects, by means of which threads synchronize with each other. All this is described precisely in [7, §17] in terms of eight kinds of low-level actions: *Use*, *Assign*, *Load*, *Store*, *Read*, *Write*, *Lock*, and *Unlock*. Here is an example of a rule from [7, §17.6, p. 407] involving locks and variables. Let T be a thread, V a variable and L a lock: "Between an *Assign* action by T on V and a subsequent *Unlock* action by T on L, a *Store* action by T on V must intervene; moreover, the *Write* action corresponding to that *Store* must precede the *Unlock* action, as seen by the main memory."

Let thread identifiers be ranged over by θ, objects by o, left values (or "variables") by l and (right) values by v. Formally, an *action* is either a triple (A, θ, o), where $A \in \{\mathit{Lock}, \mathit{Unlock}\}$, or a 4-tuple of the form (A, θ, l, v), where $A \in \{\mathit{Use}, \mathit{Assign}, \mathit{Load}, \mathit{Store}, \mathit{Read}, \mathit{Write}\}$. When $A \in \{\mathit{Use}, \mathit{Assign}, \mathit{Load}, \mathit{Store}\}$, the tuple (A, θ, l, v) records that the thread θ performs an A action on l with value v, while, if $A \in \{\mathit{Read}, \mathit{Write}\}$, it records that the main memory performs an A action on l with value v on behalf of θ. If A is *Lock* or *Unlock*, (A, θ, o) records that θ acquires, or respectively relinquishes, a lock on o. Actions with name *Use*, *Assign*, *Load*, *Store*, *Lock* and *Unlock* are called *thread actions*, while *Read*, *Write*, *Lock* and *Unlock* are *memory actions*.

Events are instances of actions, which we think of as happening at different times during execution. We use the same tuple notation for actions and their instances: the context clarifies which one is meant.

In [5, §2.2] (to which we refer for more detail) the rules of interaction that threads and memory must obey are formalised by means of 17 logical clauses viewed as well-formedness conditions on posets of events called *event spaces.* The events of such a poset, which are thought of as occurring in the given order, are meant to record the activity of memory and threads during the execution of a Java program. The ajoining of a new event to an event space $\eta = (X, \leq_X)$ can be expressed by means of an operation $\oplus$ defined as follows: $\eta \oplus a$ denotes nondeterministically an event space $\eta' = (Y, \leq_Y)$ such that:

- η' extends η consevatively w.r.t. the ordering, with $Y = X \cup \{a\}$;

– if a is a thread action performed by θ, then $a' \leq_Y a$ for all thread actions a' by θ in η';
– if a is a memory action on x, then a' for all memory actions a' on x in η'.

If no event space η' exists satisfying these conditions, then $\eta \oplus a$ is undefined.

A *configuration* of the operational semantics is a triple (T, η, μ), where T is a partial map from thread identifiers to (abstract) Java terms, η is an event space and μ is a memory (shared by all threads). Event spaces are included in the configurations to record "historical" information on the computation which constrains the execution of certain actions according to the language specification, and hence the applicability of certain operational rules. Each term $T(\theta)$ comes equipped with a stack, and represents the segment of code being currently executed by thread θ. An *initial* configuration (T, η, μ) is roughly one where the stacks of all threads in T are empty, η is empty and μ contains just the relevant type information for T. A configuration (T, η, μ) is called *final* when all abstract terms in T are results and η is *complete* (we refer to [5] for a precise definition of "results" and "complete").

In presenting the operational rules (see the appendix), only the relevant parts of a configuration are made explicit. For example, the rule [access3]:

$$(\theta, l), \eta \to (\theta, v), \eta \oplus (\mathit{Use}, \theta, l, v)$$

says that a configuration (T, η, μ), where $T(\theta)$ is a location l evaluates to a configuration (T', η', μ'), where $T'(\theta) = v$, the value stored at l, $\eta' = \eta \oplus (\mathit{Use}, \theta, l, v)$, and μ and μ', having been omitted, are the same memory. Note that this rule can be fired only if $\eta \oplus (\mathit{Use}, \theta, l, v)$ yields a well-formed event space, thus satisfying the constraints of the language specification. In this specific case the constraints require that the value v of l has been previously loaded from the main memory into the working memory of θ or assigned by this thread to l. This information is recorded in η, and this explains the apparent "guessing" of the value v made by the operational rule.

The *operational semantics* is the smallest binary relation $\to$ on configurations which is closed under the rules given in the appendix. Related pairs of configurations are written $\gamma_1 \to \gamma_2$ and called *operational judgements* or *transitions*. A *computation* is a chain $\gamma_0 \to \gamma_1 \to \dots$ of transitions, where γ_0 is initial, and which either ends with a final configuration, or is infinite, and all the event spaces in it are completable.

Let a fat arrow $\Rightarrow$ to denote "uneventful" computation. For example, $x = 4 + 1 \Rightarrow 5$, where x is a *local* variable, indicates that, for any η and μ, the evaluation $(x = 4 + 1, \eta, \mu) \to^* (5, \eta', \mu')$ is uneventful, that is $\eta = \eta'$. Such computations do not involve the memory, that is $\mu = \mu'$. The following properties hold of the operational semantics:

Property 1. Let $\gamma \to^* (T, \eta, \mu)$ and $\gamma \to^* (T', \eta', \mu')$. If $\eta = \eta'$ then $\mu = \mu'$ and there exists T'' such that $T \Rightarrow T''$ and $T' \Rightarrow T''$.

Property 2. If $(T_1, \eta_1, \mu_1) \to^* (T_2, \eta_2, \mu_2)$ and $T_1 \Rightarrow T_1'$, then there exists T_2' such that $(T_1', \eta_1, \mu_1) \to^* (T_2', \eta_2, \mu_2)$ and $T_2 \Rightarrow T_2'$.

3 Event structures for Java

Below we map Java programs P to event structures $[\![P]\!]$. Let P compile into an initial configuration γ_0, and let Γ_P be the set of all event spaces which belong to some computation from γ_0. When quantifying over computations we shall mean those starting from γ_0, unless otherwise stated.

It is easy to show that any initial segment of an event space is itself an event space. If η is an event space and $a \in \eta$, we let $\eta \downarrow e$ be the poset $\{e' \in \eta \mid e' \leq e\}$ whose order is inherited from η. Similarly, we write $\eta \downarrow^- e$ be the poset $\{e' \in \eta \mid e' < e\}$ We call *primes* of η all event spaces of the form $\eta \downarrow e$. Let E be the set of all the event spaces which are primes of some element of Γ_P. Ordering E by subposet inclusion we obtain the underlying poset of $[\![P]\!]$.

Next we define conflict on general event spaces in terms of conflict on *actions*: We write $a_1 \# a_2$ iff a_1 and a_2 are distinct thread actions performed by the same thread or distinct memory actions performed on the same object or variable. For example, $(\mathit{Use}, \theta, l, v) \# (\mathit{Store}, \theta, l', v')$.

Definition 1 *Two event spaces* η *and* η' *in* E *are* in conflict, *written* $\eta \# \eta'$, *when there exist* $a \in \eta$ *and* $a' \in \eta'$ *such that* $a \# a'$ *and* $\eta \downarrow^- a = \eta' \downarrow^- a'$.

The above definition says that one thread can do one action at a time and that only one action at a time can be done upon the same variable. Restricting # to prime event spaces we obtain:

Theorem 2 $(E, \leq, \#)$ *is an event structure.*

Proof. The relations $\leq$ and # are disjoint. In fact, assume that $\eta \leq \eta'$ and let $\eta \downarrow^- a = \eta' \downarrow^- a'$. Since $\eta \subseteq \eta'$ we have $\eta \downarrow^- a = \eta' \downarrow^- a$. If a and a' are both memory actions performed by the same thread, it must be either $a \leq a'$ or $a' \leq a$ in η'. But then it must be $a = a'$ because, by definition, neither $a' \in \eta' \downarrow^- a'$ nor $a \in \eta' \downarrow^- a$. Similarly for memory actions performed on the same variable. Therefore it cannot be $\eta \# \eta'$. Finally, # is hereditary. In fact, assume that $\eta \leq \eta'$ and $\eta \# \eta''$, that is $\eta \downarrow^- a = \eta'' \downarrow^- a'$ for some $a \# a'$. Since $\eta \subseteq \eta'$, we have $\eta \downarrow^- a = \eta' \downarrow^- a$, and hence $\eta' \# \eta''$. Similarly for memory actions. □

Lemma 3 *If* $\eta_1 \leq \eta_2$ *in* E*, there exist* η'_1 *and* η'_2 *in* Γ*, with* η_1 *a prime of* η'_1 *and* η_2 *a prime of* η'_2 *and* $(T, \eta'_1, \mu) \rightarrow^* (T', \eta'_2, \mu')$ *for some computation.*

The *configurations* of an event structure S are defined to be the the conflict-free, downward-closed subsets of S. Operational and denotational semantics are so related:

Theorem 4 Γ *is isomorphic to the set of configurations of* $(E, \leq, \#)$.

Proof (sketch). Primes which are not in conflict can be merged together to form a legal event space. Property 1 and 2 of the previous section and the above lemma are used to show that such an event space can be obtained by running the operational semantics. $\square$

This result shows the adequacy of the event structure model with respect to the operational semantics. In particular it shows that configurations obtained by pasting primes together can all be reached by computation, and vice-versa. Yet, as the following example shows, identity of denotation does not yield an abstract enough notion of program equivalence.

Example. Let T be a concurrent program such that $T(\theta) = (x.i = 1\,;\, x.j = 2)$ and let T' be such that $T'(\theta) = (x.j = 2\,;\, x.i = 1)$, and agree with T on all the rest. Let $a_1 = (\mathit{Assign}, \theta, x.i, 1)$ and $a_2 = (\mathit{Assign}, \theta, x.j, 2)$. $[\![T]\!]$ is different from $[\![T']\!]$ because $a_2 \leq a_1$ in no event space of $[\![T]\!]$ and vice-versa for T'. However, since the copying to the main memory of the values assigned by a thread to distinct shared variables are not required by the language specification (and hence by the operational semantics) to happen in the same order in which the assignments are made, the swapping of the two actions cannot be observed by any other thread. To wit, we call this property the *commutativity of assignment.* $\square$

Commutativity of assignment is captured by abstracting away from thread activity, and observing the main memory only. More precisely, let $\eta|_{read}$ be the subposet of η made of all and only *Write* actions. Define $T \equiv T'$ iff $[\![T]\!]|_{read} = [\![T']\!]|_{read}$. This equivalence relates the two programs of the example.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer-Verlag, New York, 1996. ISBN 0-387-94775-2.
2. E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.
3. L. Cardelli. Everything is an object. Dagstuhl seminar report 216, Internationales Begegnungs- und Forschungszentrum fuer Informatik, Schloss Dagstuhl, 1998.
4. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From Sequential to Multi-Threaded Java: an Event-Based Operational Semantics. In Michael Johnson, editor, *Algebraic Methodology and Software Technology (proc. of AMAST'97)*, pages 75–90, 1997. LNCS 1349, Springer.
5. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, 1523 LNCS. Springer, 1998.
6. Andrew D. Gordon and Paul D. Hankin. A Concurrent Object Calculus: Reduction and Typing. In Uwe Nestmann and Benjamin C. Pierce, editors, *Proc. Conf. High-Level Concurrent Languages*, volume 16(3) of *Electr. Notes Theo. Comp. Sci.*, Amsterdam, 1998. Elsevier.

7. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
8. B.P.F. Jacobs. Objects and classes, co-algebraically. In B. Frietag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, 1996.
9. B.P.F. Jacobs. Coalgebraic reasoning about classes in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 11:235–246, 1998.
10. K. Leino. Axiomatic semantics of object-oriented languages. Presented at the Dagstuhl seminar on "The Semantic Challenge of Object-Oriented Programming", Schloss Dagstuhl, Wadern, Germany, 28/6 - 3/7 1998.
11. I.A. Mason and C.L. Talcott. References, Local Variables and Operational Reasoning. In *Proceedings of 7th Annual Symposium on Logic in Computer Science*, pages 186–197, 1992.
12. R. Milner. Fully abstract models of typed lambda λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.
13. Michael Norrish. *C formalised in HOL.* PhD thesis, Cambridge University, Computer Laboratory, 1998. http://www.cl.cam.ac.uk/users/mn200/PhD/thesis-report.ps.gz.
14. D. Sannella. Formal program development in extended ml for the working programmer. Technical Report ECS-LFCS-89-102, Department of Computer Science, Edinburgh University, 1989.
15. G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, editor, *Handbook of Logic in Computer Science, Volume 4*, pages 1–148. Clarendon Press, Oxford, 1995.
16. Glynn Winskel. An Introduction to Event Structures. In Jacobus W. de Bakker, editor, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 354 in LNCS. Springer, 1988.

Appendix: Operational rules

[assign1]	$\dfrac{e_1 \to e_2}{e_1 = e \to e_2 = e}$	[assign2]	$\dfrac{e_1 \to e_2}{i = e_1 \to i = e_2}$
[assign3]	$\dfrac{e_1 \to e_2}{l = e_1 \to l = e_2}$	[assign4]	$i = v\,, \sigma \to v\,, \sigma[i \mapsto v]$
[assign5]	$(\theta, l = v), \eta \to (\theta, v), \eta \oplus (\mathit{Assign}, \theta, l, v)$		
[access1]	$\dfrac{e_1 \to e_2}{e_1 \,.\, f \to e_2 \,.\, f}$	[access2][1]	$\mathit{null} \,.\, f, \mu \to \mathit{throw}(o) \,.\, f, \mu'$
[access3]	$(\theta, l), \eta \to (\theta, v), \eta \oplus (\mathit{Use}, \theta, l, v)$		
[this]	$\texttt{this}\,, \sigma \to \sigma(\mathit{this})\,, \sigma$	[var]	$i\,, \sigma \to \sigma(i)\,, \sigma$
[new]	$\texttt{new}\ C\ (\,), \mu \to \mathit{new}(C, \mu)$	[lit]	$k \to \mathit{value}(k)$
[unop1]	$\dfrac{e_1 \to e_2}{\texttt{op}\ e_1 \to \texttt{op}\ e_2}$	[unop2]	$\texttt{op}\ v \to \mathit{op}(v)$
[binop1]	$\dfrac{e_1 \to e_2}{e_1\ \texttt{bop}\ e \to e_2\ \texttt{bop}\ e}$	[binop2]	$\dfrac{e_1 \to e_2}{v\ \texttt{bop}\ e_1 \to v\ \texttt{bop}\ e_2}$
[binop3]	$v_1\ \texttt{bop}\ v_2 \to \mathit{bop}(v_1, v_2)$		
[parseq1]	$\dfrac{e_1 \to e_2}{e_1\ E \to e_2\ E}$	[parseq2]	$\dfrac{E_1 \to E_2}{v\,E_1 \to v\,E_2}$
[call1]	$\dfrac{e_1 \to e_2}{e_1.m(E) \to e_2.m(E)}$	[call2]	$\dfrac{E_1 \to E_2}{o.m(E_1) \to o.m(E_2)}$
[call3]	$o.m(V) \to \mathit{frame}(o, m, V)$	[call4][1]	$\mathit{null}.m(V), \mu \to \mathit{throw}(o), \mu'$
[frame]	$\dfrac{b_1 \to b_2}{(m, b_1) \to (m, b_2)}$	[exit1]	$(m, \{\,\})\,; \to *$
[exit2]	$(m, \{\,\mathit{return}\ S\,\})\,; \to *$	[exit3]	$(m, \{\,\mathit{return}\ v\ S\,\}) \to v$
[decl]	$\dfrac{e_1 \to e_2}{\tau\ i = e_1\ D; \to \tau\ i = e_2\ D;}$		
[locvardecl1]	$\tau\ i = v\,d\,D;\ ,\ \sigma \to \tau\ d\,D;\ ,\ \sigma[i = v]$		
[locvardecl2]	$\tau\ i = v\,;\ ,\ \sigma \to *\,, \sigma[i = v]$		
[expstat1]	$\dfrac{e_1 \to e_2}{e_1\,; \to e_2\,;}$	[expstat2]	$v\,; \to *$
[skip]	$; \to *$	[if1]	$\dfrac{e_1 \to e_2}{\texttt{if}(e_1)\ s \to \texttt{if}(e_2)\ s}$
[if2]	$\texttt{if}(\mathit{true})\ s \to s$	[if3]	$\texttt{if}(\mathit{false})\ s \to *$

[1] where $(o, \mu') = \mathit{new}(\texttt{NullPointerException}, \mu)$

[statseq] $\dfrac{s_1 \to s_2}{s_1\, S \to s_2\, S}$ [$*$] $* S \to S$

[block1] $\{\ \} \to *$

[block2] $\dfrac{S_1, push(\rho_1, \sigma_1) \to S_2, push(\rho_2, \sigma_2)}{\{S_1\}_{\rho_1}, \sigma_1 \to \{S_2\}_{\rho_2}, \sigma_2}$

[syn1][2] $\dfrac{e_1 \to e_2}{\texttt{synchronized}\,(e_1)\ b \to \texttt{synchronized}\,(e_2)\ b}$

[syn2] $\dfrac{e, \eta_1 \to o, \eta_2}{(\theta, \texttt{synchronized}\,(e)\ b), \eta_1 \to \texttt{synchronized}\,(o)\ b, \eta_2 \oplus (Lock, \theta, o)}$

[syn3] $\dfrac{b_1 \to b_2}{\texttt{synchronized}\,(o)\ b_1 \to \texttt{synchronized}\,(o)\ b_2}$

[syn4] $\dfrac{b, \eta_1 \to c, \eta_2}{(\theta, \texttt{synchronized}\,(o)\ b), \eta_1 \to c, \eta_2 \oplus (Unlock, \theta, o)}$

[read][3] $T, \eta, \mu \to T, \eta \oplus (Read, \theta, l, \mu(l)), \mu$

[load][3] $T, \eta \to T, \eta \oplus (Load, \theta, l, v)$

[store][3] $T, \eta \to T, \eta \oplus (Store, \theta, l, v)$

[write][3] $T, \eta, \mu \to T, \eta \oplus (Write, \theta, l, v), \mu[l \mapsto v]$

---
[2] if $e_2 \notin RVal$

[3] if $T(\theta)$ is defined

A Core Calculus for Java Exceptions

D. Ancona, G. Lagorio, E. Zucca

A Core Calculus for Java Exceptions[⋆]

(Extended Abstract)

Davide Ancona, Giovanni Lagorio, and Elena Zucca

DISI - Università di Genova
Via Dodecaneso, 35, 16146 Genova (Italy)
email: {davide,lagorio,zucca}@disi.unige.it

Abstract. In this paper we present a simple calculus (called CJE) corresponding to a small functional fragment of Java supporting exceptions. We provide a reduction semantics for the calculus together with two equivalent type systems; the former corresponds to the specification given in [5] and its formalization in [4], whereas the latter can be considered an optimization of the former where only the minimal type information about classes/interfaces and methods are collected in order to type-check a program. The two type systems are proved to be equivalent and a subject reduction theorem is given.

1 Introduction

The aim of this paper is to deeply investigate the exception mechanism of Java (in particular its interaction with inheritance) by means of a simple calculus, called CJE for Calculus of Java Exceptions.

Although calculi for Java exceptions have been considered already in [3, 2], much still has to be said about this topic; indeed, the approach taken here originates from [2] and presents several interesting novelties in comparison with [3].

First, we have deliberately omitted from the calculus all the features which we consider orthogonal w.r.t. the exception mechanism in Java. Indeed here, following the approach of Featherweight Java (abbreviated FJ) in [6], the idea is to keep the calculus as simple as possible rather than trying to give a soundness result for the whole Java language.

Even though it is certainly useful to have a full formal description of the semantics of Java (and [4, 3] go toward this direction), we think that, when proving specific properties or trying to clarify some tricky point in the informal specification, it is better to separate the concerns and to consider an essential subset of the language. As a result, the subject reduction proof we give for CJE is rather simple and compact.

Since in this paper we want to focus on Java exception handling, CJE clearly includes all the linguistic mechanisms for handling exceptions (`finally` excluded[1]), but many others are omitted; among them, fields, method overloading (even though not completely, see the discussion below about abstract classes), constructors, `super` and even assignment; indeed, like FJ, CJE has no statements (hence is a functional language). More precisely, the calculus has only four constructs: method invocation, object creation and the `throw`, `try` and `catch` expressions.

On the other hand, we have not omitted abstract classes since there are some subtle points in the rules of the Java language specification [5] for checking conflicts in the `throws` clause and such rules are particularly complex for abstract classes. These problems do not emerge from the formalization given in [4, 3] where abstract classes are not considered.

Second, we are interested in another kind of simplification which corresponds to minimizing the information associated with classes and interfaces needed in order to type-check a program. Indeed, the type information about classes and interfaces used in the type system defined in [4, 3]

⋆ Partially supported by Murst - TOSCA Teoria della Concorrenza, Linguaggi di Ordine Superiore e Strutture di Tipi and APPlied SEMantics - Esprit Working Group 26142.

[1] The `finally` clause makes no sense in a functional setting; on the other hand, the extension of the type system to include such mechanism does not imply any problem.

is somehow redundant. Avoiding such redundancy makes, in our opinion, the type system clearer and helps to shed some light on the obscure points of the Java specification given in [5] (which can be easily misinterpreted, as shown in the example in Sect.2).

However, this kind of minimization, even though advocated for sake of clarity, neither necessarily implies a simplification of the proofs, nor aims at making type-checking of Java more efficient (even though, in principle, it could be the case and this matter deserves further investigation).

In Sect.2 we give a paradigmatic and motivating example used for showing the complexity of the rules for the compatibility checks of `throws` clauses and for explaining the notion of minimal type. Sect.3 is an outline of the formal definition of the calculus and of the main technical results. Finally, in Sect.4 we draw some conclusion and claim that some modification to the Java language could partly avoid the complexity of the rules.

An extended version of this paper can be found in [1].

2 A paradigmatic example in Java

Consider the following Java code fragment:

```
interface I {
 void m1() throws E1;
 void m2() throws E1;
 void m3();
}
abstract class C1 {
 public abstract void m1() throws E0,E2;
 public void m2() throws E0{}
 public void m3() throws E0{}
}
abstract class C2 extends C1 implements I {
 public abstract void m3();
}
```

If we assume that `E0`, `E1` and `E2` are exception types s.t. `E0` $\leq_c^\rho$ `E1` $\leq_c^\rho$ `E2` (where $\leq_c^\rho$ corresponds to the subclass relation determined by an environment ρ of classes/interfaces corresponding to a Java program; see Sect.3), then the declaration of the class `C2` is statically correct. Indeed, according to [5] 8.4.6.4, methods `m1` of `I` and `m1` of `C1` are both inherited by `C2` and no compatibility check for the `throws` clauses is required; note that if `m1` were not abstract in `C1` then the code would be not correct. Indeed, in that case `m1` in `C1` would override, and therefore implement, `m1` in `I` and a compatibility check for the `throws` clauses (which clearly would fail since `E2`$\not\leq_c^\rho$`E1`) would be required.

On the other hand, `m2` in `C1` overrides `m2` in `I`, therefore the compatibility check has to be performed and in this case it is passed since `E0` $\leq_c^\rho$ `E1`.

Finally, `m3` in `C2` overrides `m3` in `I` and `C1`, hence the two corresponding checks have to be performed, whereas no check is needed between `m3` in `I` and `m3` in `C1`; note that this last check would be performed and would fail if `m3` were not in `C2`.

From this example it should be clear that a type system modeling compatibility checks for exceptions in presence of an inheritance hierarchy including interfaces and abstract classes is really needed. Indeed, such a type system would provide a formal basis for understanding what is going on and whether these complex rules are really necessary or, rather, it is better to take a simpler approach by somehow restricting the language, as briefly discussed in Sect.4.

Let us now show what we mean by simplification of types. According to [5] and its formalization in [4, 3], the type information associated with the class `C2` can be expressed as follows:

```
{void m1() throws E1; void m1() throws E0,E2;
 void m2() throws E0; void m3()}
```

First, we can notice that the clause `throws E0,E2` contains some redundancy, since it is equivalent to `throws E2`, by virtue of the hypothesis `E0` $\leq_c^\rho$ `E2`; hence, we can apply a simplification step to the type above obtaining a new type where all the `throws` clauses are minimal.

The other redundacy is that the method `void m1()` is repeated twice, whereas for type-checking classes/interfaces it is enough to have a unique occurence where the `throws` clause is obtained by means of a sort of "intersection" operator (which formally corresponds to take the greatest lower bound w.r.t. the natural order for exception sets); therefore, applying this second simplification step we obtain the minimal type

```
{void m1() throws E1; void m2() throws E0; void m3()}.
```

3 Formal definitions and results

The abstract syntax and the reduction semantics of CJE are given in Fig.1 and 2, respectively.

In the syntax we use the notation A^* to indicate a sequence of zero or more occurrences of A and $A^{\circledast}$ (resp. $A^{\oplus}$) to indicate a set (resp. non empty set) of occurrences of A, that is, a sequence in which there are no repetitions and the order is immaterial. The terminals `iname` and `cname` indicate interface and class names respectively. A generic name is indicated by `name`.

Note that since CJE is a functional language, the `throw` and the `try` and `catch` constructs are not statements, as happens in Java, but expressions; furthermore, for sake of simplicity, the `throw` expression is built on top of class names (corresponding to exception names) rather than generic expressions.

The metavariable E ranges over *expr*, C over class names and m over method names.

There are two kinds of normal forms: those having form `new` C, corresponding to normal program terminations evaluating into an object of class C, and those of the form `throw` C, corresponding to abnormal program terminations throwing the exception C.

The reduction relation $\rightarrow_\rho$, the auxiliary function $Body_\rho$ and the subclass relation $\leq_c^\rho$ are all indexed by ρ which is the environment of classes/interfaces w.r.t. which the reduction is performed; more precisely, ρ represents a Java program P and associates with any class/interface name the corresponding declaration in P.

The auxiliary function $Body_\rho$, whose definition has been omitted for lack of space, performs method look-up in the environment ρ: $Body_\rho(C, m)$ returns the tuple $\langle E, x_1 \dots x_n \rangle$ corresponding to the body and the formal parameters, respectively, of the method named m found when starting look-up at the class C in the environment ρ; if the method is not found, then $Body_\rho(C, m)$ is undefined.

Two different type systems are considered. The former, which we call *full*, corresponds to the specification given in [5] and formalized in [4, 3], whereas the latter, which we call *minimal*, uses minimal types. For lack of space Fig.3 contains only the rules for type assignment to classes/interfaces (see [1] for the complete set of rules).

The type of a class/interface is a pair consisting of a *fields type* and a *methods type*. A fields type is a set of fields, that is, pairs consisting of a field name and a field type; a methods type is a set of methods, that is, pairs consisting of a method name qualified by the types of the arguments (what is usually called a *signature*) and a triple consisting of the *kind* (abstract or not abstract), the return type and the set of declared exceptions.

The rules in Fig.3 have the same structure for both the full and the minimal type systems. What changes is the definition of the auxiliary *sum* functions $\overset{\Gamma}{\oplus}$ used for defining the rules (see below).

The first rule defines the type of an interface I, which consists of an empty fields type and a methods type which is the sum of the methods types of the direct superinterfaces (I' s.t. $I <_i^1 I'$ in Γ), updated by the methods MST declared in I.

The second rule defines the type of a class C. The field types consists in the fields type FST' of the direct superclass updated by the fields FST declared in C. The methods type consists of the sum of the methods types of the implemented interfaces (I s.t. $C \lhd_i^1 I$ in Γ) and the abstract

```
     prog ::= decl⊛
     decl ::= idecl | cdecl
     type ::= iname | cname
 exc-type ::= cname⊛
    cdecl ::= [ abstract ] class cname extends cname
              implements iname⊛ { meth⊛ }
    idecl ::= interface iname extends iname⊛ { imeth⊛ }
   params ::= (⟨ type name ⟩*)
    imeth ::= abstract type name params throws exc-type
     meth ::= instance type name params throws exc-type { expr } |
              imeth

     expr ::= name |
              new cname |
              throw cname |
              expr.name(expr⊛) |
              try expr ⟨catch cname expr⟩⊕
```

Fig. 1. Syntax

methods of the superclass C', updated by the non abstract methods of C' updated in turn by the methods MST declared in C. The last side condition expresses the constraint that a class with abstract methods must be declared abstract (see [5] 8.4.3.1).

The auxiliary *update* operations on fields and methods types, written $\_[\_]$, return a new fields (resp. methods) type obtained updating the first argument with new fields (resp. methods), if this is possible, accordingly with Java rules on hiding, overloading and overriding (see [5] 8.4.6 and 8.4.7). For instance, updating a methods type is undefined if we try to override a method with another which has the same signature and incompatible `throws` clause.

The sum operation $\overset{\Gamma}{\oplus}$ is just set union in the full type system, whereas in the minimal type system it is a modified union that can merge method types as illustrated in Sect.2. More precisely, merging two method types having ES and ES' as sets of declared exceptions, respectively, produces just one method type whose set of declared exceptions is the "intersection" of ES and ES' defined by

$C \in ES \overset{\Gamma}{\otimes} ES'$ iff either $C \in ES$ and $\exists C' \in ES'$ $s.t.$ $\Gamma \vdash C \leq C'$ or conversely.

When trying to sum two methods with the same signature, the operation is defined (in both versions) only if such methods have the same return type.

The two auxiliary functions *Abstract* and *NonAbstract* return the set containing only the abstract and non abstract methods, respectively.

For the formal definition of update and sum operations see [1].

In order to state that the full and minimal type systems are equivalent, for each type environment Γ (containing all the type information about the classes and interfaces in a program) we define a function $simp_\Gamma : FullTypes \rightarrow MinTypes$ which, given a full type τ, returns its simplified version $simp_\Gamma(\tau)$, that is, a minimal type. This function is indexed by Γ since the simplification depends on the subclass and subinterface relations which holds for a given program, as explained in Sect.2.

Actually, the function $simp_\Gamma$ is a logical relation $\sim_\Gamma \subseteq FullTypes \times MinTypes$ defined by $\tau \sim_\Gamma \tau'$ iff $simp_\Gamma(\tau) = \tau'$. This relation is used in the following theorem stating the equivalence of the full and minimal type systems.

Theorem 1 (Equivalence of the Type Systems). *For any type environment Γ, CJE expression E and type τ, if $\Gamma \vdash_f E{:}\tau$ then there exists τ' s.t. $\Gamma \vdash_m E{:}\tau'$ and $\tau \sim_\Gamma \tau'$. Conversely, if $\Gamma \vdash_m E{:}\tau$ then there exists τ' s.t. $\Gamma \vdash_f E{:}\tau'$ and $\tau' \sim_\Gamma \tau$.*

$$\frac{E \rightarrow_\rho E'}{E.m(E_1,\ldots,E_n) \rightarrow_\rho E'.m(E_1,\ldots,E_n)}$$

$$\frac{}{(\texttt{throw}\ C).m(E_1,\ldots,E_n) \rightarrow_\rho \texttt{throw}\ C}$$

$$\frac{E_i \rightarrow_\rho E_i'}{\begin{array}{l}\texttt{new}\ C.m(\texttt{new}\ C_1,\ldots,\texttt{new}\ C_{i-1},E_i,\ldots,E_n) \rightarrow_\rho \\ \texttt{new}\ C.m(\texttt{new}\ C_1,\ldots,\texttt{new}\ C_{i-1},E_i',\ldots,E_n)\end{array}}$$

$$\frac{}{\texttt{new}\ C.m(\texttt{new}\ C_1,\ldots,\texttt{new}\ C_{i-1},\texttt{throw}\ C_i,\ldots,E_n) \rightarrow_\rho \texttt{throw}\ C_i}$$

$$\frac{}{\begin{array}{l}\texttt{new}\ C.m(\texttt{new}\ C_1,\ldots,\texttt{new}\ C_n) \rightarrow_\rho \\ E\{x_1 \mapsto \texttt{new}\ C_1,\ldots,x_n \mapsto \texttt{new}\ C_n, \texttt{this} \mapsto \texttt{new}\ C\}\end{array}} \qquad \langle E, x_1 \ldots x_n\rangle = Body_\rho(C,m)$$

$$\frac{E \rightarrow_\rho E'}{\begin{array}{l}\texttt{try}\ E\ \texttt{catch}\ C_1\ E_1 \ldots \texttt{catch}\ C_n\ E_n \rightarrow_\rho \\ \texttt{try}\ E'\ \texttt{catch}\ C_1\ E_1 \ldots \texttt{catch}\ C_n\ E_n\end{array}}$$

$$\frac{}{\texttt{try new}\ C\ \texttt{catch}\ C_1\ E_1 \ldots \texttt{catch}\ C_n\ E_n \rightarrow_\rho \texttt{new}\ C}$$

$$\frac{}{\texttt{try throw}\ C\ \texttt{catch}\ C_1\ E_1 \ldots \texttt{catch}\ C_n\ E_n \rightarrow_\rho E_i} \qquad \begin{array}{l}\forall\, i,j = 1 \ldots n \\ C \leq_c^\rho C_i \wedge \\ (j < i \Rightarrow C \not\leq_c^\rho C_j)\end{array}$$

$$\frac{}{\texttt{try throw}\ C\ \texttt{catch}\ C_1\ E_1 \ldots \texttt{catch}\ C_n\ E_n \rightarrow_\rho \texttt{throw}\ C} \qquad \begin{array}{l}\forall\, i = 1 \ldots n \\ C \not\leq_c^\rho C_i\end{array}$$

Fig. 2. Reduction rules

Here $\Gamma \vdash_f E{:}\tau$ and $\Gamma \vdash_m E{:}\tau'$ are the usual typing judgments for expressions in the full and the minimal type system, respectively.

As usual, the theorem can be proved by induction on the typing rules and by proving analogous properties for all the other judgments of the system. For instance we expect also the following property to hold: for any type environment Γ, $\Gamma \vdash_f \diamond$ iff $\Gamma \vdash_m \diamond$, where $\Gamma \vdash_f \diamond$ and $\Gamma \vdash_m \diamond$ are the judgments for well-formed type environments in the full and the minimal type system, respectively.

Another important property that we prove is subject reduction.

Theorem 2 (Subject Reduction). *For any type environment Γ, class/interface environment ρ, CJE expressions E, E' and type τ, if $\Gamma \vdash_f E{:}\tau$, $\Gamma \vdash_f \rho \diamond$ and $E \rightarrow_\rho E'$, then there exists a type τ' s.t. $\Gamma \vdash_f E'{:}\tau'$ and $\Gamma \vdash_f \tau' \leq \tau$.*

Here $\Gamma \vdash_f \rho \diamond$ is the judgment for well-formed class/interface environments ρ w.r.t. a given type environment Γ.

The subject reduction property for the simplified system follows by virtue of the equivalence stated in Theorem 1 and by the following two properties:

- $\Gamma \vdash_m \rho \diamond$ implies $\Gamma \vdash_f \rho \diamond$;
- $\Gamma \vdash_f \tau_f' \leq \tau_f$, $\tau_f \sim_\Gamma \tau_m$, $\tau_f' \sim_\Gamma \tau_m'$ implies $\Gamma \vdash_m \tau_m' \leq \tau_m$.

4 Conclusion

We have presented a core calculus for Java exceptions, defined its reduction semantics and two provably equivalent type systems, and proved subject reduction for it.

$$\frac{\begin{array}{l}\Gamma \vdash I \ \text{is}_i\ MST \quad \Gamma \vdash MST\diamond_{\text{InterfaceType}} \\ \Gamma \vdash I_1 : \emptyset\ MST_1 \ldots \Gamma \vdash I_n : \emptyset\ MST_n\end{array}}{\Gamma \vdash I : \emptyset\ (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n)[MST]_\Gamma} \quad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \{I' | I <_i^1 I' \in \Gamma\}\end{array}$$

Set $MST_c = (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n \overset{\Gamma}{\oplus} \mathit{Abstract}(MST'))[\mathit{NonAbstract}(MST')[MST]_\Gamma]_\Gamma$,

$$\frac{\begin{array}{l}\Gamma \vdash C \ \text{is}_c\ K\ FST\ MST \\ \Gamma \vdash CT\diamond_{\text{ClassType}} \\ \Gamma \vdash C <_c^1 C' \\ \Gamma \vdash C' : FST'\ MST' \\ \Gamma \vdash I_1 : \emptyset\ MST_1 \ldots \Gamma \vdash I_n : \emptyset\ MST_n\end{array}}{\Gamma \vdash C : FST'[FST]\ MST_c} \quad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \{I | C \lhd_i^1 I \in \Gamma\} \\ K = \texttt{concrete} \Rightarrow \mathit{Kind}(MST_c) = \texttt{concrete}\end{array}$$

Fig. 3. Type assignments to classes/interfaces for both systems

The first important contribution of this paper is the full formalization of the complex rules given in [5] for compatibility checks of `throws` clauses; such rules have to be performed whenever a class/interface is extended (by inheritance) and are particularly nasty when abstract classes and implementation of interfaces are involved.

The second contribution is the definition of a minimal type system which is proved to be equivalent to that given in [4] and has the advantage of avoiding redundancies in favor of a better understanding of exception handling in Java.

Furthermore, our analysis has pointed out that in some cases the rules defining the static correctness of Java programs can be very tricky. Our feeling is that this could be avoided at the cost of a minimal loss of flexibility, by adding some restriction to the language, as sketched below.

In Java if a class C is declared to implement an interface I, then all methods in I that are neither defined in C nor inherited from the superclasses of C are implicitly inherited by C. This rule implies that a class can inherit more methods with the same signature, a rather strange situation in a language where classes cannot have more than one parent.

A more coherent choice would consist in requiring C to have all methods (either defined or inherited from its superclasses) contained in I, with the consequence that C cannot inherit methods from I but only implement them (as properly suggested by the keyword `implement`). Following this approach, we would also avoid the counter-intuitive Java rule stating that an abstract method m in C implements a method m in I only if m is directly defined in C (see [5] 8.4.6.1 and 8.4.6.4).

References

1. D. Ancona, G. Lagorio, and E. Zucca. A core calculus for Java exceptions. Technical report, DISI, University of Genova, 2000. In preparation.
2. D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of Java with mixins. In E. Bertino, editor, *ECOOP 2000 - European Conference for Object-Oriented Programming*, Lecture Notes in Computer Science. Springer Verlag, 2000. To appear.
3. S. Drossopoulou and T. Valkevych. Java exceptions throw no surprises. Technical report, Dept. of Computing - Imperial College of Science, Technology and Medicine, March 2000.
4. S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Technical report, Dept. of Computing - Imperial College of Science, Technology and Medicine, October 1999.
5. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
6. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.

A Demonstration of Data Inconsistency in the Java Memory Model

V. Kotrajaras, S. Eisenbach

A demonstration of data inconsistency in the Java memory model

Vishnu Kotrajaras and Susan Eisenbach
Department of Computing, Imperial College

March 2000

Abstract

Threads in Java are not constrained to produce sensible run-time behaviour. It is possible (accidentally) to write programs with very undesirable properties. Previously we have formalized Java's multi-threaded behaviour. From our formalization, we have developed a set of applets to demonstrate the characteristics and effects of unsafe behaviours regarding the usage of shared variables, as well as the additional effects of synchronization and volatile declarations. Using synchronization prevents this level of unsafe properties, but using volatile declarations does not help prevents these unsafe properties.

1 Introduction

The Java programming language has built-in support for concurrency. A Java program can spawn threads to work concurrently. The Java Language Specification [1] defines multi-threaded Java environments to consist of a single main memory (accessible by all threads) and threads. Every thread in Java has two components. The first is its *execution engine*, which runs the Java code of each thread. The other component is the thread's *working memory*. Each thread keeps its own *working copy* of shared variables that it must access, in its working memory. As the thread executes a Java program, it operates on these working copies. The main memory contains the *master copy* of every shared variable. Furthermore, the main memory contains *locks*. There is one lock associated with each object. Threads may compete to acquire a lock for mutual exclusion of an object. There are four categories of atomic actions that a thread can perform:

- *use*: transfers the contents of the thread's working copy of a variable to the thread's execution engine.
- *assign*: transfers a value from the thread's execution engine into the thread's working copy of a variable.
- *load*: puts the value transmitted from the main memory by a *read* action into the thread's working copy of that variable.
- *store*: transfers the working copy of a variable from a thread's working memory to the main memory, to be used by a later *write* action.

The main memory manager can perform the following atomic actions:

- *read*: transmits the contents of the master copy of a variable from the main memory to a thread's working memory to be used later by a *load* action.
- *write*: puts a value transmitted by a *store* action into the master copy of a variable in the main memory.

A thread and the main memory can jointly perform these atomic actions:

- *lock*: causes a thread to attempt to acquire one claim on a lock of a particular object. Only one thread at a time is permitted to lay claim to a lock. A thread that fails to acquire a lock will be blocked at that point, waiting to compete for the lock again when the lock is released. Moreover, a thread may acquire the same lock multiple times and does not relinquish ownership of it until a matching number of *unlock* actions have been performed.
- *unlock*: causes a thread to attempt to release one claim on a lock of a particular object. A thread is not permitted to unlock a lock it does not own.

It is crucial that programmers understand how these actions work in order both to predict correctly the behaviour of concurrent programs and to implement correct concurrent programs. Moreover, it is crucial for implementors to follow the actions' definitions in order to implement a correctly behaved runtime system of which behaviour can be formally proven. How these actions work are stated (as English sentences) in the language specification. Understanding everything from words descriptions is difficult and can be ambiguous. Therefore we formalized the descriptions. Our papers on the formalization [2, 3] can be found on our website [4].

Figure 1 shows our graphical representation of what can happen to the memory model.

Notice that in this model formalization, each *store* action goes to a distinct buffer. The same characteristic is also there for each *read* action. The reason for choosing to model these actions this way is to allow maximum flexibility of actions in the model, while still following descriptions from The Java Language Specification [1]. The Language Specification itself does not have any clear description on this issue.

From our model, we formulate a safety property of the Java memory model. We call this property 'data consistency' (This property is mentioned as 'consistent visibility' in Doug Lea's book [5]). 'Inconsistent' data usage can occur when more than one thread [6] shares the same piece of data. Shared data is considered 'consistent' when:

- For each variable V, the *use* action of a thread always uses the value of V that comes from an assignment of V of which serialization order [7] is nearest (and before) that *use* action. (Or less formally, the latest assigned value will always be used.)
- If in the serialized execution sequence, an *assign* on V (say, 'assign1') comes before another *assign* on V ('assign2'), then the write that corresponds to 'assign1' must occur in the serialization before the write that

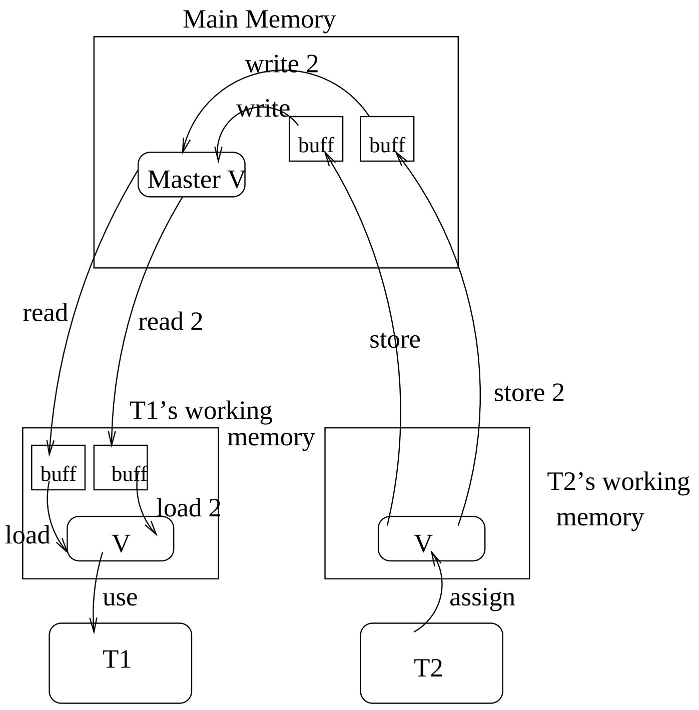


Figure 1: The memory model & its actions

> corresponds to 'assign2'. (Or less formally, the latest assigned value will always produce the latest write to the main memory.)

and 'inconsistent' otherwise.

From the formalization, we divide data inconsistency into the following categories:

- A thread does not *load* a value before performing a *use* action.
- The *write* action of a thread does not occur before the *read* action initiated by another thread, even though the corresponding *assign* occurs before *use*.
- The *write* actions do not serialize corresponding to the *assign* actions.

All these can happen under the current Java memory model. These inconsistent usages cause problems different from the thread interference problem. The thread interference problem can be seen in the following execution ordering:

```
1    thread1    temp = v;
2    thread2    temp = v;
3    thread1    v = temp + 1;
4    thread2    v = temp + 1;
```

where v has lost one of its updates.

But our inconsistency situations tell us that v can lose its update even though the statement execution appears correctly ordered, for example, in the following situation:

```
5    thread1    temp = v;
6    thread1    v = temp + 1;
7    thread2    temp = v;
8    thread2    v = temp + 1;
```

can still lose its update because v from code line 6 may not have been written to Java's main memory before thread 2, in line 7, start reading v from the main memory.

The atomic actions: *use*, *assign*, *load*, *store*, *read*, *write*, *lock* and *unlock* are all very low level. It could be argued that at this level there should be complete freedom and therefore the data inconsistency behaviour is not surprising.

At a higher level, the Java Language Specification [1] defines a collection of rules controlling how a thread's working memory should interact with the main memory. Apart from governing ordinary variables, the rules are about synchronization and volatile declarations.

Synchronization is a thread's attempt to gain an exclusive access to a lock. Synchronization is performed by *lock* and *unlock* actions. During its exclusive access to a lock, before a thread is to perform an *unlock* action, it must first copy all assigned values in its working memory back out to the main memory. Also, any shared variable must be assigned or loaded from the main memory before use. Synchronization, despite being a safe way to handle shared variables, is a very inefficient mechanism due to its forced locking. Therefore Java creators create another mechanism that seems to be more efficient and safe enough in some cases, called volatile declarations. If a variable is declared to be volatile, a thread must reconcile its working copy of the field with the master copy every time it accesses(use/assign) that variable. The main memory actions initiated by the same thread on volatile variables must also be ordered according to their working memory counterparts. The use of volatile does not lock an object and hence it seems much more efficient than synchronization.

2 Test program

We wanted to find out whether the data inconsistency is only in The Java Language Specification or whether it is a property of Java implementation as well. We therefore implemented a test program to see whether our sequence of data inconsistent actions really does happen in actual Java [8] programs. The test program uses these components:

- Thread A: continuously performs assignment to a variable for finite number of times. Each assignment is unique. Thread A records the processor time just after each assignment (In Java, it is not possible to obtain the processor time at the actual *assign/use* action itself).
- Thread B: continuously uses the same variable assigned by thread A for finite number of times. It records the processor time just before each use.
- The main program, which invokes both threads.
- The analyzer program: Analyzes data collected from thread A and thread B. Our analysis is quite simple:

1. First, for each *use* action, we find the earliest possible *assign* that the *use* can take the value from.
2. We then use that point for starting the matching search to find if any *assign* did assign the value that *use* action has read.
3. If no *assign* contains the value that the *use* action has, then it means that the *use* action used the value from earlier than the earliest possible assignment. This is our scenario for data inconsistency.

Figure 2 shows how our program works. By running the above programs on JDK 1.2, we can produce a situation demonstrating data inconsistency.

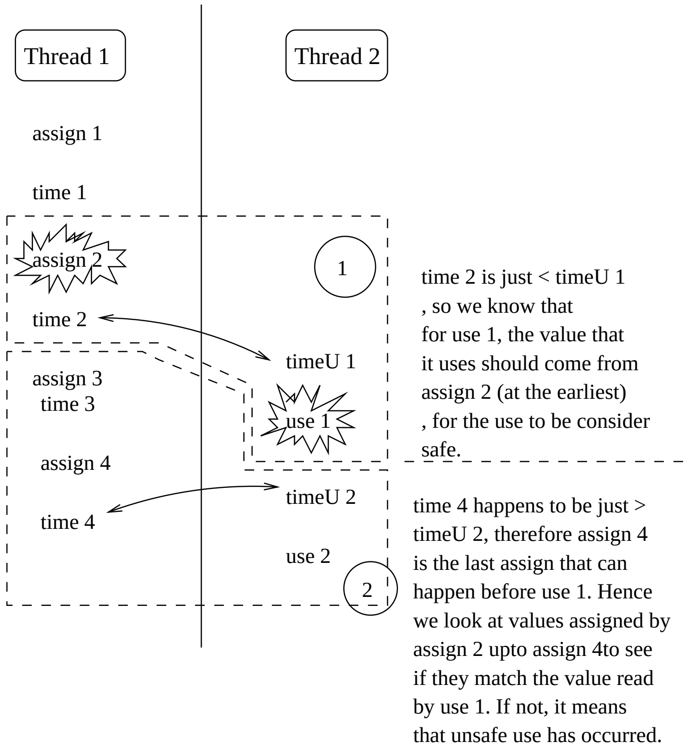

Thread 1
Thread 2
assign 1
time 1
assign 2
1
time 2
timeU 1
assign 3
time 3
use 1
assign 4
timeU 2
time 4
use 2
2
time 2 is just < timeU 1
, so we know that
for use 1, the value that
it uses should come from
assign 2 (at the earliest)
, for the use to be consider
safe.
time 4 happens to be just >
timeU 2, therefore assign 4
is the last assign that can
happen before use 1. Hence
we look at values assigned by
assign 2 upto assign 4to see
if they match the value read
by use 1. If not, it means
that unsafe use has occurred.

Figure 2: program detecting data inconsistency

Using the same implementation, but declaring the shared variable volatile, we can also produce a situation showing data inconsistency. Therefore it is clear that data inconsistencies occur despite volatile declarations.

3 Demonstration applets

The behaviour of concurrent threads, shown by our formal analysis, is not obvious to see and to describe. Understanding this kind of problem well, requires 'seeing' the problem in action. A form of visualization [9, 10] will be of great aid for understandability, hence we have developed demonstration programs following the formalized rules of the formal memory model. These demonstration programs are not in any way related to test programs mentioned in section 2.

While the test programs test the Java implementation, the demonstrations programs were entirely based on our memory model formalization, with the purpose of showing the formal model only. We implemented the demonstrations as a collection of Java applets. These applets can be found at [4].

1. Applet 1: showing a *use* action using the unsafe piece of data, as derived from the formal model.
2. Applet 2: showing that two assign actions can put values in the master copy of a variable in an unsafe order.
3. Applet 3: showing that synchronization solves problem of the first applet.
4. Applet 4: showing that synchronization solves problem of the second applet.
5. Applet 5: showing that volatile does not really solve the problem of the first applet, unlike synchronization.
6. Applet 6: showing that the use of volatile does not solve the problem for unsafe ordering of writes.

All of the applets share a common interface. A description of the displayed applet is always shown on the right frame of any applet page. The component of an applet includes:

- Two threads, $T1$ and $T2$, each perform an *assign* or a *use* action on a shared variable V. The value of each thread's working memory is displayed, unless that thread is to perform an *assign* action. A thread performing an *assign* action will have a textbox for a user to enter the desired new value of V before carrying out the *assign* action.
- The main memory, with the initial master copy of V.
- There are three 'action control' buttons associated with each thread. These buttons allow a user to manipulate the order of occurrences of actions associated with *assign* and *use*. A user can click any button, in any order, to force the execution of the action shown on the button. (The user interface may or may not allow the action, according to our formalized Java rules.)
- The 'Sequence' textbox records the interleaved sequence (serialization) of actions already taken place.
- The 'Result' textbox shows whether a finished sequence preserves data consistency.
- A 'Reset' button clears all elements of the applet to their original status.

Figure 3 shows an applet when start running. At this state, the master copy of V (in the main memory) has value 'MMM'. T2 has 'LLL' as initial value of its copy of V (a help page on the right side of the applet explains this in more detail). All our formal rule violations are handled by popping warning messages, asking the user to make another choice, as seen in figure 4.

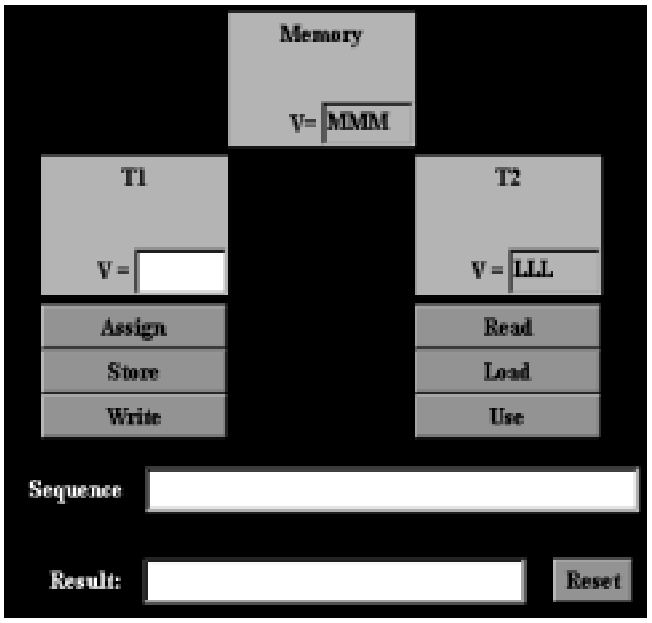
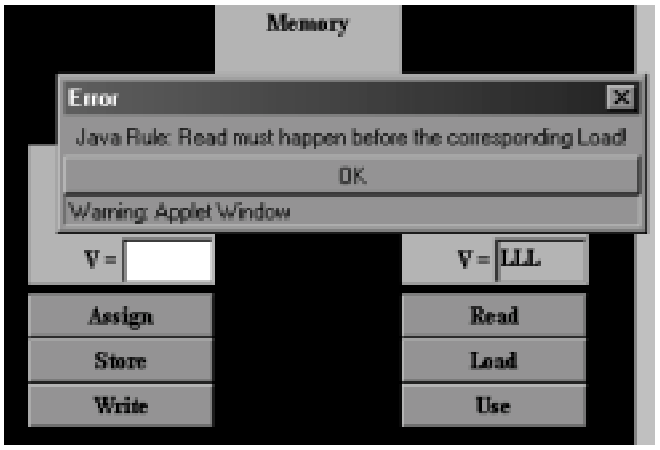


Figure 3: applet when start



Java Rule: Read must happen before the corresponding Load!

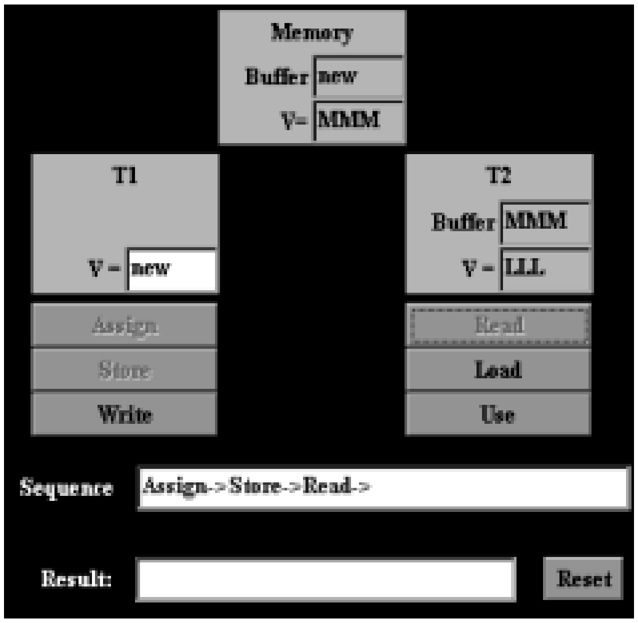
Warning: Applet Window

Figure 4: error message from formal Java rule violation

As an example of how our demonstration applets work, let us use an applet to show an execution sequence that displays execution that a thread does not safely use a shared volatile variable (applet 5).

We use the program to show the second category of unsafe data usage (mentioned in section 1), that is when the *write* action of a thread does not occur before the *read* action initiated by another thread (but the corresponding *assign* occurs before *use*). The first category of unsafe data usage does not happen under volatile because it forces *load* action to take place with every *use* action.

Let's say T1 already carried out 'assign' and 'store'. We now click 'read'. It can be seen (figure 5) that even though 'write' may be carried out right after 'read', there is no way that 'new' can be used by T2 because 'read' already obtained the old value 'MMM'. There are many possible execution sequences that can lead to this situation, such as 'read', 'assign', 'load', 'use'.



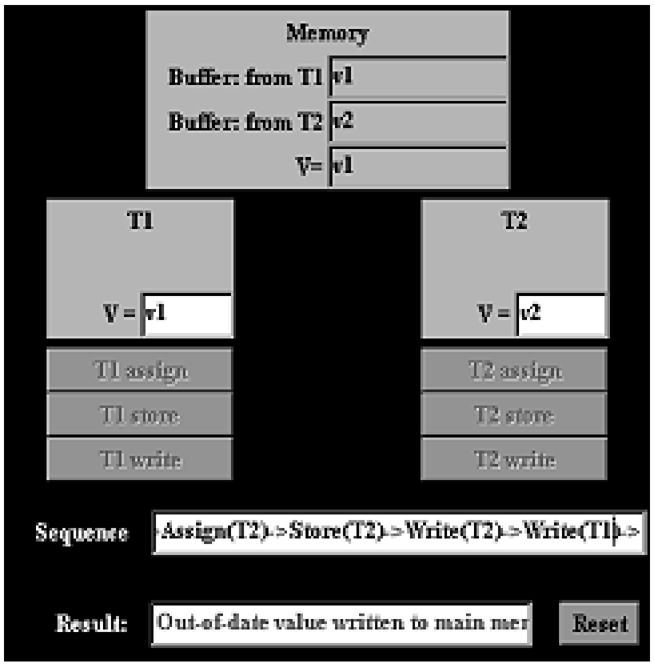
Sequence Assign->Store->Read->

Figure 5: unsafe data usage, write occurs too late

We now move on (applet 6) to show that volatile declarations still allow out-of-order writes by attempting to force 'T1 assign' click before 'T2 assign' but 'T1 write' after 'T2 write'.

To start, the 'T1 assign' is clicked. We now try to find the way to click 'T2 write' before 'T1 write'. Such execution can be achieved. The applet allows a sequence like 'T1 assign', 'T2 assign', 'T1 store', 'T2 store', 'T2 write', 'T1 write'. This execution sequence shows that 'T1 write' occurs after 'T2 write' although their assignments are in opposite order (see figure 6).

All the applets, including the effect of synchronization, can be accessed at [4].



Buffer: from T1 v1
Buffer: from T2 v2
V= v1
V = v1
V = v2
Sequence Assign(T2)->Store(T2)->Write(T2)->Write(T1)->
Result: Out-of-date value written to main mer

Figure 6: volatile declarations allow unsafe ordering of writes across threads

4 Conclusion

In this work, we illustrate a property of the Java memory model that is unsafe for shared variable access, based on our formalization.

There is one aspect that we did not include in our model. There are methods, in the Java Language, that support threads coordination. These methods are 'wait', 'notify' and 'notifyAll', and they operate on the waitset associated with an object. We left the waitset out of our formalization to simplify our model. And because the use of methods involving the waitset is always for synchronization, the behaviour to be analyzed is thus the behaviour of synchronization.

A possible future work is to code the model into a program analyzer tool (One such tool that can produce action based model is LTSA [11]). Using the tool, the model can be exhaustively analyzed, ensuring that everything is covered. Another possible direction for future work is to compare our formal model with the Java virtual machine. The Java virtual machine has various components and operations for managing shared variables. Some components and operations seem to be working very differently from what are described in the Java Language Specification [1]. The question arises whether the two models are actually compatible.

5 About the Authors

Susan Eisenbach is Director of Studies in the Department of Computing at Imperial College. She was principal investigator of the Multimedia Network Application (BECALM) project funded by the UK Engineering and Physical Science

Research Council (EPSRC), where she worked on language design for multimedia applications in large-scale distributed systems. She is co-investigator of the EPSRC Systems Engineering project SLURP investigating Java semantics. Susan Eisenbach was program chair of the OOPSLA'98 Workshop on Formal Underpinnings of Java Semantics.

Vishnu Kotrajaras obtained his Master in Engineering at Imperial College, London, in 1997. He is now a PhD student in the SLURP project in the Department of Computing at Imperial College.

References

[1] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, August 1996.

[2] Vishnu Kotrajaras and Susan Eisenbach. Threads and Main Memory Semantics. In *Workshop on Formal Techniques for Java Programs, ECOOP 99*, June 1999.

[3] Vishnu Kotrajaras. A demonstration of data inconsistency of the Java memory model (full paper). web site: http://www.formaljava.fsnet.co.uk, January 2000.

[4] Vishnu Kotrajaras. Demonstration program for formal study of concurrent Java. web site: http://www.formaljava.fsnet.co.uk, 1999.

[5] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison Wesley, 1999.

[6] Scott Oaks and Henry Wong. *Java Threads.* O'Reilly, January 1997.

[7] Alex Gontmakher and Assaf Schuster. Java Consistency: Non-Operational Characterizations for Java Memory Behaviour. Technical Report CS0922, Computer Science Department, Technion, November 1997.

[8] Mark Grand. *Java Language Reference.* O'Reilly, second edition, July 1997.

[9] Andrea Lawrence. Empirically Evaluating the Use of Animations to Teach Algorithms. Technical Report GIT-GVU-94-07, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, 1994.

[10] John Stasko. Using Student-Built Algorithm Animations as Learning Aids. Technical Report GIT-GVU-96-19, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, August 1996.

[11] Jeff Magee and Jeff Kramer. *Concurrency: state models and Java programs.* Wiley, 1999.

Verified Lightweight Bytecode Verification

G. Klein, T. Nipkow

Verified Lightweight Bytecode Verification

Gerwin Klein and Tobias Nipkow

Technische Universität München, Institut für Informatik
`http://www.in.tum.de/~{kleing|nipkow}/`

Abstract. Eva and Kristoffer Rose proposed a (sparse) annotation of Java Virtual Machine code with types to enable a one-pass verification of welltypedness. We have formalized a variant of their proposal in the theorem prover Isabelle/HOL and proved soundness and completeness.

1 Introduction

The Java Virtual Machine (*JVM*) comprises a typed assembly language, an abstract machine for executing it and the so-called *Bytecode Verifier* (*BV*) for checking the welltypedness of JVM programs. Resource-bounded JVM implementations on smart cards do not provide bytecode verification because of the relatively high space and time consumption. They either do not allow dynamic loading of JVM code at all or rely on cryptographic methods to ensure that bytecode verification has taken place off-card. In order to allow on-card verification, Eva and Kristoffer Rose [3] proposed a (sparse) annotation of Java Virtual Machine code with types to enable a one-pass verification of welltypedness. Roughly speaking, this transforms a type reconstruction problem into a type checking problem, which is easier. Based on these ideas we have extended an existing formalization of the JVM in the theorem prover Isabelle/HOL [2, 1]. In §2 we describe the general idea of bytecode verification and its formalization in Isabelle/HOL. In §3 we explain how lightweight bytecode verification works, how we formalized it and proved it correct and complete.

2 Bytecode verification

The JVM is a stack machine where each method activation has its own expression stack and local variables. The types of operands and results of bytecode instructions are fixed (modulo subtyping), whereas the type of a storage location may differ at different points in the program. Let's look at an example:

instruction	stack	local variables
Load 0	[]	[Class B, int]
Store 1	[Class A]	[Class B, unusable]
Getfield F A	[]	[Class B, Class A]
Goto -2	[Class A]	[Class B, Class A]

On the left the instructions are shown and on the right the type of the stack elements and the local variables. The type information attached to an instruction characterizes the state *before* execution of that instruction. We assume that class B is a subclass of A and that A has a field F of type A.

Execution starts with an empty stack and the two local variables hold a reference to an object of class B and an integer. The first instruction loads local variable 0, a reference to a B object, on the stack. The type information associated with following instruction may puzzle at first sight: it says that a reference to an A object is on the stack, and that the type of local variable 1 has become unusable. This means the type information has become less precise but is still correct: a B object is also an A object and an integer is now classified as unusable. The reason for these more general types is that the predecessor of the Store instruction may have either been Load 0 or Goto -2. Since there exist different execution paths to reach Store, the type information of the two paths has to be "merged". The type of the second local variable is either int or Class A, which are incompatible, i.e. the only common supertype is 'unusable'.

Bytecode verification is the process of inferring the types on the right from the instruction sequence on the left and some initial condition, and of ensuring that each instruction receives arguments of the correct type. This can be done on a per method basis because each method has fixed argument and result types. The two tables on the right are together called a *method type*, one line of the method type is called a *state type*. To simplify matters we restrict the considerations in this paper to a single method.

For theoretical investigations it has become customary to separate type inference (computation of a method type) from type checking (checking if an instruction sequence fits a method type). Type inference is usually implemented as a dataflow analysis and may require several iterations due to subtyping. We will now ignore type inference (although we have also verified it in Isabelle/HOL) and concentrate on type checking.

A first machine-checked specification of type checking for the JVM was given by Pusch [2]. Using Isabelle/HOL she connected the type checking rules with an operational semantics for the JVM by showing that execution of type correct programs is type sound, i.e. during run time each storage location contains values of the type predicted by the method type. We will now sketch some the key ingredients of the type checking specification by Nipkow *et al.* [1] that our formalization of lightweight bytecode verification builds on.

Type checking of methods is modeled by a predicate wt_method relating the instruction sequence, types of method parameters, return type, etc. with a method type φ. In essence, the definition

$$
\begin{array}{l}
\mathsf{wt\_method} :: [\mathit{jvm\_prog}, \mathit{cname}, \mathit{ty\ list}, \mathit{ty}, \mathit{nat}, \mathit{instr\ list}, \mathit{method\_type}] \rightarrow \mathit{bool} \\
\mathsf{wt\_method}\ \Gamma\ C\ pTs\ rT\ mxl\ ins\ \varphi \equiv \\
\quad \mathsf{let}\ \mathit{max\_pc} = \mathsf{length}\ \mathit{ins}\ \mathsf{in} \\
\qquad \mathit{max\_pc} < \mathsf{length}\ \varphi \wedge 0 < \mathit{max\_pc} \wedge \mathsf{wt\_start}\ \Gamma\ C\ pTs\ mxl\ \varphi\ \wedge \\
\qquad (\forall pc.\ pc < \mathit{max\_pc} \longrightarrow \mathsf{wt\_inst}\ (\mathit{ins}\ !\ pc)\ \Gamma\ rT\ \varphi\ \mathit{max\_pc}\ pc)
\end{array}
$$

states that, in a declaration context Γ (containing all class declarations of the program), wt_method holds for an instruction sequence *ins* (the method body) and a method type φ when each single instruction *ins* ! *pc* is well typed (the Isabelle/HOL operator ! returns the nth element of a list). The predicate wt_inst checking single instructions may take into account the return type rT, the current program counter *pc*, and the maximum program counter *max_pc* (the length of the instruction sequence). wt_start ensures that the types on the operand stack and of the local variables are initialized correctly with regard to the class C the method is declared in, the parameters pTs of the method, and the number of local variables mxl.

wt_inst is a case distinction over the instruction set. As the type checking conditions for single instructions are very similar to each other, we only take a look at an example:

```
wt_inst :: [instr,jvm_prog,ty,method_type,nat,nat] → bool
wt_inst (Load idx) Γ rT φ max_pc pc =
    let (ST,LT) = φ ! pc in
        pc+1 < max_pc ∧ idx < length LT ∧
        (∃t. (LT ! idx) = usable t ∧ Γ ⊢ (t # ST , LT) ≼_s φ ! (pc+1))
```

The predicate first checks some applicability conditions like $pc + 1 < max\_pc$ and $idx <$ length LT, then calculates the effect of the instruction on the current state type and eventually requires that the result be compatible with the state type at the next instruction in the control flow.

The current state type consists of the stack ST and the local variables LT at position *pc* in the method type. Both are lists containing the types before execution of the instruction. In the Load case we require some type t other than unusable at index *idx* in LT. The state type of the next instruction at position $pc + 1$ must correctly approximate a state type where t is on top of the stack (# is the list constructor in Isabelle/HOL). The local variables are unchanged. This correct approximation $\_ \vdash \_ \preceq_s \_$ is Java's *widen* relation lifted to state types and extended by the element unusable. We already used it informally in the example program.

3 Lightweight bytecode verification

Two things make the traditional bytecode verifier unsuitable for on-card verification: the type reconstruction algorithm itself is large and complex, and the whole method type is held in memory. Lightweight bytecode verification addresses both problems.

The need for dataflow analysis is caused by the fact that some instructions may have multiple preceding paths of execution and that the types constructed on these paths have to be merged. This can only occur at the targets of jumps. The basic idea of lightweight bytecode verification is to look what happens when we provide the result of the type reconstruction process at these points beforehand. This additional outside information is called the *certificate*. It becomes apparent that the type reconstruction is now reduced to a single linear pass over the instruction sequence: each time we would have to consider more than one path of execution, the result is already there and only needs to be checked, not constructed. The second effect is that apart from the certificate we only need constant memory: the type reconstruction can be reduced to a function that calculates the state type at $pc + 1$ only from the state type at pc and the global information that is already provided from outside. After having calculated the type at $pc + 1$, we can immediately forget about the one at pc.

For our example program, the situation at the start of the lightweight bytecode verification process looks like that:

instruction	stack local variables
Load 0	
Store 1	[Class A] [Class B, unusable]
Getfield F A	
Goto -2	

From that the whole method type is reconstructed in a single linear pass: The state type ([], [Class B, int]) for the Load instruction will be filled in as initialization. The state type for Store 1 is in the certificate, since Store is the target of the Goto -2 jump. The lightweight bytecode verifier calculates the effect of Load 0, i.e. ([Class B], [Class B, int]), and checks if the certificate ([Class A], [Class B, unusable]) correctly approximates this result. The types before execution of Getfield are then easily calculated from the state type and the effect of Store alone, i.e. the result is ([], [Class B, Class A]). The effect of Getfield F A also only needs the current state type and yields ([Class A], [Class B, Class A]). For the last instruction the lightweight bytecode verifier has to check if the calculated state type is correctly approximated by the jump target. We did not store this state type, but since it is a target of a jump, we have it in the certificate and only need to check if the certificate at this point correctly approximates our calculated state type. Note, that all paths of executions that entered into the merging for the state type of Store 1 were checked, but no iteration or additional memory was required.

3.1 Formalization

With that kind of process and certificate in mind, we can start a formalization of the lightweight bytecode verifier. We have two goals here: On the one hand, we want the formalization to be similar to the one of the traditional bytecode verifier, so we can easily spot commonalities and differences. On the other hand, we now not only want to model type checking, but also the simplified form of type reconstruction, i.e. we want functions, not predicates. As a solution, we write the predicates checking single instructions in a form that is similar to the traditional bytecode verifier, and that can still easily be read as a function. For example the predicate for Load

```
wtl_inst :: [instr,jvm_prog,ty,state_type,state_type,nat,nat] → bool
wtl_inst (Load idx) Γ rT s s' cert max_pc pc =
        let (ST,LT) = s in
                pc+1 < max_pc ∧ idx < length LT ∧
                (∃t. (LT ! idx) = usable t ∧ (t # ST , LT) = s')
```

can be read as a function yielding the next state type s' from the current instruction Load idx, the current state type s, the program counter pc, and the maximum program counter max_pc. Declaration context Γ, return type rT, and the certificate $cert$ are not used in the Load case. The predicate still closely mimics the corresponding wt_inst from the traditional byte code verifier: it is apparent, that we have the same applicability conditions and model the same effect the instruction has on the stack.

We now have a function that calculates the state type s' at $pc + 1$ from the state type s at pc. Iterating this process over the list of instructions we can then feed this s' as current state type to the next instruction:

```
wtl_inst_list :: [instr list,jvm_prog,ty,state_type,state_type,certificate,nat,nat] → bool
wtl_inst_list (i#is) Γ rT s₀ s₂ cert max_pc pc =
        (∃s₁. wtl_inst_option i Γ rT s₀ s₁ cert max_pc pc ∧
              wtl_inst_list is Γ rT s₁ s₂ cert max_pc (pc+1))
```

wtl_inst_option is a simple case distinction: if there is already type information stored in the certificate at the current program counter, as for Store 1 in the example, we must not use our calculated type, but the certificate containing the merged type information instead. To ensure correctness, we still have to check, if the certificate correctly approximates the calculated state type, i.e. if the certificate really is the result of a merge of our state type with another one. Therefore we have:

```
wtl_inst_option :: [instr,jvm_prog,ty,state_type,state_type,certificate,nat,nat] → bool
wtl_inst_option i Γ rT s₀ s₁ cert max_pc pc ≡
    case cert!pc of
        None      → wtl_inst i Γ rT s₀ s₁ cert max_pc pc
      | Some s₀' → (Γ ⊢ s₀ ≼ₛ s₀') ∧ wtl_inst i Γ rT s₀' s₁ cert max_pc pc
```

3.2 Soundness

When we specify a new kind of bytecode verification we of course wish to know if this new bytecode verifier does the right thing. In our case this means: if the lightweight bytecode verifier accepts a piece of code as welltyped, the traditional bytecode verifier should accept it, too. We must also show that it is safe to rely on outside information, i.e. in the soundness proof we must not make any assumption on how the certificate was produced. So the soundness theorem is

$$\forall cert.\ \mathsf{wtl\_method}\,\Gamma\ \ldots\ cert \implies \exists\varphi.\ \mathsf{wt\_method}\,\Gamma\ \ldots\ \varphi$$

where $\Gamma \ldots$ is shorthand for the same set of parameters, return type etc. for both judgments.

This means, that if the certificate was tampered with, the lightweight bytecode verifier either rejects the method as not welltyped, or if it does not reject, it was still able to reconstruct the method type correctly.

We prove this by constructing a φ from a successful run of the lightweight bytecode verifier and showing that this φ satisfies wt_method. φ must have the following properties: if the certificate contains a state type s at some point pc, φ contains that s at the same point pc. Otherwise, if the lightweight bytecode verifier has come to a position pc in its type reconstruction process and has calculated a current state type s, φ will contain that s at position pc.

If wtl_method holds, there clearly always is such a φ. By case distinction over all instructions we get that both bytecode verifiers compute the same effects of instructions on state types, and, because the certificate is always checked to correctly approximate the calculated state type, we get that for each instruction wt_inst holds. Thus the traditional bytecode verifier accepts.

3.3 Completeness

Of course, the trivial bytecode verifier that rejects all programs also would be correct in the sense above. Therefore we show that our lightweight bytecode verifier also is complete, i.e. that if a program is welltyped with respect to the traditional bytecode verifier, the lightweight bytecode verifier will accept the same program with an easy to obtain certificate.

How will this certificate look like? We get the information we need from the method type of a successful run of the traditional bytecode verifier. Since we want to minimize the amount of information we have to provide, we do not take the whole method type as the certificate, but only the state types at certain positions.

As in the example, the certificate should contain the type information at jump targets. Due to some simplifications in our formalization of the traditional bytecode verifier and the μJava language, this is not enough though. The first thing is, our traditional bytecode verifier does not ignore dead code, but requires instructions that can never be executed to be type correct, too. If the instruction directly after a Goto for instance is not a jump target, it can never be executed. Since the effect of Goto on the state type only tells us something about the target of the Goto, but nothing about the state type of the instruction at $pc + 1$, the lightweight bytecode verifier would have no means to construct this state type at $pc+1$ if it wasn't in the certificate. So we also include the state types directly after Goto and Return instructions. Since dead code should be eliminated by the compiler anyway, this is not really an issue. On the other hand, it is not hard to take dead code into account and we plan to do so in the future. We also need the state type after a method invocation in some cases. This is due to the fact that we do not really model exceptions at the JVM level. In μJava, a method invocation on a class reference containing the value null is equivalent to a halt. If the bytecode verifier discovers that this class reference is always null, the instruction after that may again be dead code and we have to include it in the certificate. Again, programs produced by an optimizing compiler should not contain such cases.

So the certificate contains the targets of jumps and some rare cases, we have to include for the completeness proof, because we do not want to make any assumptions about how the code was produced.

Let make_cert be the function that produces such a certificate from an instruction sequence and a method type. Then

$$\mathsf{wt\_method}\,\Gamma\;\ldots\;ins\;\varphi\;\Longrightarrow\;\mathsf{wtl\_method}\,\Gamma\;\ldots\;ins\;(\mathsf{make\_cert}\;ins\;\varphi)$$

follows by induction over the length of the instruction sequence.

4 Conclusion

We have formalized a variant of lightweight bytecode verification for μJava and proved its soundness and completeness in Isabelle/HOL. Our formalization is comparatively easy to transform into a functional program. The completeness result is both stronger and weaker than that of [3]. Eva and Kristoffer Rose have a more complex formalization of the lightweight bytecode verifier that only needs the certificate when a type merge really produces a different type than calculated so far. Doing so could lead to a smaller type annotation of class files (although this claim would require formal proof). It does however not save space during the verification pass, since the state type at jump targets has to be saved for later checks anyway. Our completeness result on the other hand includes the simpler and easier to implement notion that (apart from artificial cases) the targets of jumps are all that is needed for linear type reconstruction.

References

1. T. Nipkow, D. v. Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages ?–? IOS Press, 2000. To appear.
2. C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lect. Notes in Comp. Sci.*, pages 89–103. Springer-Verlag, 1999.
3. E. Rose and K. Rose. Lightweight bytecode verification. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.

Type Safety in the JVM: Some Problems in JDK 1.2.2 and Proposed Solutions

A. Coglio, A. Goldberg

Type Safety in the JVM: Some Problems in JDK 1.2.2 and Proposed Solutions

Alessandro Coglio and Allen Goldberg

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304
{coglio, goldberg}@kestrel.edu

1 Introduction

We are currently developing mathematical specifications for various components of the JVM, including the bytecode verifier [2, 4, 7], the class loading mechanism [8], and the Java 2 security mechanisms. We are also deriving a complete implementation of the bytecode verifier [2] through Specware [10], a system developed at Kestrel Institute that supports provably correct, compositional development of software from formal specifications.

In the course of our formalization efforts, we have uncovered subtle bugs in Sun JDK 1.2.2 that lead to type safety violations. These bugs are in the bytecode verifier and relate to the naming of reference types. We found that in certain circumstances these names can be spoofed by suitable use of delegating class loaders. Since the JVM specification [6] is informal English prose, we cannot crisply characterize these bugs as errors in the specification or just in one or more implementations. However, some of these bugs are consistent with a reasonable interpretation of the specification. We have verified that the bugs exist in Sun JDK 1.2.2 (on both Solaris and Windows NT). Some are fixed in Sun JDK 1.3 Beta by restricting access to system packages.

Overall, these flaws raise the issue of a more precise specification of the bytecode verifier and the loading mechanisms, and increased assurance of type safety properties. Besides fixes for all these bugs, we have devised a more general approach to insuring type safety that has additional advantages, including lazier class loading.

2 Types in the JVM

According to [6], a class in the JVM is identified by its fully qualified name (FQN) plus its defining loader. In fact, classes are in correspondence with instances of class `java.lang.Class`, and the only way to create new `Class` instances is through the `defineClass` methods of class `java.lang.ClassLoader`. These methods call internal JVM code that carries out actual class creation from byte arrays in `classfile` format. This code enforces the constraint that a class loader cannot create two classes with a same FQN. Thus it is possible to identify a class by its FQN plus its defining loader.

However, the bytecode verifier, when verifying a class, essentially uses just FQNs. In a few cases, it actually resolves class names and makes use of the `Class` instances they resolve to. In particular, the bytecode verifier sometimes needs to *merge*, that is find the first common superclass of two class names. The two class names are resolved, thus loading the classes and their superclasses, and their ancestry searched to find their first common superclass. The bytecode verifier also resolves names to check assignment compatibility (i.e., subtype relationship) between two class names.

The use of FQNs and occasional use of actual classes guarantee type safety only under certain assumptions. Examples of these assumptions are the loading constraints introduced in the Java 2 Platform [6, 5] to avoid the type safety problems exactly arising from the violation of the assumptions they enforce [9]. Simply stated, loading constraints ensure that classes exchanging objects (through their methods and fields) agree on the actual types (and not only on the FQNs) of such objects.

As it turns out, loading constraints do not cover all the assumptions needed to guarantee type safety. An example occurs when checking a stack position that contains the type (FQN) that results from merging two classes: the bytecode verifier assumes that the FQN in the stack position resolves (using the defining loader of the analyzed class as the initiating loader) to the actual first common superclass. Another example occurs when checking type constraints for the `invokespecial` instruction: the bytecode verifier assumes that an FQN of any superclass of the current class resolves to the actual superclass. Furthermore the bytecode verifier assumes that the FQNs `java.lang.Object` and `java.lang.String` resolve to the "usual" system classes. It is in fact possible to construct programs where these assumptions are violated, thus causing name spoofing and type safety failures. For reasons of space, here we only describe one of the bugs. Further details, including runnable programs, can be found in [1].

As in [6] and [5] we use the notation N^L to denote the class associated to name N and loader L when loading of N is initiated by L, i.e., L is an initiating loader of N. Furthermore $\langle N, L\rangle$ denotes the (unique) class with FQN N and defining loader L.

3 The merging bug

[6, Sect. 4.9.2] describes how, during data flow analysis of a method's code, the types assigned to stack positions and local variables along different control paths are merged. In order to merge two distinct class names `Sub1` and `Sub2`, the corresponding classes are loaded by invoking the `loadClass` method of the defining loader L of the class whose method is being verified, with argument `Sub1` first, then `Sub2`. The ancestry of the classes $\texttt{Sub1}^L$ and $\texttt{Sub2}^L$ is then searched to find the first common superclass. If the first common superclass found is $\langle \texttt{Sup}, L_0\rangle$ then the bytecode verifier writes the FQN `Sup` in the merged stack position. Suppose that, after the merging point, an instruction accesses a field or method of a class named `Sup` in the field or method reference. Since the bytecode verifier has deduced that the stack position indeed contains a class with FQN `Sup`, the check for assignment compatibility will succeed, as described in [6, Sect. 4.9.1]. This is correct only assuming that $\texttt{Sup}^L = \langle \texttt{Sup}, L_0 \rangle$, i.e., that loading of FQN `Sup` initiated by L results in the actual superclass of $\texttt{Sub1}^L$ and $\texttt{Sub2}^L$.

However, such an assumption can be violated if for example L_0 is the system class loader and L is a user-defined class loader that delegates to L_0 (by invoking `findSystemClass`) the loading of FQNs `Sub1` and `Sub2` but not of `Sup`. The following situation is arranged (`Merger` is the class being verified):

$$\begin{tikzcd}
 & \langle \texttt{Sup}, L_0 \rangle & \\
\langle \texttt{Sub1}, L_0 \rangle \arrow[ur, Rightarrow] & \langle \texttt{Merger}, L \rangle \arrow[l] \arrow[r] \arrow[d] & \langle \texttt{Sub2}, L_0 \rangle \arrow[ul, Rightarrow] \\
 & \langle \texttt{Sup}, L \rangle &
\end{tikzcd}$$

A thin arrow from a class (identified by its name and defining loader) to another indicates that the source of the arrow resolves the FQN of the target to the target. A double arrow indicates that the source is a subclass of the target. Suppose that class `Merger` contains the following code:

```
Sub1 s1 = new Sub1();
Sub2 s2 = new Sub2();
Sup s;
if (s1 != null)  s = s1;  // this test just serves to
else  s = s2;             // create two merging paths
s.m();  // type unsafe!
```

This code passes verification for the reason described above. If $\langle \texttt{Sup}, L \rangle$ has a method `m` with the right descriptor, at runtime the method call goes through because `Sup` resolves to $\langle \texttt{Sup}, L \rangle$. However, the object stored in `s` has class $\langle \texttt{Sup}, L_0 \rangle$ (as well as class $\langle \texttt{Sub1}, L_0 \rangle$). If $\langle \texttt{Sup}, L_0 \rangle$ is different from $\langle \texttt{Sup}, L \rangle$, the effect of the method call is undefined. In typical implementations, it will probably call some unrelated method that happens to have the same index, thus causing type unsafety.

This may be interpreted as a bug in the JVM specification, rather than the implementation. Although [6] does not crisply state that types are denoted by FQNs in the bytecode verifier (it typically just talks about "reference types"), that seems to be the intended meaning, or at least the most reasonable interpretation. In any case, future editions of [6] should clarify this point. This bug also exists in JDK 1.3 Beta.

A possible solution to the problem is to keep information, when merging two FQNs `Sub1` and `Sub2`, about the actual first common superclass $\langle \texttt{Sup}, L_0 \rangle$ (not only its FQN `Sup`). When checking assignment compatibility with the FQN `Sup` (referenced in the runtime constant pool), the FQN is resolved and the resulting `Class` instance is compared with the one obtained from merging. In this way there can be no confusion. Interestingly, inspection of the bytecode verifier code in JDK 1.2.2 shows that information about the actual first common superclass is indeed maintained and accessible. However, it is not used to prevent this problem. Alternatively, to avoid early loading of `Sup` by L, a loading constraint $\texttt{Sup}^L = \texttt{Sup}^{L_0}$ can be added by the bytecode verifier to the set of globally maintained loading constraints.

4 A general solution

As previously mentioned, the bytecode verifier makes use of FQNs, occasionally resolving them to actual classes. This resolution results in premature loading of classes. We now propose a design for the bytecode verifier (and related parts of the JVM) that (1) avoids premature loading and (2) allows a cleaner separation between bytecode verification and loading. This cleaner separation also promotes a better understanding of how bytecode verification and other mechanisms (such as loading constraints) cooperate to insure type safety in the JVM.

In the design we propose, the bytecode verifier uniformly uses FQNs, never actual classes. The intended disambiguation is that FQN N stands for class N^L, where L is the defining loader of the class under verification (note that, at verification time, class N^L might not be present in the JVM yet). The bytecode verifier never causes resolution (and loading) of any class.

The result of merging two FQNs is a set containing the two FQNs. More precisely, the bytecode verifier uses (finite) sets of FQNs (and not just FQNs) to type stack positions and local variables containing reference types [2, 4, 7]. Initially (e.g., in the local variables containing method invocation arguments) sets are singletons. Merging is set union. The meaning of a set of FQNs typing a local memory is that the local memory may contain an instance of a class whose FQN is in the set. No relationship among the elements of the set is intended.

When a set of FQNs is checked for assignment compatibility with a given FQN N, for each element M of the set different from N, a subtype loading constraint $M^L < N^L$ is generated. The meaning of such constraint is that class M^L must be a subclass of class N^L. The constraint is added to the global state of the JVM, and checked for consistency with the loaded class cache. If either class has not been loaded yet, the constraint is just recorded. Whenever the loaded class cache is updated, it is checked for consistency with the current subtype loading constraints. This is very similar to the equality loading constraints of the form $N^L = N^{L'}$ introduced in the Java 2 Platform. In fact, subtype constraints complement equality constraints.

Checking the consistency of the loaded class cache and loading constraints that include both subtype constraints and equality constraints is neither difficult nor inefficient. A naïve algorithm will transitively close both subtype and equality constraints and then check that when the loaded class cache is updated none of the constraints in the transitive closure is violated. An efficient algorithm will use a union-find data structure to store equivalence classes of classes asserted to be the same and track the asserted subtype dependencies of the classes.

In this design, the result of bytecode verification of a class is therefore not just a yes/no answer, but also a set of subtype constraints that explicitly and clearly express the assumptions made by the bytecode verifier to certify the class. Furthermore, the bytecode verifier is a well-defined, purely functional piece of the JVM that does not depend on the current state of JVM data structures.

Let us now see how this approach avoids the merging bug. When verifying the code in `Merger`, the creation (and initialization) of the two instances of class `Sub1` and `Sub2` has the effect of typing the local variables as $\{\texttt{Sub1}\}$ and $\{\texttt{Sub2}\}$. After the merging point, the type on top of the stack is $\{\texttt{Sub1}, \texttt{Sub2}\}$. Since the call of method `m` references class `Sup` (through the costant pool), subtype constraints $\texttt{Sub1}^L < \texttt{Sup}^L$ and $\texttt{Sub2}^L < \texttt{Sup}^L$ are generated. When the code is eventually executed, before the method is called all of $\texttt{Sub1}^L$, $\texttt{Sub2}^L$, and $\texttt{Sup}^L$ will have been loaded. Since subtype constraints are violated, the JVM will throw an exception preventing resolution of the method (and therefore its invocation).

Our approach also allows a cleaner treatment of interface types in the bytecode verifier. Since an interface can have more than one superinterface, two given interfaces may not have a unique first common superinterface. According to [6], the result of merging two interface FQNs is therefore `java.lang.Object`, which is indeed a superclass of any interface. However, this requires a special treatment of `java.lang.Object` when checking its assignment compatibility with an interface FQN: the bytecode verifier just passes the check because `java.lang.Object` might derive from merging interfaces, even though `java.lang.Object` itself is not assignment-compatible with an interface. This "looseness" does not cause type unsafety because the `invokeinterface` instruction performs a search of the methods declared in the runtime class of the object on which it is executed. If no method matching the referenced descriptor is found, an exception is thrown. This runtime check does not impose any additional runtime penalty. Our scheme is cleaner in that it provides a uniform treatment of classes and interfaces.

In [8] we provide formal arguments that this design of the bytecode verifier, together with (subtype and equality) loading constraints, guarantees type safety in the JVM. In that paper we formalize the operational semantics of a simplified JVM that includes class loading, resolution, bytecode verification, and execution of some instructions, and we prove type safety results about it.

Our approach of having a self-contained bytecode verifier that generates constraints is similar in spirit to [3]. However, they do not consider multiple class loaders. Their bytecode verifier generates, besides subtype constraints, several other kinds of constraints, e.g. for fields and methods referenced in the code being verified. We only generate subtype constraints because the others can be checked at runtime (as specified in [6]) without performance penalty or premature loading.

References

1. Alessandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in JDK 1.2.2 and proposed solutions. `http://www.kestrel.edu/java`, 2000.
2. Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Towards a provably-correct implementation of the JVM bytecode verifier. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, 1998.
3. Philip W. L. Fong and Robert D. Cameron. Proof linking: An architecture for modular verification of dynamically-linked mobile code. *ACM SIGSOFT Software Engineering Notes*, 23(6):222–230, November 1998. Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering.
4. Allen Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security*, 1998.
5. Sheng Liang and Gilad Bracha. Dynamic class loading in the Java™ virtual machine. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44. ACM Press, 1998.
6. Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
7. Zhenyu Qian. A formal specification of Java™ virtual machine instructions for objects, methods and subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java™*. LNCS 1523, Springer-Verlag, 1998.
8. Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. `http://www.kestrel.edu/java`, 2000.
9. Vijay Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997. `http://www.research.att.com/vj/bug.html`.
10. Yellamraju Srinivas and Richard Jüllig. Specware: Formal support for composing software. In B. Moeller, editor, *Proceedings of the Conference on Mathematics of Program Construction*, pages 399–422. LNCS 947, Springer-Verlag, Berlin, 1995.

Formalization in Coq of the Java Card Virtual Machine

G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. Sousa, S. Yu

Formalization in Coq of the Java Card Virtual Machine

Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard Serpette, Simão Sousa, and Shen-Wei Yu

Projet Oasis
INRIA Sophia-Antipolis
France
e-mail: `FirstName.LastName@sophia.inria.fr`

1 Introduction

Java has quickly become a standard for Internet and, through Java Card, for smartcard programming, putting security issues at stake. Yet recent research has unveiled several problems in the Java Card security model, most notably with object sharing and the associated mechanism of shareable interfaces, see e.g. [10, 17]—there are further security breaches related to some Java features not present in Java Card but we are not concerned with them here. It is therefore fundamental to develop environments to verify the security of the Java Card platform and of Java Card programs. Thus far Java/Java Card security has been studied at two main levels:

1. platform level: here the goal is to prove safety properties of the language, in particular type safety and properties related to memory management. This first level may be instrumented with proof assistants, which are well-suited to formalization and logical reasoning;
2. application level: here the goal is to prove that a specific program obeys a specific security policy, such as the sandbox security policy or information-flow based security policies. This second level may be instrumented with a variety of tools, such as model-checkers, which allow to perform automatically a symbolic evaluation of programs to check whether a given program adheres to a given security policy.

The long-term goal of our work, part of which is being carried in the context of the *Action de Recherche Coopérative* S-Java (see `http://www-sop.inria.fr/oasis/SJava`), is to provide an environment that integrates both layers and allows to prove security properties for Java Card programs. In the present article, we set up the foundations for such an environment by providing:

1. an executable formal specification in Coq of a substantial fragment of the JCVM and JCRE. The specification is written in functional style and formalizes one step execution of the JCVM by a function `exec_jcvm` of type
   ```
   jcvm_program -> jcvm_state -> (Exc jcvm_state)
   ```
   where `jcvm_state` is the type of JCVM states (see Section 2), and `Exc` is the Coq representation of the lift monad. Our specification is rather precise: in particular, it implements the mechanisms of firewalls and shareable interfaces upon which Java Card security crucially relies;
2. a tool, written in Java, that translates CAP files into their Coq representation. In order to achieve a compact and readable representation of Java Card programs, the CAP tool performs various program transformations, including linking and index manipulation, and may be viewed as a fair part of a bytecode verifier.

The remaining of the abstract is organized as follows: in Section 2, we present our formalization of Java Card programs and of the JCVM and JCRE. In Section 3, we present the CAP tool that translates CAP files into their Coq formalizations. We conclude in Section 4 with related work and perspectives for future research.

2 Formalization of the JCVM

2.1 Java Card programs

Java Card programs are formalized as a record of lists of classes, methods and interfaces:

```
  Record jcvm_program : Set := {
         classes    : (list Class);
         methods    : (list Method);
         interfaces : (list Interface)
         }.
```

The Class and Interface types are records that contain several fields not described in this abstract. The bytecode instructions to be executed by JCVM are encapsulated in the Method type:

```
Record Method : Set := {
       nargs      : nat;                (* Number of arguments of the method. *)
       local      : nat;                (* Number of local variables. *)
       owner      : nat;                (* Class index. *)
       bytecode   : (list Instruction)  (* All the instructions to be executed. *)
       }.
```

Here Instruction is an inductive type specifying the mnemonic and the format of all the instructions as sketched below:

```
Inductive Instruction : Set :=
          ... |
          invokevirtual   : nat -> nat -> Instruction |
          ...
```

The invokevirtual instruction is formalized as a constructor taking as arguments two naturals that respectively represent the number of arguments and the position of the invoked virtual function in the method list of the class of the object under consideration.

2.2 Memory model

The JCVM memory consists of two main elements : the heap and the stack. The heap involves all the objects that an application may use. It is formalized in Coq using a list of objects. The latter are either class instances, described using a class reference and a data area (a list of values), or arrays, constructed from the type of their elements and another data area. The type of values is the cartesian product type*Z where type represents the data types supported in Java Card (Void, Boolean, Byte, Short, Int, Ref) and Z is the type of integers.

The stack contains all the running methods and it is represented as a list of frames. Frames are described in Coq as a record type featuring an operand stack, a list of local variables, a reference to the current class, the method location, the current context (the type Package contains information that allows to identify a context) and a program counter that points to the next instruction to be executed:

```
  Record frame : Set := {
         opstack       : (list valu);
         locvars       : (list valu);
         current_class : nat;
         method_loc    : nat;
         context_ref   : Package;
         p_count       : nat
         }.
```

2.3 One-step execution

We have formalized in Coq one-step execution function of the JCVM as a term exec_jcvm of type

```
jcvm_program -> jcvm_state -> (Exc jcvm_state)
```

where jcvm_state is the cartesian product of the heap, the stack and the exception that may be returned. The term exec_jcvm is built from auxiliary functions that describe the behavior of each instruction. Our current formalization treats most instructions of the JCVM, including checkcast, athrow, instanceof, invokeiterface, aastore, new, getfield. In this abstract, we focus on the formalization of the invoke-virtual instruction, which performs the invocation of an instance method.

The invokevirtual instruction performs the invocation of an instance method. Its semantics is captured by the Coq function INVOKEVIRTUAL. The function returns an element of type (Exc jcvm_state) and is described below.

```
Definition INVOKEVIRTUAL := [nargs,idx:nat][h:frame][hp:heap][s:stack][cap:jcvm_program]

(* The number of arguments must be greater than zero *)
 Cases nargs of
  O     => Error | (S _) =>

    (* The object reference which is of type valu is the nargth element of the operand stack.
       An element of valu is a pair (t,v) where t is the type of the value v.  *)
    Cases (Nth_func (opstack h) nargs) of (value x) =>
      Cases x of (tx,vx) =>

         (* The nargth element must be a reference. *)
         Case tx of Ref =>

           (* This function raises a Null Pointer exception (Value NullPointer)
              if the value vx is null, otherwise returns error. *)
           Cases (test_nullpointer vx) of error=>

               (* The referenced object is extracted from the heap.
                  An object in the heap is either an instance or an array.  *)
                          Cases (Nth_func hp (absolu vx)) of
                             (value nhp) =>  Cases nhp of

                                            (* Tests if this object is an instance. *)
                                            (Instance ti) =>
                                              Cases ti of (Build_type_instance c _ _ _ _ _ _) =>

                                                (* The current frame is updated and the new
                                                frame is set up and pushed into the stack. *)
                                                (res_invoke c nargs idx h hp s cap nhp)
                                              end |
                                           (Array ta)   => ...
```

The new_frame_invokevirtual function, called by res_invoke, returns an element of type jcvm_state and, if needed, updates the stack frame.

```
Definition new_frame_invokevirtual:=
[cref:nat][idx:nat][l,l':(list valu)][h:frame][hp:heap][s:stack][nhp:obj]
  Cases (jcre_invoke h nhp) of

     (* If an exception is raised then an error state is returned
        in which u is the thrown exception. *)
     (value u) =>((Value u),(hp,s))

     (* Else the current frame is updated, a new frame is set up,
        pushed into the stack and becomes the new current frame.
        No exception is thrown. *)
      error    =>((Error), (hp,
                           (cons (Build_frame (nil valu)  (rev l) cref idx
                                           (get_owner_context nhp) (* The currently active context
                                                                     is the object owner's context. *)
                                           (1))                    (* The program counter points
                                                                      to the first instruction. *)
                                      (cons (update_current_frame l' h)
                                           (tail s)))))
  end.
```

The JCVM may throw an exception under certain conditions. For example:

- a `Nullpointer exception` is thrown if the object reference extracted from the operand stack is null;
- a `Security exception` is thrown if the Java Card security policy is violated.

The following example is part of the `jcre_invoke` Coq function which performs security checks upon instances (recall that objects are either instances or arrays).

```
Definition jcre_invoke := [h:frame][the_object:obj]

...
(* Tests if the referenced object is an instance. *)
Cases the_object of (Instance ti) =>

   (* Tests if the currently active context is the object owner context. *)
   Cases (eqb_AID (AID_pi (context_ref h)) (AID_pi (owner_i ti))) of
     true  => Error |
     false =>

        (* Tests if this object is an Entry Point. *)
        Cases (eqb (ptE ti) is_ptE) of true  => Error |
           false =>

              (* Tests if the currently active context is the JCRE context. *)
               Cases (eqb_AID (AID_pi  (context_ref h)) jcre_AID) of true  => Error |

                 (* A Security Exception is thrown. *)
                 false => (Value Security)
               end ...
```

2.4 Discussion

Our formalization uses a "functional style" of specification. In our opinion, such a style offers several advantages over the usual "logic programming" style of formalization using inductive relations. In particular, a functional style specification

1. is close to an implementation and may be tested for increasing confidence in the formalization;
2. is well suited to verify program properties that require an exhaustive check of execution traces, including security properties like information flow. In particular, functional style specifications are easily amenable to model-checking;
3. is re-usable in other proof-assistants, as it mostly relies on standard typing constructs such as first-order inductive types and record types, and do not make use of dependent types—in fact, we only make use of a non-standard feature of Coq, namely implicit coercions, to simulate subtyping on inductive types in the representation of Java Card programs.

3 The CAP tool

In order to test our formalizations, we have developed a CAP tool that transforms CAP files into their Coq representations. To this end, the CAP tool performs the following operations:

1. *Linking.* During this stage, the constant pools (reference tables) of different packages are fully resolved (for external references) and thus can finally be removed. This stage also involves transforming the flat format of the CAP files into a more realistic tree structure. For instance, bytes indexes in the `Method component` are translated to pointers to `method` objects;
2. *Indexes transformation.* The JCVM specification and the CAP file format are word oriented: for example the stack is described as a short vector and thus `int` local variables must take two consecutive indexes. In contrast, our Coq specification is value oriented: the stack is a list of `valu` and a method contains a list of `Instruction`. Therefore, the CAP tool must also make a deep transformation of the byte code:
   – indexes in the byte code (for example for the `goto` instruction) are resolved as `Instruction` indexes;
   – indexes in the stack (for example for the `aload` instruction) are resolved as `valu` indexes.
   Such a transformation can be viewed as a fair part of a byte code verifier.

4 Conclusion

We have reported about a precise formalization of a substantial part of the JCVM in Coq. This work is part of a larger effort to build up a toolbox for proving security properties of Java Card programs and many facets of the project still remain to be addressed.

4.1 Future work

Our most immediate objective is to complete the formalization of the JCVM and JCRE. In parallel, we intend to develop tools for verifying security properties that are not enforced by the JCVM security mechanisms. For example, the Java Card language does not make any provision to prevent information leakage through transitive information flow; in order to verify the absence of such information leakage, we plan to use [23] to set up a connection between our formalization of the JCVM and model-checkers.

In the longer term, we are also interested in integrating abstract interpretations to our environment. Ideally, these abstract interpretations should be certified within Coq itself—preliminary work in this direction may already be found in [16].

4.2 Related work

Due to space constraints, we mostly focus on projects that make use of specification and/or verification tools for Java or Java Card. For the sake of clarity, we distinguish between platform-oriented projects and application-oriented projects.

Platform-oriented projects The proof assistant Coq has been used in a number of case studies on the JVM. For example, Bertot [2] has formalized the type system of Freund and Mitchell [9], and Segouat [21] has formalized the correctness of the converter from bytecode to CAP files following Denney and Jensen [6].

Yet the most impressive work to date is that of the Bali team at Munich, see e.g. [18], who formalized in Isabelle/HOL a large body of the Java platform, including:

1. the type system and the operational semantics of both the source language, with a proof of type-safety at both levels;
2. the compiler and an abstract bytecode verifier, with their proof of correctness.

In addition, there have been a number of similar efforts by Syme [22], who formalized the operational semantics of Java and machine-checked the proof of type soundness in DECLARE, by Lanet and Requet [15] who formalized most of the JCVM in B, and by Cohen [4] who formalized the so-called Defensive JVM in ACL2.

Application-oriented projects The LOOP project see e.g. [13] has developed a tool to compile Java classes into PVS or Isabelle theories that form the basis for actual verifications. Other projects, such as the Java Path Finder [12] and Bandera [5], emphasize the use of algorithmic techniques for verifying properties of Java programs. For example, the Bandera toolset combines several program analyses/program transformation techniques, including slicing and partial evaluation, to extract from Java source code compact finite-state models to be submitted to model-checkers.

Specification projects A number of teams have developed formal specifications of Java and its variants: these include executable specifications as in [11] or precise pen-and-pencil specifications as in [3, 1, 8]. There is also a variety of works that focus on specific aspects of Java, see e.g. [7, 14, 20], and Java Card, see e.g. [17, 19].

Acknowledgements Thanks to Henrik Nilsson for numerous discussions on the JCVM.

References

1. P. Bertelsen. Semantics of Java Byte Code. Master's thesis, Department of Computer Science, Royal Veterinary and Agricultural University of Copenhagen, 1997.
2. Y. Bertot. Formalizing in Coq a type system for object initialization in the Java bytecode language. Talk at the SJava meeting, Sophia-Antipolis, March 2000.
3. E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 353–404. Springer-Verlag, 1999.
4. R. M. Cohen. Defensive Java Virtual Machine Specification Version 0.5. Manuscript, 1997.
5. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of ICSE'2000*, 2000.
6. E. Denney and T. Jensen. Correctness of Java Card method lookup via logical relations. In G. Smolka, editor, *Proceedings of ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 104–118. Springer-Verlag, 2000.
7. S. Drossopoulou. Towards an abstract model of Java dynamic linking and verification. Manuscript, 2000.
8. S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and Proving Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 41–82. Springer-Verlag, 1999.
9. S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. In *Proceedings of OOPSLA '98*, volume 33(10) of *ACM SIGPLAN Notices*, pages 310–328. ACM Press, October 1998.
10. P. Girard. Which security policy for multiapplication smart cards? In *Proceedings of Usenix workshop on Smart Card Technology (Smartcard'99)*, 1999.
11. P. H. Hartel, M. J. Butler, and M. Levy. The Operational Semantics of a Java Secure Processor. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 313–352. Springer-Verlag, 1999.
12. K. Havelund. Java PathFinder—A Translator from Java to Promela. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, page 152. Springer-Verlag, 1999.
13. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes. *ACM SIGPLAN Notices*, 33(10):329–340, October 1998.
14. T. Jensen, D. Le Métayer, and T. Thorn. Security and Dynamic Class Loading in Java: A Formalisation. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 4–15, 1998.
15. J.-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In *Proceedings of CARDIS'98*, 1998. To appear.
16. D. Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998.
17. M. Montgomery and K. Krishna. Secure Object Sharing in Java Card. In *Proceedings of Usenix workshop on Smart Card Technology, (Smartcard'99)*, 1999.
18. T. Nipkow, D. von Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proceedings of the International Summer School Marktoberdorf 1999*, pages xxx–xxx. IOS Press, 2000. To appear.
19. M. Oestreicher and K. Krishna. Object Lifetimes in Java Card. In *Proceedings of Usenix workshop on Smart Card Technology, (Smartcard'99)*, 1999.
20. Z. Qian. A formal specification of Java[TM] virtual machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 271–312. Springer-Verlag, 1999.
21. G. Ségouat. Preuve en Coq d'une mise en oeuvre de Java Card. Master's thesis, University of Rennes 1, 1999.
22. D. Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 83–118. Springer-Verlag, 1999.
23. K. N. Verma and J. Goubault-Larrecq. Reflecting BDDs in Coq. Technical Report RR3859, INRIA projet Coq, January 2000.

Formal Models of Linking for Binary Compatibility

D. Wragg

Formal Models of Linking for Binary Compatibility

David Wragg

Department of Computing, Imperial College, London

Abstract. The Java Language Specification devotes a chapter to the issue of binary compatibility, but at the time it was written there was no formal work on binary compatibility to serve as a rigorous foundation for understanding of the issues involved. This is reflected in the flaws in the specification.

This paper introduces two formal models of linking suitable for reasoning about binary compatibility in Java. The first model gives an abstract view of linking. The second model represents the dependencies between fragments (replaceable sections of programs). The need for these two models and the difference between their purposes are discussed.

1 Introduction

With separate compilation, as in Java[6], a compiler cannot atomically type check the whole program, since it does not necessarily receive the whole program. When a compiler type checks an individual source file, it may require type information corresponding to other source files, but that type information can be superseded if those source files are later modified and recompiled. So in general, type checking of individual source files is not sufficient to guarantee *type safety*. That is, values in the executing program are not guaranteed to conform to their types given in the source files, leading to the risk that execution will not be *well behaved*[2].

If a program is written by a single developer, or a group of developers working closely together, it is relatively easy for them to ensure that source files are recompiled as necessary to ensure type safety. One way to do this is to recompile every source file after any source file is modified, but there are tools which can achieve the same effect with fewer recompilations[4].

However, much software is now delivered and installed as separate *object files* (the files containing compiled code, sometimes known as *binaries*) which can be independently upgraded to later versions. For instance, a Java application consists of the code written by the application developers, but also of the sections of the Java platform used by the application. In this context, no recompilations occur after an object file is upgraded, since the software will usually be installed on a machine without access to the source files or the development tools needed to recompile them, and type safety may be lost.

The process of combining object files together and resolving references between them to form a complete program is known as *linking*, and when this process coincides with the execution of the program it is known as *dynamic linking*. Dynamic linking has become commonplace, and is a feature of the Java Virtual Machine[7]. Since it allows object files to be independently upgraded up until (and possibly concurrently with) execution of the program, dynamic linking threatens type safety.

This hazard is the motivation for the notion of binary compatibility. While arbitrary changes to object files cause type safety to be lost, there are certain changes that preserve type safety: If a *binary compatible change* is made to a source file, which is compiled to give a new version of the corresponding object file, then substitution of the new version of the object file for the old version in any well behaved program yields another well behaved program[5]. An example of a binary compatible change in Java is the addition of a method to a class. By evolving a source file (and hence the corresponding object file) using only binary compatible changes, so that later versions are *binary compatible* with earlier versions, developers can ensure that well behaved execution is preserved.

In order to maintain the integrity of its security mechanisms, the Java virtual machine does type checking of object files when they are loaded and dynamically linked. This type checking traps ill behaviour before it can arise during execution; instead the virtual machine throws a linking error exception, effectively aborting the program. But from the perspective of a developer, while an aborted program may be preferable to an incorrectly functioning program, these linking errors are still undesirable. So for the purpose Java binary compatibility, we consider well behaved programs to be those for which linking does not fail.

In order to put the idea of binary compatibility into practice, developers need a set of guidelines laying out the binary compatible changes they can make. The Java Language Specification contains such guidelines. Unfortunately, they are slightly flawed[3]. By studying binary compatibility formally, we hope to show that the corrected guidelines are indeed sound. A formal basis for understanding binary compatibility may also help in the development of binary compatibility guidelines for other languages, and perhaps help the designers of future languages to incorporate good support for binary compatibility.

2 Fragment Systems

Since Java binary compatibility is defined in terms of the success or failure of linking, in order to express it formally we first need an appropriate model of linking. This model will be introduced in terms of *static linking* — linking where the set of object files that comprise a program is known without executing (as opposed to dynamic linking, where this might not be the case). Later on, we will discuss how to apply the model more generally.

Let B be the set of all possible object files. We will call sets of object files *fragments*, since in general they are fragments of full programs[1]. Let F be the set of fragments: F =3D $\mathscr{P}(B)$.

A *complete fragment* is one containing object files that successfully link together to yield a whole program. Let C_F be the set of complete fragments from

F. Testing whether a fragment is in C_F corresponds to testing whether the object files in the fragment link together successfully.

According to the description of binary compatibility in the last section, an object file $a' \in B$ is binary compatible with another object file $a \in B$ iff for all $b \in F$

$$\{a\} \cup b \in C_F \Rightarrow \{a'\} \cup b \in C_F$$

where the term $\{a\} \cup b$ represents a program containing the old version of the object file, and $\{a'\} \cup b$ is that program with the old version replaced by the new version.

Extending that formulation to express binary compatibility between sets of object files, *i.e.* fragments, gives: a fragment $a' \in F$ is binary compatible with $a \in F$ iff for all $b \in F$

$$a \cup b \in C_F \Rightarrow a' \cup b \in C_F$$

for $a, a' \in F$. This also allows changes that involve the addition or removal of object files to be considered.

This example of the static linking of object files is an instance of a *fragment system*:

Definition 1 *A fragment system is a set of fragments F equipped with*

– *An associative, commutative monoid $(F, +, 0_F)$. That is, given $a, b, c \in F$*

$$\begin{aligned} a + (b + c) &=3D\ (a + b) + c \\ a + b &=3D\ b + a \\ a + 0_F &=3D\ a \end{aligned}$$

– *The set $C_F \subseteq F$ of complete fragments.*

In the example

$$\begin{aligned} F &=3D\ \mathscr{P}(B) \\ C_F &=3D\ \{\, a \in F \mid \mathit{links}(a) \,\} \\ a + b &=3D\ a \cup b \\ 0_F &=3D\ \emptyset \end{aligned}$$

where *links* is a predicate that tests whether a set of object files will link successfully.

The notion of *completions* is useful for expressing several other concepts within a fragment system:

Definition 2 *For a fragment a of a fragment system F, $[\![a]\!]_F$ denotes the set of* completions *of a in F, defined by*

$$[\![a]\!]_F =3D\ \{\, b \in F \mid a + b \in C_F \}$$

The set of completions of a fragment a gives the full behaviour of a within the fragment system, since for any fragment $c =$3D $a + b$ (*i.e.* c contains a), c is complete if and only if $b \in [\![a]\!]_F$.

The formulation for binary compatibility given above can be rewritten in terms of completions: $a' \in F$ is binary compatible with $a \in F$ iff

$$[\![a]\!] \subseteq [\![a']\!] \ .$$

So binary compatible changes are exactly those that preserve completions.

Another use for completions is to express the concept of an *ill-formed* fragment: one that causes linking to fail when combined with any other fragment. Examples of ill-formed fragments are those containing corrupt object files, or those containing object files with conflicting definitions for a class. For an ill-formed fragment a,

$$a + b \ /\!\in \mathcal{C}_F$$

for all $b \in F$. So

$$[\![a]\!]_F = \text{3D} \ \emptyset \ .$$

With dynamic linking, the set of object files that comprise a program may depend upon the run-time behaviour of that program; the sequence in which those object files are linked in to the image of the executing program may also vary. So it is possible that a program will sometimes abort with a linking error, and sometimes execute normally to termination. However, the issue that is of most practical interest is whether there is any execution of the program that will abort with a linking error, since that possibility will usually indicate a defect in the program.

Although with dynamic linking the object files that make up a program are not fixed, the object files must come from somewhere. For Java, they typically reside on local storage or are retrieved from the World Wide Web. So while there is no fixed set of object files that make up a program, there is a fixed pool of object files from which an executing program can draw.

In practice, there must also be some means to obtain the initial program. For example, Java applications and Java applets begin execution with the name of a class which indicates an initial object file to be loaded and an initial method call to perform. Such mechanisms will not be made explicit in the model.

So, a fragment system representing dynamic linking can again use sets of object files as fragments, but instead of representing sets that may comprise programs, they now represent the pools of object files which executing programs load from. A complete fragment is one which permits no executions that result in linking errors.

3 Requirements and Provisions

Fragments systems offer an elegant way to express binary compatibility through the preservation of completions, but in order to reason about binary compatibility as it applies to a particular programming language we have to specify the internal structure of the fragments. We could develop language specific models, where the internal structure of the fragments is directly derived from the language under consideration. However, there are some similarities in how binary compatibility manifests itself in many different languages. For example, in Java adding a method to a class is a binary compatible change, and in C++ adding a member function to a class can be a binary compatible change (although most C++ implementation impose restrictions that Java avoids[5]). Furthermore, within a single language, there can be similarities in the way binary compatibility applies to different language features. For example, in Java adding a method to a class is a binary compatible change, and so is adding a field to a class.

Fragment systems are too abstract to account for these similarities, so we need another model which is closer to the realm of concrete programming languages, but still general enough to applied to many languages. This section introduces such a model, though the full details of the formalization are too lengthy to be presented here.

Programming languages typically allow their modules to define entities (variables, procedures, types, classes, and so on), and to refer to entities defined in other modules. We will use the term *requirement* for a dependency of a fragment (corresponding to a set of zero or more modules) upon some entity, and *provision* for a feature of a fragment that can resolve such dependencies.

For example, consider the following two fragments for Java: a consists of the class

```
class A {
        String m() { return "I am an instance of A"; }
}
```

and b consists of the class

```
class B {
        String n(A x) { return "x says: " + x.m(); }
}
```

Here the provisions of a are the existence of a class called `A` and its method `A.m` taking no arguments and returning a string. The provisions of b are the existence of a class called `B` and its method `B.n`, and b requires a class called `A` having a method `A.m` taking no arguments and returning a string.

The requirements of fragments are represented formally by a requirements structure.

Definition 3 *A* requirements structure $(R, \sqsubseteq)$ *is a bounded lattice. Let* $\sqcup$, $\sqcap$ *denote the join and meet operators, and* $\top_R$, $\bot_R$ *denote the top and bottom elements.*

$\bot_R$ represents the least requirements in the model; that is, the requirements of the empty fragment 0_F. $\top_R$ represents requirements that can never be satisfied. For instance, by interleaving recompilations with modifications to source files, it is possible to create a pair of Java object files that require an inheritance circularity, *e.g.* with one requiring that a class `C` is a subclass of `D` and the other requiring that `D` is a subclass of `C`. The fragment consisting of these two object files can never have its requirements satisfied.

Fragment provisions are represented by a provisions structure:

Definition 4 *For a* provisions structure $(P, P_v, +_P, 0_P)$,

- P *is the set of provisions.*
- P_v *is the set of* valid provisions*;* $P_v \subseteq P$*.* Invalid provisions *are those in* P *but not in* P_v*.*
- $+_P$ *is a commutative associative binary operator on* P *that combines provisions:*

$$+_P \;:\; P \times P \rightarrow P$$

  Invalid provisions always combine to give invalid provisions. Conversely, for $p, q \in P$

$$p +_P q \in P_v \Rightarrow [p \in P_v \textit{ and } q \in P_v]$$

- $0_P \in P_v$ *is the unit of* $+_P$*. For* $p \in P$

$$p +_P 0_P =3\mathrm{D}\; p$$

A provisions structure is more sophisticated than a requirements structure because we wish to to make certain combinations of provisions invalid. For example, a fragment may contain a set of classes with an inheritance circularity. A Java implementation will abort with a `ClassCircularityError` in this case, and the model should reflect this.

Requirements and provisions are combined through a function *Sat* which determines which provisions satisfy which requirements.

Definition 5 *A* requirements/provisions system $(P, P_v, +_P, 0_P, R, \sqsubseteq, Sat)$ *is a provisions structure* $(P, P_v, +_P, 0_P)$*, a requirements structure* $(R, \sqsubseteq)$*, and a function*

$$Sat \;:\; P_v \rightarrow R$$

A requirements/provisions system must obey the following:

1. *For* $p, q \in P_v$ *such that* $p + q \in P_v$*,* $Sat(p) \sqcup Sat(q) =3\mathrm{D}\; Sat(p + q)$*.*
2. *For* $p, q, q' \in P_v$ *such that* $p + q \in P_v$ *and* $Sat(q') \sqsubseteq Sat(q)$*,* $p + q' \in P_v$*.*

The *satisfies* relation is defined in terms of *Sat*: For $p \in P_v$ and $r \in R$,

$$p \vdash r \Leftrightarrow r \sqsubseteq Sat(p)$$

Note that the domain of *Sat* is P_v, not P; this is the sense in which the provisions in $P - P_v$ are not valid.

The corresponding fragment system is constructed using pairs of provisions and requirements:

$$\begin{aligned}
F &=3\text{D } P \times R \\
+ &=3\text{D } (+_P) \times (\sqcup) \\
0_F &=3\text{D} (0_P, \bot_R) \\
C_F &=3\text{D } \{\, (p, r) \mid r \sqsubseteq \mathit{Sat}(p) \,\}
\end{aligned}$$

We can derive properties of the completions in this fragment system in terms of requirements and provisions. For $p, q \in P_v$ and $r, s \in R$,

$$(q, s) \in [\![(p, r)]\!]_F$$

iff

$$\begin{aligned}
s &\sqsubseteq \mathit{Sat}(p) \sqcup \mathit{Sat}(q)\,, \text{ and} \\
r &\sqsubseteq \mathit{Sat}(p) \sqcup \mathit{Sat}(q)\,.
\end{aligned}$$

So the completions of a fragment (p, r) grow as the requirements r are reduced, and as $\mathit{Sat}(p)$, the ability of the provisions p to satisfy requirements, grows. Since the binary compatible changes are those that preserve completions, binary compatible changes can be stated in terms of the way they affect the requirements and provisions of fragments written in the programming language under consideration. These statements are not very different to the informal descriptions of the binary compatible changes given in the Java Language Specification.

4 Conclusions

This paper has introduced two models of linking for investigating binary compatibility. These models have different purposes. The model of fragment systems is more abstract, and so enables the concise expression of high-level concepts. The model of requirements and provisions is closer to concrete programming languages, and so allows us to reason about how binary compatibility manifests itself, but still without being restricted to a particular language.

5 About the Author

David Wragg is studying formalizations of binary compatibility for Java towards a PhD in computing at Imperial College, London, with funding from the EPSRC. He is a member of the SLURP group
`<URL:http://www-dse.doc.ic.ac.uk/projects/slurp/>`.

References

1. L. Cardelli. Program Fragments, Linking, and Modularization. In *POPL'97 Proceedings*, January 1997.
2. Luca Cardelli. Type Systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997. Available at `<URL:http://www.luca.demon.co.uk/>`.
3. Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java Binary Compatibility? In *OOPSLA'98 Proceedings*, 1998.
4. S. I. Feldman. Make — A Program for Maintaining Computer Programs. *Software–Practice & Experience*, 9(3):255–265, March 1979.
5. Ira Forman, Michael Conner, Scott Danforth, and Larry Raper. Release-to-Release Binary Compatibility in SOM. In *OOPSLA'95 Proceedings*, 1995.
6. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
7. Tim Lindholm and Frank Yellin. *The Java Virtual Machine*. Addison-Wesley, 1997.

A Case Study in Java Software Verification: SecurityManager.checkConnect()

P. Brisset

A Case Study in Java Software Verification: SecurityManager.checkConnect()

Pascal Brisset*

France Telecom R&D, DTL/MSV
Technopole Anticipa, 2 avenue Pierre Marzin
22307 Lannion, France
pascal.brisset@rd.francetelecom.fr

Abstract. Run-time authorization of network communications between untrusted applications and remote hosts is one of many security features in the Java platform. We describe the corresponding security mechanism, which is part of the runtime library, and observe that establishing its correctness requires a surprisingly complex line of argument involving a number of salient features of the Java language. This makes it a challenging case study in formal verification of object-oriented software. Finally, we give an overview of our current work which is based on refutation in temporal logic, Floyd/Hoare style semantics and on-the-fly generation of program specifications.

1 Introduction

Java security is commonly associated with advanced techniques such as byte-code verification and dynamic class loading [1]. However, several critical security mechanisms consist of simple run-time checks performed by the Java runtime library. There is strong incentive to formally verify that these mechanisms cannot be bypassed by hostile applications.

Principals who execute untrusted Java code expect the runtime library to guarantee that dangerous operations are always authorized by the so-called SecurityManager, a distinguished object which embodies the security policy of the virtual machine.[1] Consider the following hypothetical code for deleting files:

```
public class File {
  static public void delete(String path) {
    SecurityManager security = System.getSecurityManager();
    if ( security != null ) security.checkDelete(path);
    really_delete(path);
  }
  static native private void really_delete(String path);
}
```

* This work was partially supported by Eurescom project P1005.

[1] How the decision is taken (e.g. by matching cryptographic credentials with access control lists, as in JDK1.2) is beyond the scope of this paper.

Native method really_delete performs the actual system calls. Applications can only reach it through method delete, which first calls method checkDelete of the SecurityManager (if one has been installed). checkDelete(path) is expected to throw a SecurityException if the security policy forbids deletion of the file specified by its argument.

In the next section we will see that establishing the effectiveness of other basic run-time security checks can be much more complex.

2 Network access control in the Java runtime library

2.1 Description

Class Socket allows Java programs to open TCP connections. The security model specifies that all connections will be authorized by method checkConnect of the SecurityManager. This could have been implemented like the file access control in section 1; unfortunately, the actual design is more complex:

- A Socket object is not associated with a system socket. Instead, it contains a pointer to an instance of SocketImpl.
- For the sake of flexibility, class Socket can be instructed to obtain SocketImpl objects either by instanciating the default class PlainSocketImpl, or by invoking method createSocketImpl of a user-supplied SocketImplFactory.
- The native methods whose access must be controlled are members of class PlainSocketImpl, but the security checks are performed in class Socket.

For clarity, we provide in figure 1 a simple, stand-alone Java package which essentially reflects publicly available APIs and documentation [2]. Readers are encouraged to examine Sun's reference implementation, which contains many additional features, but is not significantly harder to analyze.

2.2 Informal verification

We assume that compiled, dynamically-loaded classes obey language-level constraints such as access modifiers and structured control flow. This is essentially what bytecode verification, secure class loading and many other design decisions aim to guarantee. Correctness of the checkConnect() mechanism can then be informally specified as the negation of:

Definition 1. *(Attack) There exists an application A, two pointers* sm *and* addr*, two states S and S', an execution path P from S to S' for A, and a thread T such that:*

- *In state S,* System.security=sm.
- sm≠null.
- *In state S, the machine is executing untrusted application code.*
- *P does not contain a successful invocation of* sm.checkConnect(addr) *by T.*
- *In state S', T is about to invoke* PlainSocketImpl.connect(addr).

File runtime/Socket.java

```
package runtime;            // ...reflects java.lang and java.net
public class Socket {
    SocketImpl impl;        // Recall that the default visibility is "package"
    public Socket(Address a) {
        impl = new PlainSocketImpl();
        SecurityManager sm = System.getSecurity();
        if ( sm != null ) sm.checkConnect(a);
        impl.connect(a);
    }
}
```

File runtime/SocketImpl.java

```
package runtime;
public abstract class SocketImpl {
    protected abstract void connect(Address a);
}
```

File runtime/PlainSocketImpl.java

```
package runtime;
class PlainSocketImpl extends SocketImpl {
    protected native void connect(Address a);
}
```

File runtime/System.java

```
package runtime;
public final class System {
    private static SecurityManager security = null;
    public static synchronized void setSecurity(SecurityManager sm) {
        if ( security == null ) security = sm;
    }
    public static SecurityManager getSecurity() { return security; }
}
```

File runtime/SecurityManager.java

```
package runtime;
public class SecurityManager {
    public void checkConnect(Address a) { }
}
```

File runtime/Address.java

```
package runtime;
public class Address { }
```

Fig. 1. A stand-alone case study. The only external reference is class java.lang.Object.

Our impression is that the design in figure 1 actually satisfies this specification. Due to lack of space, details of the informal proof are left as an exercise for the reader. We suggest refuting attacks by tracing execution paths P backwards and obtaining contradictions before reaching untrusted application code. This approach requires the following auxiliary results:

– **Field System.security can be assigned only once in the lifetime of the virtual machine**. Note that attribute synchronized on method setSecurity prevents interleaved executions by multiple threads.
– **Pointers to instances of PlainSocketImpl cannot be leaked to untrusted code**. This requires a non-trivial data flow analysis.

Note also that several arguments about visibility rules can only be carried out by exhaustively searching whole classes or even packages for specific instructions such as assignments and instanciations of particular fields and classes.

3 Formal verification

From the perspective of formal verification, the checkConnect() mechanism has several interesting characteristics:

– **Assurance requirements**: The implementation of this mechanism is security-sensitive code installed in millions of computers. It is neither trivial nor excessively hard to verify, and makes brilliant use of Java's language-level security features: most attempts to change access modifiers would either introduce vulnerabilities or decrease functionality.
– **Hybrid techniques required**: It is doubtful that a single verification technique could handle all kinds of arguments mentioned in section 2.
– **Scalability**: Although the problem boils down to 30 lines of Java source code, verification cannot be carried out without examining large volumes of surrounding code.
– **Object-oriented features involved**: Difficulties stem from subtle interactions between control flow, inheritance, exceptions and Java's sophisticated visibility rules.

We are currently designing verification techniques to address these issues. Our approach is characterized by:

– **Verification on bytecode rather that source code**: For verification purposes, bytecode is generally a convenient representation of Java programs (with the exception of subroutines).
– **Semi-automated refutation in temporal logic**: We extend CTL with JVM-specific atomic formulae, embed it in a proof assistant, and use a library of theorems to mimic the backward-branching reasoning style of section 2. For example, $[[\ (\text{EX}\ P) \Rightarrow Q\ ]] \rightarrow [[\ (\text{EF}\ P) \Rightarrow P \vee (\text{EF}\ Q)\ ]]$ is a basic theorem for reasoning about purely sequential programs. We hope to obtain human-readable proofs appropriate for use in government-sponsored security evaluation frameworks such as ITSEC and Common Criteria.

- **Floyd/Hoare-like semantics**: We do not explicitly describe the semantics of Java bytecode; instead, logic formulae are transformed by pre- and post-condition generators. This is inspired from the verification condition generator in [5], with extensions for threads and temporal logic.
- **On-the-fly generation of program specifications**: Thanks to Java's structured control flow, bytecode can be translated into pre- and post-condition generators for CTL formulae one method at a time, as required by the current goal. This is essential for scalability.

4 Related work and perspectives

Because of its industrial relevance, reasonable design and advanced security features, the Java platform is a target of choice for formal methods. Significant work has already been invested in the formalization of both the language and the virtual machine. Surprisingly enough, verification techniques for Java programs have received less consideration.

Both ESC [7] and LOOP [3] use a translation into an intermediate language followed by theorem proving in order to verify generic safety properties, possibly involving complex invariants. Our work is more focused on security properties and temporal reasoning. It was strongly motivated by [4], which also uses a temporal logic to automatically verify programs with more complex security properties, but does not address data-dependent behaviour.

One of our major concerns, also found in [6], is easy application to existing code: although we discussed a simplified case study in this paper, our goal is to provide programmers with tools that can process programs and libraries with minimal adaptation. Significant work will be required to handle peculiar features of Java such as static class initialization, finalizers, object locks, reflection and serialization.

References

1. Drew Dean, Edward W. Felten, and Dan S. Wallach, *Java Security : From HotJava to Netscape and Beyond*, IEEE Symposium on Security and Privacy, 1996.
2. Sun Microsystems Inc, *Extending Sockets in JDK 1.1*, JDK documentation, http://java.sun.com/products/jdk/1.1/docs/guide/net/extendingSocks.html.
3. B. Jacobs, M. Huisman, M. van Berkum, U. Hensel, and H. Tews, *Reasoning about java classes (preliminary report)*, Object-Oriented Programming Systems, Languages and Applications, ACM Press, 1998, pp. 329–340.
4. T. Jensen, D. Le Métayer, and T. Thorn, *Verification of control flow based security policies*, Tech. Report 1210, IRISA, 1998.
5. George C. Necula and Peter Lee, *Proof-Carrying Code*, Tech. Report CMU-CS-96-165, CMU, September 1996.
6. Robert O'Callahan, *The Design of Program Analysis Services*, Tech. Report CMU-CS-99-135, Carnegie Mellon University, 1999.
7. K. Rustan, M. Leino, James B. Saxe, and Raymie Stata, *Checking Java programs via guarded commands*, Tech. Report 1999-002, SRC Technical Note, 1999.

Simulation and Class Refinement for Java

A. Cavalcanti, D. Naumann

Simulation and class refinement for Java

Ana Cavalcanti[1] and David A. Naumann[2]

[1] Centro de Informática
Universidade Federal de Pernambuco (UFPE), Box 7851 50740-50 Recife PE Brazil
Phone: +55 81 271 8430 Fax: +55 81 271 8438
`alcc@di.ufpe.br` `www.di.ufpe.br/~alcc`
[2] Department of Computer Science
Stevens Institute of Technology, Hoboken NJ 07030 USA
`naumann@cs.stevens-tech.edu` `www.cs.stevens-tech.edu/~naumann`

Keywords: behavioral subclassing, modular specification and verification, inheritance and dynamic binding, refinement calculi, semantics

1 Introduction

This document is an extended abstract of ongoing work on reasoning about correctness and behavioral subclassing for programs using some of the challenging features of Java. We include dynamic binding and inheritance, visibility control, and mutually recursive classes and recursive methods (but not pointers or concurrency). In [CN99a,CN99b,CN00] we give a predicate-transformer semantics for a language with these features as well as specification constructs from refinement calculi [Mor94,BvW98]. Here we discuss intrinsic notions of behavioral subclassing and refinement, and we discuss soundness of proof by forward simulation.

This section describes the context and objectives for our project. Section 2 presents the language and semantics; Section 3 discusses refinement; Section 4 sketches the background of the authors and our expectations for the workshop.

Our work is part of an ongoing collaboration involving others at UFPE (P. Borba and A. Sampaio) and Birmingham (U. Reddy) and our research assistants. A long-term goal is to use refinement laws as the basis both for a development method and for tools supporting program analysis, verification, and compilation. In our current project the focus is on semantics, behavioral subclassing and refinement, and laws of refinement that are sufficient to reduce programs to a normal form. Laws, however, are beyond the scope of this abstract.

The specification statement $x : [pre, post]$ is treated as a command (albeit one that is not executable). The relation $\sqsubseteq$ of (algorithmic) refinement subsumes the usual notion of satisfaction of specifications; in particular, $x : [\phi, \psi] \sqsubseteq c$ holds just if c meets the specification "modifies x, requires ϕ, ensures ψ". Rules of Hoare logic are reformulated in terms of refinement laws.

Intermixing specifications with programs is very natural for use with abstract classes. For stepwise development of object-oriented programs, one needs a notion of refinement to account for correctness-preserving replacement of one definition of a class by another. We write $cds \bullet c$ for a program c using classes declared in a collection *cds* of class declarations. When a class declaration *cd* is behaviorally refined by another class declaration cd', we should have that

$$cds, cd \bullet c \sqsubseteq cds, cd' \bullet c \tag{1}$$

for suitable main programs c. Indeed, this can be taken as the intrinsic definition of class refinement.

A complementary notion of behavioral refinement is that of behavioral subclassing: here, two classes *cd* and cd' coexist, with cd' declaring a class by extension from *cd*. Syntactic subclassing allows objects from cd' to be used in place of objects from *cd*, so the behavior of cd' objects should refine the behavior of *cd* objects. If this is not the case, it can be extremely difficult to reason about program behavior, and many programs will exhibit undesirable behavior.

Adequate formalization of refinement depends on language features used in programs and specifications. Unbridled type tests and casts can thwart behavioral subclassing because their semantics is based on naming. For example, in a context where cd' declares a subclass C' of the class C declared by *cd*, and x is a variable of type C, the command

if x **instanceof** C' **then abort else skip fi**

is ill behaved even if *cd* and cd' define identical behavior.

By making a careful study of behavioral refinement in a language with many of the challenging features of Java, we believe we will contribute to the advance of formal techniques for analysis and verification of Java programs. Many of the issues apply to both notions of behavioral refinement described above. Our results shed light on issues relevant beyond the relatively narrow and long-term goals of program development based on refinement. In particular, our semantic model can illuminate issues that are obscured in detailed operational models, just as other abstractions —Hoare logic, coalgebras, and monads— are useful conceptual tools.

On the other hand, abstract semantic models must be justified in terms of a suitable connection with concrete operational semantics, and that connection should be as transparent as possible. Weakest precondition predicate transformers are attractive because, for imperative programs, the connection with operational semantics is well known (and crucial in completeness proofs connecting Hoare logics with operational semantics). To a surprising extent, we succeeded in using definitions that extend standard definitions of weakest preconditions for sequential programs [Mor94]. One of the most exciting moments in our work came when we found a semantics for dynamic dispatch that is, on the one hand, a transparent reflection of the operational mechanism, and, on the other hand, compatible with the usual definitions of weakest preconditions for ordinary commands.

2 Syntax and semantics

We present the syntax by means of a small contrived example.

```
class C    pri x : int; nxt: C
   meth Dec ≙ avar y • x : [x = y, x < y] end y
   meth Inc ≙ res r • x := x + 1; nxt := (new C; nxt : self.nxt); r := x
• c.Inc()
```

The main program works in a state space with variable c of type C, in a context where C has private attributes x and nxt. Public method Dec is specified to decrease x, leaving nxt unchanged; the specification uses "angelic varible" y as a logical constant. As in most other work, our expressions have no side effects, although they can fail, as in null dereferencing. Values are returned via result parameters like r in method Inc. Pointers are a complication we choose to set aside. In method Inc, the expression **new** C constructs a fresh object, and in the asignment to nxt there is an update expression which creates a copy of **new** C with its nxt field set to a copy of **self**.nxt.[1] Constructors and related issues are ignored throughout this abstract.

As in other current work (e.g, [PHM99]), our formalism is not fully compositional: to reason about method calls we need the complete collection of existing classes. But we do allow the main program to have variables of declared object types (not just primitive data) which allows compositional reasoning about main programs in a fixed context. Public attributes can be used to specify such programs.

Semantically, commands denote predicate transformers. We have also given a set-theoretic semantics for expressions, which we use to prove type-correctness results; eventually this semantics will be used as part of the connection between predicate transformer and state transformer semantics. In the semantics of expressions, object states are tuples, finitely nested in the case of recursive classes like C above.

To organize the semantic definitions we adopt type-theoretic techniques. A complete program $\mathit{cds} \bullet c$ is typed in the context of its global variables, as in $x : T \rhd \mathit{cds} \bullet c$. For command c that can occur in method bodies in class N, we use a typing judgement $\Gamma;\ x : T, N \rhd c$ where Γ contains information about declared classes and subclassing, attributes, and method signatures; here x is a list of attributes, parameters, and local variables with corresponding types T. A similar judgement is used for expressions. The typing rules embody Java's rules for visibility and for subsumption in assignments and parameter passing.

The semantics of a typing judgement is given by induction on the typing derivation. The semantics of a command depends on an environment that gives the semantics of methods. Because commands can have recursive calls, the environment is obtained by a fixpoint construction. Given the environment η, the semantics $[\![\Gamma;\ x : T, N \rhd c]\!]\eta$ is a function from and to formulas, with the property that if ψ is typable for $\Gamma;\ x : T, N$ then so is $[\![\Gamma;\ x : T, N \rhd c]\!]\eta\ \psi$.

[1] In conventional syntax this is $t :=$ **new** $C;\ t.\mathit{nxt} := \mathit{nxt};\ \mathit{nxt} := t$ with local t.

Parameter passing is by value and result; self is treated as an implicit value-result parameter. A call $x.m(e)$ has the following semantics, explained below.

$$[\![\Gamma, N \rhd x.m(e)]\!]\eta\ \psi = (\vee_{N' \leq_\Gamma N''} \bullet\ x\ \mathbf{isExactly}\ N' \wedge f_{N'}\psi)$$

where $f_{N'} = [\![\Gamma, N \rhd (\eta\ N'\ m)((N')x, e)]\!]\eta$ for each $N' \leq_\Gamma N''$, and N'' is the type of x. The environment provides program text $\eta\ N'\ m$ for method m in class N'; this text is obtained using fixpoint techniques from [LM98,CSW99]), and refers to attributes by way of an explicit self parameter. It is applied to argument e and also $(N')x$ which passes x explicitly as self, so that the application can be typed in the context of the call. The cast (N') is needed because the self value-result parameter in class N' has exactly type N'.

This semantics captures dynamic dispatch and the actual semantics of method bodies: an exact type test ensures that the method body applied is exactly that determined by the dynamic type of the object. To avoid unsoundness [MS98] and simplify logic [PHM99,AL97] it is natural to reason in terms of a base class specification, and some work goes so far as to require a specification which is then taken to be the semantics of a method call [Lei98]. But we want to study reasoning methods rather than presupposing them in semantic definitions.

Algorithmic refinement of commands is defined in terms of the pointwise order on predicate transformers: omitting typing context we have $c \sqsubseteq c'$ iff $[\![c]\!]\psi \Rightarrow [\![c']\!]\psi$ for all suitably typed predicate formulas ψ. Typing of predicates poses an interesting issue with respect to visibility. If we consider only predicates that respect the visibility rules, then we cannot distinguish appropriately between commands with inadequate visible state. For the example program, no postcondition for *c.Inc* could distinguish the given method body from one in which *Inc* sets *nxt* to null. As a result, our semantics would fail to be operationally sound. Thus for some semantic purposes we use an extended typing system in which visibility constraints are dropped.

3 Refinement

In this abstract we confine attention to a relation $cds \rhd cd \preccurlyeq_= cd'$ intended to capture the situation that cd and cd' declare the same class name (and superclass), with cd' a behavioral refinement of cd in the context of other classes cds with which there may be mutually recursive references. The intrinsic requirement for behavioral refinement is the one mentioned at start of paper: (1) should hold for all main programs c. Basically, that is our definition of $\preccurlyeq_=$, but there are some other concerns as explained in [CN00].

The standard technique for proving such a relation is simulation. Even for data refinement of ordinary modules (without dynamic binding and inheritance) it is known that two forms of simulation, forward and backward, are needed for this to be a complete proof technique even when nondeterminacy is bounded [HHS86]. So it is a little surprising that forward simulation has been proposed as a definition for class refinement (e.g. [LW94,BMvW97,MS98]). In any case, simulation needs to be shown sound in the sense that it implies the relation we call $\preccurlyeq_=$. We are aware of few such results (except in the context of concurrent objects using method call traces rather than pre-post specifications). One such result appears in [BMvW97] for a restricted language and main programs of a fixed shape.

Our main conjecture says that if $cds, ci \rhd cd \preccurlyeq cd'$ then $cds \rhd cd \preccurlyeq_= cd'$. Here $cds, ci \rhd cd \preccurlyeq cd'$ expresses that there is a forward simulation using coupling relation ci, as we explain below. We are confident in the result but call it a conjecture because some details have not been checked at the time of writing. Our work on this result has been fruitful in uncovering subtleties. For example, in (1) the algorithmic refinement relation $\sqsubseteq$ needs to be defined in terms of predicates that are typable in the state space of the command c. Also, we ought consider only commands that do not refer to globals of the type being refined. Otherwise, the comparison would be between commands in different state spaces. The corresponding result for behavioral subclassing is even more interesting due to type tests and casts, as mentioned earlier.

Predicate transformers seem to be at a good level of abstraction for such proofs. Despite our rigorous treatment of typing and so forth, the proofs are comprehensible and tractable by hand with diligence.

For a command c in the context of a method body for the refined class, the forward simulation condition for coupling invariant ci takes the form:

$$(\exists\, vs \bullet ci \wedge [\![\Gamma, N_S \rhd c]\!]\eta\ \psi) \Rightarrow [\![\Gamma', N_S \rhd c']\!]\eta'\ (\exists\, vs \bullet ci \wedge \psi) \qquad (2)$$

for all suitable postconditions ψ. Here vs are the abstract variables used in cd; the coupling invariant relates them to concrete variables vs' used in cd'. Also, Γ and η (respectively, Γ' and η') are environments for $cds\ cd$, N_S, and c (respectively, $cds\ cd'$, N_S, and c'), and N_S is the class declared by both cd and cd'.

Outside these classes, commands should not need to be modified, as only private attributes of the classes are changed. Nevertheless, even though these commands cannot access or affect the attributes being changed, their weakest precondition semantics is defined in terms of postconditions that can refer to these attributes. Therefore, we define a forward simulation relation for all commands. A function gci is needed to generalize the coupling invariant ci from the state space of the refined class to state spaces in other classes (and to impose suitable existential quantifications). We omit some arguments to gci in the forward simulation condition for commands:

$$gci\ ci\ ([\![\Gamma, N \rhd c]\!]\eta\ \psi) \Rightarrow [\![\Gamma', N \rhd c]\!]\eta'\ (gci\ ci\ \psi) \qquad (3)$$

We show that this holds for all c provided (2) holds for method bodies of the refined class. For predicates that do not refer to the private attributes of the refined class, gci reduces to the identity function; this "identity extension lemma" is used to derive the main soundness result from (3).

In set theoretic models, it is easy to define the extension of a coupling relation to arbitrary data types and state spaces [BMvW97,Nau00b], but gci must act in purely syntactic terms on formulas, which is one of the main challenges in obtaining our result. The benefit is that it is the syntactic form that is needed in proof rules anyway, if such rules are to be expressed at the level of specifications.

4 The authors

For reasons of funding, we expect that only Naumann can attend the workshop.

Cavalcanti extended Morgan's refinement calculus to give sound semantics and proof rules for recursive procedures [CSW99]. She is writing a book with Jim Woodcock on a refinement calculus for Z, based on their joint research [Cav97,CW98]. She has designed one of the object-oriented Z extensions, called MooZ [MC90,MC92,MCS94]. She is interested in semantics for object-oriented languages in general [MSM$^+$00].

Currently she is also working with Sampaio on laws for transforming specifications of concurrrent systems written in a combination of CSP and Z to Java programs. This work is based on a Java library that implements occam programming primitives.

Our joint work also grew out of Naumann's set theoretic predicate transformer semantics for a refinement calculus based on Oberon, using transformers of semantic predicates [Nau00a]. This language includes extensible records and stored procedures which can encode dynamic dispatch and inheritance. The semantics is based on an abstract categorical analysis of predicate transformers at higher types [Nau95b,Nau98b]. That analysis has also been used to extend calculational programming methods for functional programs [BdMH96] to higher order imperative programming with inductive data types [Nau94,Nau98c,Nau98a]. Naumann has studied data refinement both in abstract categorical settings [Nau95a] and for the Oberon-like language [Nau00b]. He is currently writing an undergraduate textbook on data structures in Java with emphasis on data refinement.

With Kedar Namjoshi, Naumann is currently working on decidable logics for reasoning about pointers. In particular, they are using model checking to decide questions of aliasing and also dynamic typing of references in Java.

Our expectation is that the workshop will be a fruitful forum in which to discuss what are the most crucial research issues and promising directions. In particular we are interested in becoming more familiar with the work of others on specification and abstraction and on reasoning about pointers. We expect that techniques like dependencies (for "modifies" specifications) will fit smoothly in our setting, and that the usual model of the heap as an array can be adapted to our setting; it will be very helpful to discuss these features and others, with researchers who are currently dealing with them.

We are also keen to explore possible connections with operational semantics of Java. Our project plan originally called for development of a state-transformer model based on semantics of Idealized Algol, but it is not clear that a complete connection can be found in the near term. We would like a more concrete operational foundation in any case.

We would be especially interested to learn of semantics for Java that are compositional at the level of classes. We have some tentative ideas on extending our semantics with a "hook" that would represent a dynamic dispatcher, but this has not been worked out carefully.

Finally, we are interested in current experience and available libraries for theorem provers. Naumann has been talking with Shankar for several years about using PVS for the higher order refinement calculus, and we would like to take advantage of the considerable work that has already been done for Java.

References

[AL97] M. Abadi and K. R. Leino. A Logic of Object-oriented Programs. In *Proceedings TAPSOFT'1997*. Springer-Verlag, 1997.

[BdMH96] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6:1–28, 1996.

[BMvW97] R.J.R. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct object substitutability. Technical Report TUCS-TR-147, Turku Center for Computer Science, December 1997.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[Cav97] A. L. C. Cavalcanti. *A Refinement Calculus for Z*. PhD thesis, Oxford University Computing Laboratory, Oxford - UK, 1997. Technical Monograph TM-PRG-123, ISBN 00902928-97-X.

[CN99a] A. L. C. Cavalcanti and D. Naumann. A Weakest Precondition Semantics for an Object-oriented Language of Refinement. In J. M. Wing, J. C. P. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1439 – 1459. Springer-Verlag, September 1999.

[CN99b] A. L. C. Cavalcanti and D. Naumann. A Weakest Precondition Semantics for an Object-oriented Language of Refinement - Extended Version. Technical Report CS 9903, Stevens Institute of Technology, September 1999.

[CN00] A. L. C. Cavalcanti and D. Naumann. A Weakest Precondition Semantics for an Object-oriented Language of Refinement. *IEEE Transactions on Software Engineering*, 2000.

[CSW99] A. L. C. Cavalcanti, A. Sampaio, and J. C. P. Woodcock. An Inconsistency in Procedures, Parameters, and Substitution in the Refinement Calculus. *Science of Computer Programming*, 33(1):87 – 96, 1999.

[CW98] A. L. C. Cavalcanti and J. C. P. Woodcock. A Weakest Precondition Semantics for Z. *The Computer Journal*, 41(1):1 – 15, 1998.

[HHS86] J. He, C. A. R. Hoare, and J. W. Sanders. Data Refinement Refined. In G. Goos and H. Hartmants, editors, *ESOP'86 European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187 – 196, March 1986.

[Lei98] K. R. M. Leino. Recursive Object Types in a Logic of Object-oriented Programming. In C. Hankin, editor, *7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[LM98] K. R. M. Leino and R. Manohar. Joining Specification Statements. *Theoretical Computer Science*, 1998. To appear.

[LW94] B. H. Liskov and J. M. Wing. A Behavioural Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[MC90] S. R. L. Meira and A. L. C. Cavalcanti. Modular Object-Oriented Z Specifications. In J. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 173 – 192, Oxford - UK, December 1990. Springer-Verlag.

[MC92] S. R. L. Meira and A. L. C. Cavalcanti. MooZ Case Studies. In R. Barden, S. Stepney, and D. Cooper, editors, *Object Orientation in Z*, chapter 5, pages 37 – 58. Springer-Verlag, 1992.

[MCS94] S. R. L. Meira, A. L. C. Cavalcanti, and C. S. Santos. The Unix Filing System: A MooZ Specification. In K. Lano and H. Haughton, editors, *Object Oriented Specification Case Studies*, chapter 4, pages 80 – 109. Prentice-Hall, 1994.

[Mor94] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[MS98] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *Proceedings of ECOOP'98*, number 1445 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

[MSM$^+$00] L. C. S. Meneses, S. Soares, J. B. Meneses, H. Moura, and A. L. C. Cavalcanti. A framework for defining object-oriented languages using action semantis. In *Brazilian Symposium on Programming Languages 2000*, 2000. To appear.

[Nau94] David A. Naumann. A recursion theorem for predicate transformers on inductive data types. *Information Processing Letters*, 50:329–336, 1994.

[Nau95a] David A. Naumann. Data refinement, call by value, and higher order programs. *Formal Aspects of Computing*, 7:652–662, 1995.

[Nau95b] David A. Naumann. Predicate transformers and higher order programs. *Theoretical Computer Science*, 150:111–159, 1995.

[Nau98a] David A. Naumann. Beyond Fun: Order and membership in polytypic imperative programming. In Johan Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *Springer LNCS*, pages 286–314, 1998.

[Nau98b] David A. Naumann. A categorical model for higher order imperative programming. *Mathematical Structures in Computer Science*, 8(4):351–399, August 1998.

[Nau98c] David A. Naumann. Towards squiggly refinement algebra. In David Gries and Willem-Paul de Roever, editors, *Programming Concepts and Methods*, pages 346–365. Chapman and Hall, 1998. Proceedings of IFIP PROCOMET '98.

[Nau00a] David A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtypes. *Science of Computer Programming*, 2000. To appear.

[Nau00b] David A. Naumann. Soundness of data refinement for a higher order imperative language. *Theoretical Computer Science*, 2000. To appear.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

A Formal Methods Based Static Analysis Approach for Detecting Runtime Errors in Java Programs

S. Skevoulis

A Formal Methods Based Static Analysis Approach for Detecting Runtime Errors in Java Programs

Sotiris Skevoulis

School of Computer Science and Information Systems
Pace University, New York NY 10038, USA

Abstract. This paper discusses the foundation of a generic approach to statically analyze Java programs in order to detect potential errors (bugs). We discuss a framework that supports our approach and carries out the static analysis of Java code automatically. Our approach can automatically detect such potential bugs and report them before the program is executed. For a Java class, invariants related to the category of error under examination are automatically generated and used to assess the validity of variable usage in the implementation of this class. A research prototype is under development. It assists us in evaluating the feasibility and effectiveness of our approach.

1 Introduction

In recent years extensive research effort has been made in the area of formal methods with remarkable results. This effort though, did not have much practical impact in the sense of acceptance within the software development industry. The reason for such a misfortune can be found in certain constraints and burdens that have been imposed by the promise, coming from the formal methods research community, that the use of formalization could bring absolute correctness in designs and implementations. If we allow formal methods to aim at more modest goals than absolute correctness and absence of errors, then we may be able to apply it at a relatively low cost in mainstream software industry. The detection of common programming errors is definitely such a goal. Our goal is to provide a practical mechanism to assist the application of formal methods in specific areas of software development such as the detection of common potential errors in programs at compile time.

Specifically, we work on a generic approach to statically analyze Java programs in order to detect potential errors (bugs). Our approach is based on the concept of class invariant. Java language has been selected as the target language in our research, for a number of reasons: it is quickly becoming one of the most widely used programming languages in software industry and it is more manageable than C and C++, since no pointer arithmetic is allowed, the semantics of its control flow is simpler without the goto statement and offers widely used class libraries with documentation and source code available for analysis.

Class invariants related to the category of error under examination are automatically generated and used to assess the validity of variable usage in the implementation of this class. Our approach bridges two areas namely, formal methods and static analysis under a unified framework. It is distinctive in its emphasis to provide a practical generic mechanism for error detection that is capable of addressing error detection for a variety of error categories via a web of specialized components. Our research effort is experimental in nature. Its success can be judged through experiments rather than theoretical proofs and analyses. Experimentation is used as a feedback mechanism to our theoretical studies and solutions. The goal is to gain and demonstrate effectiveness, through experimentation. A research prototype is under development and a number of test cases have been examined. Small-scale projects have been tried and the results are being evaluated. The key characteristics of our approach could be summarized as follows:

Generic detection. The approach presented here is both specific enough to detect errors deterministically and generic enough to provide an efficient and effective framework for an extensive variety of error categories.

Full automation. No effort is needed from the software practitioner. The entire process of analyzing the source code and reporting potential violations is fully automatic.

Absence of formal specifications. No need for formal specifications. Indeed, we recover the relevant specifications from the source code.

Extensibility. User provided specifications in the form of simple annotations, can be easily incorporated in order to strengthen the capability of the analysis component.

Existing static analysis techniques use flow based analysis to detect errors and fail to address the above classes of run-time errors. Over the past twenty years there have been approaches that intend to verify fully the functional correctness of programs. These approaches very often use subsets of existing programming languages as their targets.[2]. Ongoing research work at DEC laboratories has resulted in *Extended Static Checking* (ESC) [6] and checking object invariants [5]. They attempt to use formal methods to identify particular kinds of bugs in a programming language and provide also some kind of feedback to the programmer about those potential bugs. ESC translates each procedure into a guarded command representation similar to Dijkstra's approach. These research efforts require the presence of some kind of specification given by the developer, either in the form of simple program annotations or even more complicated formal specifications. Our approach generates automatically the relevant specifications.

2 Our Approach

The goal of the research work that is presented here is to develop a static analysis technique that can detect program bugs that cannot be caught by compilers and flow-control approaches. We have identified a restricted domain of potential run-time errors that can be detected by analyzing the source code and developed algorithms that will allow us to detect these during compilation time. Automated discharge of generated proof obligation is achieved by a properly tailored general purpose *theorem prover* with tactics and specialized decision procedures.

Complete source code verification is infeasible and impractical partly due to the high complexity and size of software systems that are currently built. As Jackson and Wing [3] have suggested, it is not necessary to insist in absolute formalization in developing software models. We follow this lightweight approach by limiting the scope of our detection mechanism to gain automation and determinism. The selection of the potential run-time errors to be included in the restricted domain of our research depends on the frequent occurrence in everyday programming tasks so their early detection will have a positive impact in the quality of the developed software and their automatic detection potential. Currently we focus on detecting illegal dereference and array bounds violations. Such an approach poses challenges in the generation of the class invariants regarding the restricted domain of potential errors, the formulation of the appropriate verification conditions to detect potential violations and the tailoring a general purpose light-weight automatic theorem prover with the appropriate heuristics and decision procedures to carry out the proofs automatically

2.1 Foundation

The theoretical foundation is based on the definition of *weakest precondition.* We have extended the rules for partial correctness of a small language as defined by Gries in order to be able to cover syntactically and semantically richer Java language. Our approach includes mechanisms of reasoning about the partial correctness of Java programs with respect to specific potential errors. The extension assumes that an expression e has no side effect. In general, this is not the case in most programming languages and in Java in particular. In order to make the application of these rules to Java expressions possible, we need to consider a subset of Java language that includes *only* expressions with no side effects. We defined a subset of Java language in which we enforce the following rule: *One expression is not allowed to have side effects.* We do so by providing a small number of transformation rules,[7] that transform compound expressions, into *semantically equivalent* ones. In doing so, we are able to cover the entire Java language without lessening the effectiveness of our approach.

In the following set of rules, we use the notation $R[x :=3D\ e]$ to denote textual substitution on predicate R as defined by Gries [1]. If we consider the predicate P as the precondition and the predicate R as the desired postcondition upon termination of execution of a number of statements, `op` is a binary operator in Java and `L` is a statement label, then the following set of rules applies:

Empty Statement:

$$\frac{\text{true}}{\{P\}\ ;\ \{P\}}$$

$$\frac{P \Rightarrow R}{\{P\}\ ;\ \{R\}}$$

Assignment Statement:

$$\frac{\text{true}}{\{R[x{:=}((e]\}\ x{=}((e;\ \{R\}}$$

$$\frac{P \Rightarrow R[x{:=}((e]}{\{P\}\ x{=}((e;\ \{R\}}$$

Composition of Statements:

$$\frac{\{P\}\ S_1\ ;\ \{Q\},\{Q\}\ S_2\ ;\ \{R\}}{\{P\}\ S_1;\ S_2;\ \{R\}}$$

If-Else Statement

$$\frac{\{P\wedge B\}\ S_1;\ \{R\},\{P\wedge\neg B\}\ S_2;\ \{R\}}{\{P\}\ \mathit{if}\,(B)\ S_1;\ \mathit{else}\ S_2;\ \{R\}}$$

Break Statement:

$$\frac{\{P\}\Rightarrow\{R\}}{\{P\}\ \mathit{break}[\mathit{id}];\ \{R\}}$$

Continue Statement:

$$\frac{\{P\}\Rightarrow\{R\}}{\{P\}\ \mathit{continue}[\mathit{id}];\ \{R\}}$$

Return Statement:

$$\frac{\{P\}\Rightarrow\{R\}}{\{P\}\ \mathit{return}\ v;\ \{R\}}$$

Throw Statement:

$$\frac{\{P\}\Rightarrow\{R\}}{\{P\}\ \mathit{throw}\ e;\ \{R\}}$$

Labeled Statement:

$$\frac{L\colon\ \{P\}\ S;\ \{R\}}{\{P\}\ L\colon\ S;\ \{R\}}$$

Synchronized Statement:

$$\frac{\{P\}\ S;\ \{R\}}{\{P\}\ \mathit{synchronized}\ (e)\ S;\ \{R\}}$$

While Statement:

$$\frac{\{P\wedge B\}\ S;\ \{P\}}{\{P\}\ \mathit{while}(B)\ S;\ \{P\wedge\neg B\}}$$

While Statement: (cont'd)

$$\frac{P\wedge B\Rightarrow I,\ \{I\wedge B\}\ S;\ \{I\},\ I\wedge\neg B\Rightarrow Q}{\{P\}\ \mathit{while}(B)\ S;\ \{Q\}}$$

Switch Statement:

$$\frac{\{P\wedge B_1\}\ S_1;\ \{R\},\ \ldots\ \{P\wedge B_n\}\ S_n;\ \{R\},\{P\wedge\neg B_1\wedge\ \ldots\ \wedge\neg B_n\}\Rightarrow\{R\}}{\{P\}\ \mathit{switch}(\mathit{exp})\ \mathit{case}(B_1)\colon S_1;\ \ldots\ \mathit{case}(B_n)\colon S_n;\ \{R\}}$$

Try Statement:

$$\frac{\{P\wedge Q_{E_1}\}\ S_t;\ S_1;\ S_f\{R\},\ \ldots\ \{\ P\wedge\neg Q_{E_1}\ \wedge\ldots\ \wedge\neg Q_{E_n}\}\ S_f;\ \{R\}}{\{P\}\ \mathit{try}\ S_t;\ \mathit{catch}(E_1)\ S_1;\ \ldots\ \mathit{catch}(E_n)\ S_n;\ [\mathit{finally}\ S_f;\,]\ \{R\}}$$

where En is an expression of *ExceptionType* e, S_t, S_f, S_n are Java statements and Q_{E_n} is a condition created from the input in `catch` clause E_n:

e `instanceof` *ExceptionType*

The detection mechanism of our approach has certain limitations, mainly due to the intractability of the theorem proving process. We can not guarantee absolute success in finding *all* the code anomalies even for the restricted types of analysis discussed earlier. Our goal is not to find *all* potential errors rather to find the majority of them automatically. Despite the limitations, experimental results show that the set of potential errors that our approach can detect is substantial. We used our prototype tool to run an extensive number of test cases for both types of specialized analysis (null pointer and array bounds) and the results were encouraging. Our approach did effectively detect the majority of program anomalies in the test cases.

2.2 The Algorithms

We have developed two generic algorithms[4, 7]: *DetermineInvariant* and *Check-Violation* which are the cornerstone of our approach. The former accepts as input a Java class and returns a predicate that is satisfied by all non-transient instances of this class. It reads in the class fields, both static and non-static, and forms a candidate invariant. It breaks down the formed invariant into two predicates: one that expresses a condition about static variables only and one about all the class level variables. Invariant is broken down into two sets in order to examine separately the properties that do not depend on the instantiation of the Java class from the ones that do. Each predicate is examined separately and if it fulfills the requirements of an invariant, is added to the invariant of the class. The output of the algorithm is the invariant for the Java class under examination. The main steps of the algorithm, are as follows.

1. construct two sets of predicates as a candidate static and instance invariants from all class level class variables and break it down into a number of possible combinations of them, each one of which will be tried to verify if it satisfies the invariant property.
2. mutate the candidate for static invariant into a number of mutated forms.
3. determine a static invariant checking each of the mutated predicates from the static candidate invariant set until an invariant is found.
4. mutate the candidate for instance invariant into a number of mutated forms.
5. determine an instance invariant checking each of the mutated predicates from the candidate invariant set until an invariant is found.
6. return the union of those two sets of invariants as the invariant set of predicates for the class.

As soon as the invariants have been detected, we use them to assess the validity of particular usage of variables. This is illustrated below:

```
void aMethod(...) {
   $\{Invariant\}$           the pre-condition
   // ...                 other statements
   $\{\neg isNull(v)\}$      the assertion we attempt to prove
   v.m(...)               the dereference
   // ...                 other statements
}
```

We can now check the usage of each relevant variable to detect any possible violations with *CheckViolation* algorithm. It scans though the implementation of the class, locates all the relevant variables and forms the appropriate verification conditions. An overview of the CheckViolation algorithm follows. The term *variable of interest* refers to the variables related to the class of potential errors that the algorithm attempts to detect. For each use of a variable of interest in *StaticBlock*, the algorithm forms the verification condition and prove it using as precondition the predicate *true*. It repeats the same process for each constructor (using as precondition the StaticInvariant) and public method with precondition the class invariant.

3 Future Work

The research outlined here is currently applied to intra class - intra method cases. Our work continues in extending the foundation to cover a more comprehensive analysis of Java classes. Our work will evolve in two main directions in order to provide: stronger analysis techniques which will cover more complicated cases within the already defined categories of potential bugs and an extensive array of algorithms capable of handling different cases. The inter-class and inter-methods domain analysis to automatically extract relevant specifications for null pointer and array index is well underway. Possible directions of this effort also include: stronger analysis by incorporating user provided simple and intuitive annotations in a notation closely resembling the Java notation and broader coverage of kinds of errors that can be automatically checked. These include but are not limited to: illegal downcasting, string index out of bounds, etc.

References

1. D. Gries. *The science of Programming.* Springer-Verlag, 1981.
2. S. V. Group. Stanford pascal verifier user manual. Technical report, Stanford University Computer Science Department, 1979. STAN-CS-79-731.
3. D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, 29(4):21, Apr. 1996.
4. X. Jia and S. Skevoulis. A generic approach of static analysis for detecting run-time errors in java programs. In *Proc. 23rd Annual Int'l Computer Software and Applications Conf.*, Phoenix, Arizona, Oct. 1999.
5. K. R. M. Leino. Ecstatic: An object-oriented programming language with axiomatic semantics. In *Proc. FOOL4*, 1997. Fourth International Workshop on Foundations of Object-Oriented Languages.
6. K. R. M. Leino and R. Stata. Checking object invariants. Technical report, Digital Equipment Corporation Research Center, 1997. Palo Alto, CA.
7. S.Skevoulis. A light-weight approach to applying formal methods in software development. Technical report, DePaul University, 1999. PhD dissertation. Available as Technical Report from DePaul University, `http://www.cs.depaul.edu`.

Axiomatic Semantics for Java^{light}

D. von Oheimb

Axiomatic Semantics for Java$^{\ell ight}$
– Extended Abstract –

David von Oheimb[⋆]

Technische Universität München
http://www.in.tum.de/~oheimb/

Abstract. We introduce a Hoare-style calculus for a nearly full subset of sequential Java, which we call Java$^{\ell ight}$.
This axiomatic semantics has been proved sound and complete w.r.t. our operational semantics of Java$^{\ell ight}$, described in earlier papers. The proofs also give new insights into the role of type-safety. All the formalization and proofs have been done with the theorem prover Isabelle/HOL.

1 Introduction

Since languages like Java are widely used in safety-critical applications, verification of object-oriented programs has grown more and more important. A first step towards verification seems to be developing a suitable axiomatic semantics (a.k.a. "Hoare logic") for such languages.

Recently several proposals for Hoare logics for object-oriented languages, e.g. [dB99,PHM99,HJ00], have been given. They deal with some small core language and are partially proved sound (on paper), but are known to be incomplete or at least have not been proved complete. Our new logic, in part inspired by [PHM99], has the following special merits.

– Apart from static overloading and dynamic binding of methods as well as references to dynamically allocated objects, it also covers full exception handling, static fields and methods, and static initialization of classes. Thus our sequential sublanguage Java$^{\ell ight}$ is almost the same as Java Card[Sun99].
– Instead of modeling expressions with side-effects as assignments to intermediate variables, it handles all expressions and variables first-class. Thus programs to be verified do not need to undergo an artificial structural transformation.
– It is both sound – w.r.t. a mature formalization of the operational semantics of Java – and complete. This means that programs using even non-trivial features like mutual recursion, dynamic binding, and static initialization can be proved correct.
– Apart from being rigorously and unambiguously defined (in the interactive theorem proving system Isabelle/HOL[Pau94]), it has been proved sound and complete within the system. This gives maximal confidence in the results obtained.

⋆ Research funded by the DFG Project Bali, http://isabelle.in.tum.de/Bali/

2 Some basics of the Java$^{\ell ight}$ formalization

Our axiomatic semantics inherits all features concerning type declarations and the program state from our operational semantics of Java$^{\ell ight}$. See [ON99] for a more detailed description.

Here we just recall that a program Γ (which serves as the context for most judgments) consists of a list of class and interface declarations and that the execution state is defined as

datatype $st = \mathsf{st}\ (globs)\ (locals)$
types $state = xcpt\ option \times st$

where *globs* and *locals* map class references to objects (including class objects) and variable names to values, respectively, and *xcpt* references an exception object. Using the projection operators on tuples, we define e.g. $\mathsf{normal}\ \sigma \equiv \mathsf{fst}\ \sigma = \mathsf{None}$, which expresses that in state σ there is no pending exception, and write $\mathsf{snd}\ \sigma$ to refer to the state without the information on exceptions, typically denoted by s.

A term of Java$^{\ell ight}$ is either an expression, a statement, a variable, or an expression list, and has a corresponding result. For uniformity, even a statement has a (dummy) result, called Unit. The result of a variable is an *lval*, which is a value (for read access) and a state update function (for write access).

types $terms = (expr + stmt) + var + expr\ list$
types $vals\ = val + lval + val\ list$
types $lval\ = val \times (val \to state \to state)$

There are many other auxiliary type and function definitions which we cannot define here for lack of space. The complete Isabelle sources, including an example, may be obtained from `http://isabelle.in.tum.de/Bali/src/Bali4/`.

3 The axiomatic semantics

3.1 Assertions

In our axiomatic semantics we shallow-embed assertions in the meta logic HOL, i.e. define them as predicates on (basically) the state, making the dependence on the state explicit and simplifying their handling within Isabelle. This general approach is extended in two ways.

– We let the assertions depend also on so-called *auxiliary variables* (denoted by the meta variable Z of any type α), which are required to relate variable contents between pre- and postconditions, as discussed in [Sch97].
– We extend the state by a stack (implemented as a list and denoted by Y) of result values of type res, which are used to transfer results between Hoare triples. In an operational semantics, these nameless values can be referred to via meta variables, but in an axiomatic semantics, such a simple technique is impossible since all values in a triple are logically bound to that scope (by universal quantification).

As a result, we define the type of assertions (with parameter α) as

types $\alpha\ assn = res\ list \times state \to \alpha \to bool$
datatype $res = \mathsf{Res}\ (vals) \mid \mathsf{Xcpt}\ (xcpt\ option) \mid \mathsf{Lcls}\ (locals) \mid \mathsf{DynT}\ (tname)$

We write e.g. $\mathsf{Val}\ v$ as an abbreviation for $\mathsf{Res}\ (\mathsf{In1}\ v)$, injecting a value v into *res*. Names like Val and DynT are used not only as constructors, but also as (destructor) patterns. For example, $\lambda\mathsf{Val}\ v{:}Y.\ f\ v\ Y$ is a function on the result stack that expects a value v as the top element and passes it to f together with the rest of the stack, referred to by Y.

In order to keep the Hoare rules short and thus more readable, we define several assertion (predicate) transformers.

- $\lambda s:\ P\ s \equiv \lambda(Y,\sigma).\ P\ (\mathsf{snd}\ \sigma)\ (Y,\sigma)$ allows P to peek at the state directly.
- $P \wedge.\ p\ \equiv \lambda(Y,\sigma)\ Z.\ P\ (Y,\sigma)\ Z \wedge p\ \sigma$ means that not only P holds but also p (applied to the program state only). The assertion $\mathsf{Normal}\ P \equiv P \wedge.\ \mathsf{normal}$ is a simple application stating that P holds and no exception has occurred.
- $P{\leftarrow}{:}f \equiv \lambda(Y,\sigma).\ P\ (Y,f\ \sigma)$ means that P holds for the state transformed by f.
- $P\ ;.\ f \equiv \lambda(Y,\sigma')\ Z.\ \exists\sigma.\ P\ (Y,\sigma)\ Z \wedge \sigma' = f\ \sigma$ means that P holds for some state σ and the current state is then derived from σ by the state transformer f.

3.2 Hoare triples and validity

We define triples as judgments of the form $prog{\vdash}\{\alpha\ assn\}\ terms{\succ}\ \{\alpha\ assn\}$ with some obvious variants for the different sorts of terms, e.g.
$\Gamma{\vdash}\{P\}\ e{\text{-}}{\succ}\ \{Q\} \equiv \Gamma{\vdash}\{P\}\ \mathsf{In1}(\mathsf{Inl}\ e){\succ}\ \{Q\}$ and $\{P\}\ .c.\ \{Q\} \equiv \{P\}\ \mathsf{In1}(\mathsf{Inr}\ c){\succ}\ \{Q\}$.

Here we simplify the presentation by leaving out triples as assumptions within judgments, which are necessary to handle recursion; we have discussed this issue in detail in [Ohe99]. The validity of triples is defined as

$$\begin{aligned}\Gamma{\models}\{P\}\ t{\succ}\ \{Q\} \equiv \forall Y\ \sigma\ Z.\ \ & P\ (Y,\sigma)\ Z \longrightarrow \mathsf{type\_ok}\ \Gamma\ t\ \sigma \longrightarrow \\ & \forall v\ \sigma'.\ \ \Gamma{\vdash}\sigma\ {-}t{\succ}{\rightarrow}\ (v,\sigma') \longrightarrow Q\ (\mathsf{res}\ t\ v\ Y,\sigma')\ Z\end{aligned}$$

where Y stands for the result stack and Z denotes the auxiliary variables. The judgment $\mathsf{type\_ok}\ \Gamma\ t\ \sigma$ means that the term t is well-typed (if σ is a normal state) and that all values in σ conform to their static types. This additional precondition is required to ensure soundness, as discussed in §3.5. $\Gamma{\vdash}\sigma\ {-}t{\succ}{\rightarrow}\ (v,\sigma')$ is the evaluation judgment from the operational semantics meaning that from the initial state σ the term t evaluates to a value v and final state σ'. Note that we define partial correctness.

Unless t is statement, the result value v is pushed onto the result stack via $\mathsf{res}\ t\ v\ Y \equiv \mathsf{if}\ \mathsf{is\_stmt}\ t\ \mathsf{then}\ Y\ \mathsf{else}\ \mathsf{Res}\ v{:}Y$.

3.3 Result value passing

We define the following abbreviations for producing and consuming results:

- $P{\uparrow}{:}w\ \equiv \lambda(Y,\sigma).\ P\ (w{:}Y,\sigma)$ means that P holds where the result w is pushed.
- $\lambda w{:}.\ P\ w \equiv \lambda(w{:}Y,\sigma).\ P\ w\ (Y,\sigma)$ expects and pops a result w and asserts $P\ w$.

A typical application of the former is the rule for literal values v:

$$Lit\ \frac{}{\Gamma{\vdash}\{\mathsf{Normal}\ (P{\uparrow}{:}\mathsf{Val}\ v)\}\ \mathsf{Lit}\ v{\text{-}}{\succ}\ \{P\}}$$

Analogously to the well-known assignment rule, it states that for a literal expression (i.e., constant) v the postcondition P can be derived if P – with the value v inserted – holds as the precondition and the (pre-)state is normal.

The rule for array variables handles result values in a more advanced way:

$$AVar\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e_1\text{-}\!\succ\ \{Q\}\qquad \Gamma\vdash\{Q\}\ e_2\text{-}\!\succ\ \{\lambda\mathsf{Val}\ i\!:\!.\ \mathsf{RefVar}\ (\mathsf{avar}\ \Gamma\ i)\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e_1[e_2]\text{=}\!\succ\ \{R\}}$$

where $\mathsf{RefVar}\ vf\ P \equiv \lambda(\mathsf{Val}\ a\!:\!Y,(x,s)).\ \mathsf{let}\ (v,x') = vf\ a\ x\ s\ \mathsf{in}\ (P\!\uparrow\!:\!\mathsf{Var}\ v)\ (Y,(x',s))$. Both subexpressions are evaluated in sequence, where Q as intermediate assertion typically involves the result of e_1. The final postcondition R is modified for the proof on e_2 as follows: from the result stack two values are expected and popped, namely i (the index) and a (an address) of e_2 and e_1, respectively. Out of these and the intermediate state (x,s), the auxiliary function avar computes the variable v, which is pushed as the final result, and (possibly) an exception x'.

For terms involving a condition, we define the assertion $P\!\uparrow\!:\!\mathsf{Bool}\!=\!b \equiv \lambda(Y,\sigma)\ Z.\ \exists v.\ (P\!\uparrow\!:\!\mathsf{Val}\ v)\ (Y,\sigma)\ Z \wedge (\mathsf{normal}\ \sigma \longrightarrow \mathsf{the\_Bool}\ v = b)$ expressing (basically) that the result of a preceding boolean expression is b. Together with the meta-level conditional expression ($\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$) depending on b and $P'\!\uparrow\!:\!\mathsf{Bool}\!=\!b$ identifying b with the result of a boolean expression e_0, we can describe both branches of conditional terms with a single triple, like in

$$Cond\ \frac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e_0\text{-}\!\succ\ \{P'\}\quad \forall b.\ \Gamma\vdash\{P'\!\uparrow\!:\!\mathsf{Bool}\!=\!b\}\ (\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\text{-}\!\succ\ \{Q\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e_0\ \ ?\ e_1\ \ :\ e_2\text{-}\!\succ\ \{Q\}}$$

The value b is universally quantified, such that when applying this rule, one has to prove its second antecedent for any possible value, i.e., both True and False. What is a notational convenience here (to avoid two triples, one for each case), will be essential for the *Call* rule, given below.

The rules for the standard statements appear almost as usual:

$$Skip\ \frac{}{\Gamma\vdash\{P\}\ .\mathsf{Skip}.\ \{P\}}\qquad Loop\ \frac{\Gamma\vdash\{P\}\ e\text{-}\!\succ\ \{P'\}\quad \Gamma\vdash\{P'\!\uparrow\!:\!\mathsf{Bool}\!=\!\mathsf{True}\}\ .c.\ \{P\}}{\Gamma\vdash\{P\}\ .\mathtt{while}(e)\ c.\ \{P'\!\uparrow\!:\!\mathsf{Bool}\!=\!\mathsf{False}\}}$$

Note that in all[1] rules (except Loop for obvious reasons) the postconditions of the conclusion is a variable. Thus in the typical “backward-proof” style of Hoare logic the rules are applied easily.

3.4 Dynamic binding

The great challenge of an axiomatic semantics for an object-oriented language is dynamic binding in method calls, for two reasons.

First, the code selected depends on the class D dynamically computed from a reference expression e. The range of values for D depends on the whole program and thus cannot be fixed locally, in contrast to the two possible boolean values appearing in conditional terms described above. Standard Hoare triples cannot express such an unbound case distinction. We handle this problem with the strong technique given above, using universal quantification and the precondition $R\!\uparrow\!:\!\mathsf{DynT}\ D \wedge\ldots$ with the special result value $\mathsf{DynT}\ D$. An alternative solution is

[1] The rules not mentioned here may be found in the appendix.

given in [PHM99], where D is referred to via This and the possible variety of D is handled in a cascadic way using several special rules.

Second, the actual value D often can be inferred statically, but in general for invocation mode "virtual", one can only know that it is a subtype of some reference type rt computed by static analysis during type-checking. The intuitive – but absolutely non-trivial – reason why the subtype relation $\mathsf{Class}\ D \preceq \mathsf{RefT}\ rt$ holds is of course type-safety. The problem here is how to establish this relation. [PHM99] simply places it into the precondition of the consequence of the appropriate rule, but in general this puts a heavy burden on the rule user, making the calculus at least practically incomplete. In contrast, our solution puts the relation (as the formula $\Gamma \vdash mode \rightarrow D \preceq rt$) into the precondition of an antecedent and thus provides the user with an additional helpful assumption, transferring the proof burden once and for all to the soundness proof.

The remaining parts of the rule for method calls deals with the unproblematic issues of argument evaluation, setting up the local variables (including parameters) of the called method and restoring the previous local variables on return, for which we use the special result value Lcls.

$$Call\ \frac{\begin{array}{c}\Gamma \vdash \{\mathsf{Normal}\ P\}\ e\text{-}\!\succ\ \{Q\} \\ \Gamma \vdash \{Q\}\ args \doteq\!\succ\ \{\lambda \mathsf{Vals}\ vs{:}\mathsf{Val}\ a{:}.\ \lambda s{:}\ \mathsf{let}\ D = \mathsf{dyn\_class}\ mode\ s\ a\ \tau\ \mathsf{in} \\ \quad R{\uparrow}{:}\mathsf{DynT}\ D{\uparrow}{:}\mathsf{Lcls}\ (\mathsf{locals}\ s) {\leftarrow}{:}\mathsf{init\_lvars}\ \Gamma\ D\ (mn,pTs)\ mode\ a\ vs\} \\ \forall D.\ \Gamma \vdash \{R{\uparrow}{:}\mathsf{DynT}\ D\ \wedge.\lambda\sigma.\ \mathsf{normal}\ \sigma \longrightarrow \Gamma \vdash mode \rightarrow D \preceq rt\} \\ \mathsf{Body}\ D\ (mn,pTs)\text{-}\!\succ\ \{\lambda \mathsf{Val}\ v{:}\mathsf{Lcls}\ l{:}.\ S{\uparrow}{:}\mathsf{Val}\ v {\leftarrow}{:}\mathsf{set\_lvars}\ l\}\end{array}}{\Gamma \vdash \{\mathsf{Normal}\ P\}\ \{rt,\tau,mode\}e.mn(\{pTs\}args)\text{-}\!\succ\ \{S\}}$$

3.5 Soundness and completeness

With the help of Isabelle/HOL, we have proved soundness and completeness:

$$\mathsf{wf\_prog}\ \Gamma \implies \Gamma \models \{P\}\ t\!\succ\ \{Q\} \ = \ \Gamma \vdash \{P\}\ t\!\succ\ \{Q\}$$

where $\mathsf{wf\_prog}\ \Gamma$ means that the program Γ is well-formed. As usual, soundness is proved by rule induction on the derivation of triples. Surprisingly, type-safety plays a crucial role here. The important fact that for method calls the subtype relation $\mathsf{Class}\ D \preceq \mathsf{RefT}\ rt$ holds can be derived only if the state conforms to the environment. This was the reason for bringing the judgment $\mathsf{type\_ok}$ into our definition of validity, which also gives rise to the new rule (required for the completeness proof)

$$hazard\ \frac{}{\Gamma \vdash \{P\ \wedge.\ \mathsf{Not} \circ \mathsf{type\_ok}\ \Gamma\ t\}\ t\!\succ\ \{Q\}}$$

indicating that if at any time conformance was violated, anything could happen.

Completeness is proved (basically) by structural induction with the MGF approach discussed in [Ohe99]. This includes an outer auxiliary induction on the number of methods already verified, which requires well-typedness in order to ensure that for any program there is only a finite number of methods to consider. Due to class initialization, an extra induction on the number of classes already initialized is required.

References

dB99. Frank de Boer. A WP-calculus for OO. In *Foundations of Software Science and Computation Structures*, volume 1578 of *LNCS*. Springer-Verlag, 1999.

HJ00. Marieke Huisman and Bart Jacobs. Java program verfication via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering (FASE'00)*, LNCS. Springer-Verlag, 2000. to appear.

Ohe99. David von Oheimb. Hoare logic for mutual recursion and local variables. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *FST&TCS'99*, volume 1738 of *LNCS*, pages 168–180. Springer-Verlag, 1999.

ON99. David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1999. http://isabelle.in.tum.de/Bali/papers/Springer98.html.

Pau94. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994. For an up-to-date description, see http://isabelle.in.tum.de/.

PHM99. Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.

Sch97. Thomas Schreiber. Auxiliary variables and recursive procedures. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 697–711. Springer-Verlag, 1997.

Sun99. Sun. *Java Card Spec.*, 1999. http://java.sun.com/products/javacard/.

A The remaining rules

$$\mathit{conseq}\ \frac{\begin{array}{c}\forall Y\ \sigma\ Z\,.\ P\ \ (Y,\sigma)\ Z\ \longrightarrow (\exists P'\ Q'.\ \mathit{\Gamma}{\vdash}\{P'\}\ t{\succ}\ \{Q'\} \wedge (\forall w\ \sigma'.\\ (\forall Y'\ \ Z'.\ P'\ (Y',\sigma)\ Z' \longrightarrow Q'\ (\mathsf{res}\ t\ w\ Y',\sigma')\ Z') \longrightarrow Q\ \ (\mathsf{res}\ t\ w\ Y,\sigma')\ Z))\end{array}}{\mathit{\Gamma}{\vdash}\{P\}\ t{\succ}\ \{Q\}}$$

$$\mathit{Xcpt}\ \frac{}{\mathit{\Gamma}{\vdash}\{(\lambda(Y,\sigma).\ P\ (\mathsf{res}\ t\ (\mathsf{arbitrary3}\ t)\ \ Y,\sigma))\ \wedge.\ \mathsf{Not} \circ \mathsf{normal}\}\ t{\succ}\ \{P\}}$$

$$\mathit{Super}\ \frac{}{\mathit{\Gamma}{\vdash}\{\mathsf{Normal}\ (\lambda s:\ P{\uparrow}\,{:}\mathsf{Val}\ (\mathsf{val\_this}\ s))\}\ \mathtt{super}\text{-}{\succ}\ \{P\}}$$

$$\mathit{LVar}\ \frac{}{\mathit{\Gamma}{\vdash}\{\mathsf{Normal}\ (\lambda s:\ P{\uparrow}\,{:}\mathsf{Var}\ (\mathsf{lvar}\ vn\ s))\}\ \mathsf{LVar}\ vn{=}{\succ}\ \{P\}}$$

$$\mathit{FVar}\ \frac{\mathit{\Gamma}{\vdash}\{\mathsf{Normal}\ P\}\ .\mathsf{init}\ C.\ \{Q\} \qquad \mathit{\Gamma}{\vdash}\{Q\}\ e\text{-}{\succ}\ \{\mathsf{RefVar}\ (\mathsf{fvar}\ C\ \mathit{stat}\ \mathit{fn})\ R\}}{\mathit{\Gamma}{\vdash}\{\mathsf{Normal}\ P\}\ \{C,\mathit{stat}\}e.\mathit{fn}{=}{\succ}\ \{R\}}$$

$$\mathit{Acc}\ \frac{\mathit{\Gamma}{\vdash}\{\mathsf{Normal}\ P\}\ va{=}{\succ}\ \{\lambda\mathsf{Var}\ (v,f)\,{:}.\ Q{\uparrow}\,{:}\mathsf{Val}\ v\}}{\mathit{\Gamma}{\vdash}\{\mathsf{Normal}\ P\}\ \mathsf{Acc}\ va\text{-}{\succ}\ \{Q\}}$$

$$\mathit{Ass}\ \frac{\begin{array}{c}\mathit{\Gamma}{\vdash}\{\mathsf{Normal}\ P\}\ va{=}{\succ}\ \{Q\}\\ \mathit{\Gamma}{\vdash}\{Q\}\ e\text{-}{\succ}\ \{\lambda\mathsf{Val}\ v{:}\mathsf{Var}\ (w,f)\,{:}.\ R{\uparrow}\,{:}\mathsf{Val}\ v{\leftarrow}{:}\mathsf{assign}\ f\ v\}\end{array}}{\mathit{\Gamma}{\vdash}\{\mathsf{Normal}\ P\}\ va\mathtt{:=}e\text{-}{\succ}\ \{R\}}$$

$$\mathit{Nil}\ \frac{}{\{\mathsf{Normal}\ P{\uparrow}\,{:}\mathsf{Vals}\ []\}\ []{\doteq}{\succ}\ \{P\}}$$

$$\mathit{Cons}\ \dfrac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e\text{-}\!\succ\ \{Q\}\qquad \Gamma\vdash\{Q\}\ es\doteq\!\succ\ \{\lambda\mathsf{Vals}\ vs{:}\mathsf{Val}\ v{:}.\ R\!\uparrow{:}\mathsf{Vals}\ (v{:}vs)\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e{:}es\doteq\!\succ\ \{R\}}$$

$$\mathit{NewC}\ \dfrac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{init}\ C.\ \{\mathsf{Alloc}\ \Gamma\ (\mathsf{CInst}\ C)\ \mathit{id}\ Q\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ \mathtt{new}\ C\text{-}\!\succ\ \{Q\}}$$

where $\mathsf{Alloc}\ \Gamma\ \mathit{otag}\ f\ P\equiv$
$\lambda(Y,(x,s))\ Z.\ \forall\sigma'\ a.\ \Gamma\vdash(f\ x,s)\ {-}\mathsf{halloc}\ \mathit{otag}{\succ}a{\rightarrow}\ \sigma'\longrightarrow (P\!\uparrow{:}\mathsf{Val}\ (\mathsf{Addr}\ a))\ (Y,\sigma')\ Z$

$$\mathit{NewA}\ \dfrac{\begin{array}{c}\Gamma\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{init\_comp\_ty}\ T.\ \{Q\}\\ \Gamma\vdash\{Q\}\ e\text{-}\!\succ\ \{\lambda\mathsf{Val}\ i{:}.\ \mathsf{Alloc}\ \Gamma\ (\mathsf{Arr}\ T\ (\mathsf{the\_Intg}\ i))\ (\mathsf{check\_neg}\ i)\ R\}\end{array}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ \mathtt{new}\ T[e]\text{-}\!\succ\ \{R\}}$$

$$\mathit{Cast}\ \dfrac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e\text{-}\!\succ\ \{\lambda\mathsf{Val}\ v{:}.\ Q\!\uparrow{:}\mathsf{Val}\ v\leftarrow\!{:}\lambda(x,s).\ (\mathsf{raise\_if}\ (\neg\Gamma,s\vdash v\ \mathsf{fits}\ T)\ \mathtt{ClassCast}\ x,s)\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ \mathsf{Cast}\ T\ e\text{-}\!\succ\ \{Q\}}$$

$$\mathit{Inst}\ \dfrac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e\text{-}\!\succ\ \{\lambda\mathsf{Val}\ v{:}.\ \lambda s{:}\ (Q\!\uparrow{:}\mathsf{Val}\ (\mathsf{Bool}\ (v{\neq}\mathsf{Null}\wedge\Gamma,s\vdash v\ \mathsf{fits}\ \mathsf{RefT}\ T)))\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e\ \mathtt{instanceof}\ T\text{-}\!\succ\ \{Q\}}$$

$$\mathit{Body}\ \dfrac{\begin{array}{c}\mathsf{the}\ (\mathsf{cmethd}\ \Gamma\ C\ \mathit{sig})=(\mathit{md},\ \_,\ \_,\ \mathit{blk},\ \mathsf{res})\\ \Gamma\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{init}\ \mathit{md}.\ \{Q\}\qquad \Gamma\vdash\{Q\}\ .\mathit{blk}.\ \{R\}\quad \Gamma\vdash\{R\}\ \mathsf{res}\text{-}\!\succ\ \{S\}\end{array}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ \mathsf{Body}\ C\ \mathit{sig}\text{-}\!\succ\ \{S\}}$$

$$\mathit{Expr}\ \dfrac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e\text{-}\!\succ\ \{\lambda w{:}.\ Q\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{Expr}\ e.\ \{Q\}}\qquad \mathit{Comp}\ \dfrac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .c_1.\ \{Q\}\quad \Gamma\vdash\{Q\}\ .c_2.\ \{R\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .c_1;c_2.\ \{R\}}$$

$$\mathit{If}\ \dfrac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e\text{-}\!\succ\ \{P'\}\qquad \forall b.\ \Gamma\vdash\{P'\!\uparrow{:}\mathsf{Bool}{=}b\}\ .(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2).\ \{Q\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .\mathtt{if}(e)\ c_1\ \mathtt{else}\ c_2.\ \{Q\}}$$

$$\mathit{Throw}\ \dfrac{\Gamma\vdash\{\mathsf{Normal}\ P\}\ e\text{-}\!\succ\ \{\lambda\mathsf{Val}\ a{:}.\ Q\leftarrow\!{:}\lambda(x,s).\ (\mathsf{throw}\ a\ x,s)\}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .\mathtt{throw}\ e.\ \{Q\}}$$

$$\mathit{Try}\ \dfrac{\begin{array}{c}\Gamma\vdash\{\mathsf{Normal}\ P\}\ .c_1.\ \{Q\}\\ \Gamma\vdash\{(Q\ \wedge.\lambda\sigma.\ \ \Gamma,\sigma\vdash\mathsf{catch}\ C)\ ;.\ \mathsf{new\_xcpt\_var}\ \mathit{vn}\}\ .c_2.\ \{R\}\\ \Gamma\vdash\{Q\ \wedge.\lambda\sigma.\ \neg\Gamma,\sigma\vdash\mathsf{catch}\ C\}\ .\mathsf{Skip}.\ \{R\}\end{array}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .\mathtt{try}\ c_1\ \mathtt{catch}(C\ \mathit{vn})\ c_2.\ \{R\}}$$

$$\mathit{Fin}\ \dfrac{\begin{array}{c}\Gamma\vdash\{\mathsf{Normal}\ P\}.c_1.\{\lambda(Y,(x,s)).\ (Q\!\uparrow{:}\mathsf{Xcpt}\ x)\ (Y,(\mathsf{None},s))\}\\ \Gamma\vdash\{\mathsf{Normal}\ Q\}.c_2.\{\lambda\mathsf{Xcpt}\ x'{:}.\ R\leftarrow\!{:}\lambda(x,s).\ (\mathsf{xcpt\_if}\ (x'{\neq}\mathsf{None})\ x'\ x,s)\}\end{array}}{\Gamma\vdash\{\mathsf{Normal}\ P\}\ .c_1\ \mathtt{finally}\ c_2.\ \{R\}}$$

$$\mathit{Done}\ \dfrac{}{\Gamma\vdash\{\mathsf{Normal}\ (P\ \wedge.\ \mathsf{initd}\ C)\}\ .\mathsf{init}\ C.\ \{P\}}$$

$$\mathit{Init}\ \dfrac{\begin{array}{c}\mathsf{the}\ (\mathsf{class}\ \Gamma\ C)=(sc,\_,\_,\_,\mathit{ini})\quad \mathit{sup}=\mathsf{if}\ C=\mathtt{Object}\ \mathsf{then}\ \mathsf{Skip}\ \mathsf{else}\ \mathsf{init}\ sc\\ \Gamma\vdash\{\mathsf{Normal}\ ((P\ \wedge.\ \mathsf{Not}\circ\mathsf{initd}\ C)\ ;.\ \mathsf{supd}\ (\mathsf{new\_stat\_obj}\ \Gamma\ C))\}\ .\mathit{sup}.\ \{Q\!\uparrow{:}.\lambda s.\ \mathsf{Lcls}\ (\mathsf{locals}\ s)\}\\ \Gamma\vdash\{Q\ ;.\ \mathsf{set\_lvars\ empty}\}\ .\mathit{ini}.\ \{\lambda\mathsf{Lcls}\ l{:}.\ R\leftarrow\!{:}\mathsf{set\_lvars}\ l\}\end{array}}{\Gamma\vdash\{\mathsf{Normal}\ (P\ \wedge.\ \mathsf{Not}\circ\mathsf{initd}\ C)\}\ .\mathsf{init}\ C.\ \{R\}}$$

Provers for Behavioural Logics: Combining Model Checking with Deduction

A. Nakagawa

Provers for Behavioural Logics: Combining Model Checking with Deduction

Ataru T. Nakagawa

SRA Software Engineering Laboratory
Marusho Bldg. 5F, 3-12 Yotsuya, Shinjuku-ku, Tokyo 160-0004, Japon
E-mail: nakagawa@sra.co.jp, nakagawaa@acm.org

1 Introduction

We are constructing a system that supports automated theorem proving based on a combination of model checking and deduction. A major application area we have in mind is a distributed component service where a component is searched or invoked based on interface definitions and some semantic constraints. The system may also be of use in choosing appropriate codes from distributed programme libraries. In view of the current trends, java libraries are among the most promising targets for the system.

2 Preliminaries

The system deals with CafeOBJ, a language based on the combinations of (restricted) rewriting logic and (a variant of) hidden-algebraic logic. A subset of CafeOBJ is an algebraic specification language in the traditional sense. The aspect of rewriting logic, although heavily used for some case studies, is not used in this paper so that I give no explanation thereof.

The aspect of hidden-algebraic logic is the main concern of the system and the paper. In this logic, two kinds of sorts (or types) are distinguished. A visible sort is for an abstract data type, and its elements are indentified up to equality. A hidden sort is for a state space, and its elements are identified up to behavioural equivalence. Two elements are behaviourally equivalent iff they behave identically under any context.

As an illustration I show how current bank accounts are defined in CafeOBJ. To get rid of unnecessary complications, I modified keywords and conventions a little.

```
module ACCOUNT {
  importing (INT)
  *[ Account ]*
  bop balance : Account -> Int
  op open : -> Account
  op deposit : Nat Account -> Account
  op withdraw : Nat Account -> Account
  var N : Nat
```

```
    var A : Account
    eq balance(deposit(N,A)) = balance(A) + N .
    ceq balance(withdraw(N,A)) = balance(A) if N > balance(A) .
    ceq balance(withdraw(N,A)) = balance(A) - N
                                      if not N > balance(A) .
  }
```

- `module` introduces a specification unit. `ACCOUNT` is the name of this module.
- `importing` another module makes its sorts and operators available here. I assumed `INT` is a module that specifies integers `Int` and natural numbers `Nat`. Infix operators `+` and `-`, and a predicate `>` were assumed to be defined in `Int`.
- `*[,]*` enclose names of hidden sorts.
- `bop` introduces operators that define observational contexts. In the example `balance` is the only context.
- `op` introduces other operators.
- Both `bop` and `op` are followed by the name of the operator, `:`, the arity, `->`, and the coarity.
- `var` is a variable declaration.
- `eq` (`ceq`) asserts an unconditional (conditional) axiom. For the purpose of this paper, universally quantified (conditional) equations are the only forms axioms can take.
- A condition is an expression built out of user-defined predicates and such propositional connectives as `and` and `not`.

The module defined the sort `Account` to be the state space observable only by `balance` (getting the balance of an account). As their names suggest, `deposit` (`withdraw`) increases (decreases) a balance. An overdraft is not allowed. `open` is to open a new account, but the balance of a newly opened account is not defined. Thus an implementation that puts a bonus into a new account of a valuable client does satisfy this specification. The possibility of such underspecifications is one important advantage in using hidden-algebraic logic, in contrast to the traditional equational logic.

3 Asserting and Checking Properties

For a hidden-algebraic specification in CafeOBJ, our system allows the user to define a property, and to prove it to be an invariance. A general scheme is as follows. Let h be a hidden sort for which we want to show an invariant property, such as safety.

1. The user introduces a predicate P monadic on h.
2. He defines P with a (full) first-order logic with equality. Unlike the CafeOBJ proper, the system allows negations, existential quantifiers, and so on.

After these preparations from the user, the system takes charge.

3. For each “constant” c of sort h (see below), the system tries to prove $P(c)$ from the definition of P.
4. If there is c with which the system failed to prove $P(c)$, the proof failed. Otherwise,
5. Let $S = \{X\}$ where X is a variable of sort h.
   - 5-1. Let F be the set of operators monadic on h and with coarity h, and let $S' = \{f(\overline{Y}, t, \overline{Y'}) | f : \overline{s}h\overline{s'} \to h \in F, t \in S\}$ where $\overline{s}$, $\overline{s'}$ are (possibly empty) lists of sorts, and $\overline{Y}$, $\overline{Y'}$ are corresponding lists of variables.
   - 5-2. For each $t \in S'$,
     - 5-2-1. The system tries to find $t' \in S$ bisimilar to t. If such t' is found, let $S' = S' - \{t\}$ and continue. Otherwise,
     - 5-2-2. Writing down t as $f(\overline{Y}, t', \overline{Y'})$, the system tries to prove the implication $\forall\overline{X}.[P(t') \Rightarrow \forall\overline{Y}, \overline{Y'}.P(t)]$ where $\overline{X}$ is the list of variables that appear in t'.
     - 5-2-3. If the implication is proven, continue. Otherwise the overall proof failed.
   - 5-3. If S' is empty, the proof succeeded. Otherwise, let $S = S \cup S'$ and go back to 5-1.

The scheme is just a sketch, and there are a lot of technical details to fill in. For example, to find a bisimilar element at step 5-2-1. is not a trivial problem: in fact, since t may contain variables, a “bisimilar element” is even an abuse of language. The efficiency is also not taken into account. For example, at step 5-1. unnecessary repetitions may occur quite often. For this brief paper, however, I concentrate on the overall picture and give explanations on a couple of key points.

– A “constant” c at step 3. is relative to h, and can be any operator with coarity h, as long as h is not in its arity. For example, an operator

```
op open' : Nat -> Account
```

  is regarded as a constant of sort `Account` in this procedure. In such a case, the proof is on the universal closure of $P(c)$. Constants correspond to initial states when viewed from the state transition perspective.

– The condition of monadicity at step 5-1. is inherited from the underlying CafeOBJ. Its variant of hidden-algebraic logic restricts attention to monadic operators.

– S at step 5. acts as a set of states visited so far. At each iteration terms of one more depth, which correspond to one-step transitions, are considered. At 5-2-1. the system cuts out the transitions leading to states bisimilar to ones already visited. At 5-2-2., the system checks if the remaining transitions preserves P.

– That S' is empty at step 5-3. means every further one-step transition leads to a state already checked. To put the iteration at step 5. in another perspective, it is to calculate a fixed-point of an operator on the state space, and upon successful termination at 5-3. S is indeed a fixed-point.

– The procedure may not stop if S' remains non-empty forever. A control by a loop counter is necessary to implmenent an actual system.
– If you only want to check reachable states, the state space S may be initiated to the set of constants. The above scheme tries to prove a slightly stronger condition, and the user should be given a choice on which condition he wants to check.

As an example, let us see how to prove that an overdraft is not allowed for the currect accounts as defined in `ACCOUNT`. This property may be stated as a predicate `P` defined as

```
  def P(A:Account) = balance(A) >= 0 .
```

(Here `>=` is assumed to be defined in `INT` already.) As a matter of fact, since `open` is underspecified, this property is not an invariance. But we may at least show that, if an account is non-negative, it remains so forever. Apply step 5. to this example and you get the following result. Assume that `N1` etc. are variables of relevant sorts.

```
start: S = { A0 }
5-1.: S' = { deposit(N1,A1), withdraw(N2,A2) }
5-2.: for t = deposit(N1,A1) or withdraw(N2,A2),
5-2-1.: a bisimilar element is not found
5-2-2.: P(A1) (or P(A2)) => P(t) is proven
5-2-3.: continue
5-3.: S = { A0, deposit(N1,A1), withdraw(N2,A2) }
5-1.: S' = { deposit(N3,A3), withdraw(N4,A4),
             deposit(N5,deposit(N6,A6)),
             deposit(N7,withdraw(N8,A7)),
             withdraw(N9,deposit(N10,A8)),
             withdraw(N11,withdraw(N12,A9)) }
5-2. for each t in S',
5-2-1.: a bisimilar element is found;
        S' = S' - { t }
5-2-3.: continue
5-3.: S' is empty
```

In this example, just by one-step transitions the state spaces are exhausted.

4 Searching for Specifications

The proof system sketched in the previous section is intended to be part of a component search engine, to be invoked autonomously. Targets of a search procedure are specifications (of components) written in CafeOBJ. The search engine first restricts the search space by signature matching, and then checks which component, if any, satisfies the desired properties.

Signature matching consists of generating signature morphisms by purely syntactical means: given a source signature, for each component specification, the system tries to generate (possibly more than one) mappings of sorts, visibles to visibles and hiddens to hiddens, and those of operators, consistently. If predicates are also given, the system checks candidate components against that predicates, under the translations defined by the signature morphisms.

5 Implementation

A preliminary prototype was already implemented. It is an extension to a CafeOBJ processor, and contains the following features.

- To find a bisimilar element, a kind of context induction and a coinduction were implemented. The former is suitable for automatic invocation but there is no guarantee that the procedure terminates, and is controlled by a loop counter. The latter is useful if an easy congruent relation is found, but otherwise unfit for automatic proof procedures.
- As a prover for general first-order sentences, a resolution procedure was implemented. The implementation utilises the heterogeneousness of the carrier set, and a couple of experiments have shown that it leads to a drastic efficiency gain. For equational reasoning, paramodulation and demodulation are used.
- The model checking procedure explained in the paper is presented as a forward search but a backward search is also possible. The user may choose either.
- To have a realistic image of the system, a CORBA-based component search service is also implemented experimentally.
- On some technical details. CafeOBJ does not have a concept of predicate per se: there are only Boolean operators. The actual implementation did not deviate from this convention so that all the "predicate"s that appeared in the paper are actually Boolean operators. First-order sentences are also treated as plain Boolean expressions.

6 Final Remarks

Most of the current model checking tools deal with modal logics, such as LTL and CTL. In contrast, the target logic of our system is a first-order logic with equality. One reason behind this choice is that, for a software specification (as against a hardware specification), a naive formulation of state transitions will lead to space explosions, so that an abstraction at the level of state spaces is essential.

We made preliminary case studies of our language and system, mainly in the field of protocol specifications, Not every expression of modal logics (especially of CTL) has a counterpart in our logic, but most prominent examples of properties to check, such as various kinds of safeties, can be formulated in our language with ease. Moreover, the prototype did prove some of them within a reasonable timespan (the order is of minutes or seconds, not of hours), in spite of the fact that no tuning for efficiency was attempted. The results so far are thus encouraging.

We have not seen how large a component search space can be. A little experiment on a score of specifications, for various kinds of containers, has shown that the response time is of the order of seconds, again without any tuning for efficiency. If a search occurs in a more serious scale over a distributed, hence transmission-sensitive, environment, such usual tricks as cachings and encodings may become necessary. However, we have not yet worked out a stragegy on this point.

First Steps in Formalising JML

J. van den Berg, E. Poll, B. Jacobs

First steps in formalising JML: exceptions in predicates

Joachim van den Berg, Erik Poll, Bart Jacobs

Dept. Computer Science, Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{joachim,erikpoll,bart}@cs.kun.nl
http://www.cs.kun.nl/~{joachim, erikpoll, bart}

Abstract This note describes the first steps toward a rigorous formal semantics of JML specifications for Java. This semantics is developed within the framework of the LOOP project [5,11]. JML specifications stating invariants, pre- and post-conditions are expressed as Hoare sentences tailored to Java.

1 Introduction

The Java Modeling Language [7,6], JML for short, is a behavioural interface specification language tailored to Java. JML is a superset of Java. Using JML, a Java class can be specified by invariants, pre- and post-conditions, in the tradition of Eiffel [9] and following the 'Design by Contract' approach [10].

Our goal is to develop a formal semantics for JML to enable proof tool-assisted verification. Within the LOOP project [5,11] a semantics for sequential Java (*i.e.* a semantics for Java without threads) has already been developed (and used for many verifications). Our JML semantics extends this Java semantics.

The Java semantics is used in a special purpose compiler, the LOOP tool, which translates Java classes into their Java semantics. This semantics is generated as a series of logical theories of the proof tools PVS or Isabelle/HOL. Assisted by PVS or Isabelle/HOL, a user can then prove properties about these Java classes. A large invariant verification for Java's Vector class has shown that such proof tool-assisted verification is feasible [4]. The development of a formal semantics for JML makes it possible to extend the LOOP tool and to reason with JML specifications.

A design goal of JML is to use a notation which is "readily understandable by Java programmers" [6]. Therefore, its syntax is an extension of the Java expression syntax. This introduces several complications, namely the possibility of side-effects, abrupt termination, and even non-termination in predicates; these complications cannot arise in ordinary logical formulae. Any formal semantics for JML will have to deal with these issues. At some point we will have to transform Java expressions (which can terminate normally with true or false, or can terminate abruptly) into ordinary logical values true and false[1]. Moreover, ab-

[1] We do not want to use some three-valued logic.

ruptly terminating pre-conditions may lead to inconsistencies (see Subsection 4.3 below).

JML specifications stating invariants and pre- and post-conditions are expressed as Hoare sentences. Due to possible abrupt termination of statements and expressions in Java, traditional Hoare sentences are not adequate. Therefore, we use special Hoare sentences tailored to Java, as introduced in [3].

This note is organised as follows. It starts with a brief introduction to the type theoretic language that will be used. Section 3 describes our semantics of JML predicates. Section 4 describes the translation of behaviour specifications to Hoare sentences.

2 Technical preliminaries

This section describes very briefly the semantics of Java expressions. A more detailed description of this semantics can be found in [5,3,1].

Expressions in Java, and thus in JML as well, may hang, terminate normally, or terminate abruptly by throwing an exception. If an expression evaluated in a state x terminates normally, it produces a result and a successor state (because expressions may have side-effects). If its evaluation in state x terminates abruptly, a successor state together with a reference to an exception object (see [2, §15.5]) is produced. In our semantics, expressions are modelled as state transformer functions of the type below, where Self is the state space, Out the type of the expression, and $\mathsf{RefType}$ the type of references:

$$\mathsf{Self} \longrightarrow \mathsf{ExprResult}[\mathsf{Self}, \mathsf{Out}] \overset{\text{def}}{=} \begin{array}{l} \{\, \mathsf{hang} \quad : \mathsf{unit} \\ \mid \mathsf{norm} \quad : [\, \mathsf{ns}\colon \mathsf{Self}, \mathsf{res}\colon \mathsf{Out} \,] \\ \mid \mathsf{abnorm} : [\, \mathsf{es}\colon \mathsf{Self}, \mathsf{ex}\colon \mathsf{RefType} \,] \,\} \end{array}$$

The type $\mathsf{ExprResult}$ (with type parameters Self and Out) is a (labelled) variant type, which makes a distinction between non-termination (labelled hang), normal termination (labelled norm) and abrupt termination (labelled abnorm). For the latter two options, a labelled product type $[\_, \_]$ is used. Functions on a variant type like $\mathsf{ExprResult}$ are defined by pattern matching, using a CASES notation as below.

3 Semantics of JML predicates

JML extends the Java expression syntax with logical operators, like implication, `==>`, and the universal and existential quantors, `\forall` and `\exists` (see [6, §3.1] for a complete overview). Obviously, JML predicates have type `boolean`. Like for Java expressions, the semantics of a JML predicate P is written as $[\![\, P \,]\!]\colon \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{boolean}]$.

Just like there are two different meanings for disjunction in Java, namely `|` and `||`, there are two different meanings for implication, namely (1) `!p | q` and

(2) `!p || q`. The difference between these two options is in whether the right-hand operand is always evaluated. If the evaluation of `p` terminates normally, then in (1) `q` is always evaluated, but in (2) `q` is evaluated only if `p` evaluates to true. An example predicate, `j != 0 ==> 5/j != 0`, illustrates the difference: in (1) it terminates abruptly if `j` equals 0, in (2) it terminates normally for all `j`. Therefore, we choose $[\![$ `p ==> q` $]\!] = [\![$ `!p || q` $]\!]$.

Before fixing the semantics of the universal quantor, let's consider an example: the predicate `\forall (int j) 5/j != 0`, where, obviously, `5/j` will terminate abruptly for `j` equals 0. We do not wish this universally quantified predicate to terminate normally, because it is not properly defined for all `j`. Thus, its semantics would be either (1) hang or (2) abnorm. We choose for option (2). In order to let this predicate terminate normally, an implication can be added: `\forall (int j) j != 0 ==> 5/j != 0`. Our semantics of a universally quantified predicate `\forall (`T `t)` P is: if for any element of type T predicate P terminates normally, then `\forall (`T `t)` P terminates normally, otherwise it terminates abruptly. Abrupt termination is caused by throwing a null reference instead of a reference to an exception object. Since abrupt termination with a null reference cannot be caused by Java statements and expressions, it identifies ill-defined quantified predicates. Summarising:

$$
\begin{array}{l}
\mathit{pred}\colon \mathsf{Out} \to \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{boolean}] \;\vdash \\
\quad \mathsf{FORALL} \cdot \mathit{pred} : \; \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{boolean}] \;\stackrel{\text{def}}{=} \\
\qquad \lambda x\colon \mathsf{Self}.\ \mathsf{IF}\ \forall (o\colon \mathsf{Out}).\, \mathsf{CASES}\ \mathit{pred} \cdot o \cdot x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad \mid \mathsf{hang}() \mapsto \mathsf{false} \\
\qquad\qquad\qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{false}\,\} \\
\qquad\quad \mathsf{THEN}\ \mathsf{norm}(\mathsf{ns} = x, \mathsf{res} = \forall (o\colon \mathsf{Out}).\mathsf{BE2B} \cdot (\mathit{pred} \cdot o) \cdot x) \\
\qquad\quad \mathsf{ELSE}\ \mathsf{abnorm}(\mathsf{es} = x, \mathsf{ex} = \mathsf{null}) \\
\qquad\quad \mathsf{ENDIF}
\end{array}
$$

where $\mathsf{BE2B}$ is a function that turns JML predicates into ordinary predicates; it is defined as

$$
\begin{array}{l}
p\colon \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{boolean}] \;\vdash \\
\quad \mathsf{BE2B} \cdot p : \; \mathsf{Self} \to \mathsf{boolean} \;\stackrel{\text{def}}{=} \\
\qquad \lambda x\colon \mathsf{Self}.\ \mathsf{CASES}\ p \cdot x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad \mid \mathsf{hang}() \mapsto \mathsf{false} \\
\qquad\qquad\qquad \mid \mathsf{norm}\, y \mapsto y.\mathsf{res} \\
\qquad\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{false}\,\}
\end{array}
$$

An example of an existentially quantified predicate is `\exists (int j) 5/j != 0`. Again, the division `5/j` will terminate abruptly for `j` equals 0. Similarly to universally quantified predicates, we define the interpretation of `\exists (int j) 5/j != 0` to be abrupt termination[2]. The semantics of `\exists` is

[2] For some, this might be counter-intuitive, since there exists a `j` satisfying `5/j != 0`, *e.g.* `j` equals 1. Our reason for not interpreting $\exists j.5/j \neq 0$ as true is that tools like

defined logically equivalent to the semantics of `\forall`, such that the property $\forall x.\phi = \neg\exists x.\neg\phi$ holds for JML quantors.

4 Semantics of JML specifications

This section will focus on JML specifications stating invariants and pre- and post-conditions for methods expressed in so-called behaviour specifications. There are several ways to specify a method's behaviour. Basically, each behaviour specification consists of a pre-condition (the `requires` clause), and a post-condition (in case of normal termination, the `ensures` clause, and/or in case of abrupt termination, the `signals` clause). Besides specifying a method's pre- and post-condition, a behaviour specification can specify much more, like a `modifiable` clause listing the fields which may be changed by the method. Clauses other than pre- and post-conditions are not considered here.

An example of a JML specification is:

```
public int firstElement (int[] array)
/*@ normal_behavior
  @   requires : array != null && array.length > 0;
  @    ensures : true;
  @*/
{ return array[0]; }
```

The JML predicates used here are `array != null && array.length > 0` and `true`.

In JML also invariants can be specified. An invariant should hold after object creation via one of the constructors, and should be preserved by all non-`private` methods. So, an invariant is implicitly included in pre- and post-conditions for each non-`private` method.

4.1 From JML specifications to Hoare sentences

The semantics of behaviour specifications can be expressed as Hoare sentences. Traditional Hoare sentences do not deal with abrupt termination, and therefore they are not suitable in this context. In [3] a Hoare logic tailored to Java is introduced, covering abrupt termination.

A `normal_behavior` specification is interpreted as follows. The proof obligation for a method m with a pre-condition $Pre\colon \mathsf{Self} \to \mathsf{boolean}$, a post-condition $Post\colon \mathsf{Self} \to \mathsf{boolean}$ is the total correctness Hoare sentence $[Pre]\ m\ [Post]$. This Hoare sentence says that for any state x such that $Pre \cdot x$, executing m in x terminates normally resulting in a successor state y such that $Post \cdot y$.

An `exceptional_behavior` specification is interpreted as follows. The proof obligation for a method m with a pre-condition $Pre\colon \mathsf{Self} \to \mathsf{boolean}$, a post-condition $Post\colon \mathsf{Self} \to \mathsf{RefType} \to \mathsf{boolean}$, an exception type $E\colon \mathsf{string}$ is the

PVS define this predicate to be logically equivalent to $\neg\forall j.5/j = 0$ (which is not defined for j equals 0).

total exception correctness Hoare sentence $[Pre]\, m\, [\mathsf{exception}(Post, E)]$. This Hoare sentence says that for any state x such that $Pre \cdot x$, executing m in x terminates abruptly, because of an exception e, resulting in a successor state y such that $Post \cdot y \cdot e$, and e is an instance of E.

Things are a bit more complicated, since the post-condition may contain expressions of the form $\texttt{\textbackslash old}(T)$ which refer to the value of T in the 'pre-state'. Therefore, the post-condition has type $\mathsf{Self} \to \mathsf{Self} \to \mathsf{boolean}$ instead of $\mathsf{Self} \to \mathsf{boolean}$. Thus, the proof obligation generated for a `normal_behavior` specification becomes: for any state x such that $Pre \cdot x$, executing m in x terminates normally resulting in a successor state y such that $Post \cdot x \cdot y$. To express this as a Hoare sentence, we use the standard technique of introducing logical variables. Concretely, we introduce a logical variable $Z \colon \mathsf{Self}$ and take as pre-condition $Pre' = \lambda x \colon \mathsf{Self}.\, Pre \cdot x \wedge x = Z$ and as post-condition $Post' = \lambda x \colon \mathsf{Self}.\, Post \cdot Z \cdot x$. The resulting Hoare sentence becomes $[Pre']\ m\ [Post']$, which is equivalent to the proof obligation above.

4.2 From JML Boolean expressions to truth values

As explained above, given pre- and post-conditions of type $\mathsf{Self} \to \mathsf{boolean}$, behaviour specifications can be expressed as Hoare sentences. However, pre- and post-conditions in JML have type $\mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{boolean}]$. Earlier, we defined the function BE2B to convert JML predicates into logical predicates. Using BE2B, a `normal_behavior` specification with a pre-condition P and a post-condition Q would yield the following proof obligation:

$$[\mathsf{BE2B} \cdot [\![P]\!]]\ m\ [\mathsf{BE2B} \cdot [\![Q]\!]]$$

But what does this mean if the pre-condition P terminates abruptly? Applying BE2B to P then yields false, and makes the whole proof obligation trivially true. Obviously, this is not what we want. One option is to say that BE2B should be partial and then the proof obligation would not be well-defined. Because we want to generate proof obligations for PVS, in which the use of partial functions is not practical, we define a total function BE2B_pre.

$$
\begin{array}{l}
p \colon \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{boolean}] \ \vdash \\
\quad \mathsf{BE2B\_pre} \cdot p : \ \mathsf{Self} \to \mathsf{boolean} \ \stackrel{\text{def}}{=} \\
\qquad \lambda x \colon \mathsf{Self}.\ \mathsf{CASES}\ p \cdot x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad \mid \mathsf{hang}() \mapsto \mathsf{true} \\
\qquad\qquad\qquad \mid \mathsf{norm}\, y \mapsto y.\mathsf{res} \\
\qquad\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{true}\ \}
\end{array}
$$

Using this function for converting pre-conditions, non-terminating and abruptly terminating JML predicates are interpreted as true, so that they become useless, as assumptions. This makes the proof obligation as hard as possible. The obligation for the `normal_behavior` specification is thus:

$$\big[\mathsf{BE2B\_pre} \cdot [\![P]\!]\big]\ m\ \big[\mathsf{BE2B} \cdot [\![Q]\!]\big]$$

If there is also an invariant I, the specification's semantics becomes:

$$\big[\mathsf{BE2B\_pre} \cdot [\![I \ \texttt{\&\&}\ P]\!]\big]\ m\ \big[\mathsf{BE2B} \cdot [\![I \ \texttt{\&\&}\ Q]\!]\big]$$

4.3 Example

We wish to illustrate that abruptly terminating pre-conditions should be avoided with our semantics. Consider therefore:

```
public void copyInto (int[] from, int[] to);
/*@ normal_behavior
  @   requires : from.length <= to.length;
  @    ensures : true;
  @ also
  @ exceptional_behavior
  @   requires : from.length > to.length;
  @    signals : (MyException e) true;
  @*/
```

These behaviour specifications are expressed as a conjunction of Hoare sentences:

$$\begin{array}{l}
\big[\mathsf{BE2B\_pre} \cdot [\![\texttt{from.length <= to.length}]\!]\big] \\
\texttt{copyInto(from, to)} \\
\big[\mathsf{BE2B} \cdot [\![\texttt{true}]\!]\big] \\
\quad\wedge \\
\big[\mathsf{BE2B\_pre} \cdot [\![\texttt{from.length > to.length}]\!]\big] \\
\texttt{copyInto(from, to)} \\
\big[\mathsf{exception}(\mathsf{BE2B} \cdot [\![\texttt{true}]\!], \texttt{MyException})\big]
\end{array}$$

Applying BE2B_pre to the pre-conditions in case one of the parameter arrays is a null reference will make them true. Thus, in this case abruptly terminating pre-conditions make this specification inconsistent: method `copyInto` will have to terminate both normally and abruptly. Adding non-null reference checks, `from != null && to != null`, to the pre-conditions prevents abrupt termination of these predicates. The proof obligation generated for this behaviour specification is then consistent.

5 Conclusion

We have given a strict interpretation of JML specifications (closely following Java), specifically by using BE2B_pre (instead of BE2B) for pre-conditions. Thus a pre-condition `a[2] == b[3]` is problematic, and requires a lengthy alternative:

`a != null && a.length > 2 && b != null && b.length > 3 && a[2] == b[3]`, called a "protective" specification in [8]. In the end, this is a matter of choice. We think it is best to make all assumptions explicit, because after all this the point of specification languages.

The approach described in this paper treats JML predicates as much as possible as Java Booleans, following the prescribed Java evaluation order for *e.g.* `&&` and `||`. We are currently experimenting to evaluate its practicality by verifying JML-annotated programs using PVS. An alternative is to use the intented JML semantics mentioned in [6], where partial functions are modelled as underspecified total functions. This would require some effort to change the LOOP tool, since such a JML semantics would no longer be an extension of the Java semantics; for example, `5/i == 0 || i == 0` for `i` equals 0 would no longer terminate abruptly, but would be true.

References

1. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in Lect. Notes Comp. Sci. Springer, Berlin, 2000.
2. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
3. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci. Springer, Berlin, 2000.
4. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Rep. CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen, 2000.
5. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.
6. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 1998, revised May 2000.
7. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
8. G.T. Leavens and J.M. Wing. Protective interface specifications. *Formal Aspects of Computing*, 10:59–75, 1998.
9. B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.
10. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2^{nd} rev. edition, 1997.
11. LOOP Project. `http://www.cs.kun.nl/~bart/LOOP/`.

A Dynamic Logic for Java Card

B. Beckert

A Dynamic Logic for Java Card

Bernhard Beckert

University of Karlsruhe
Institute for Logic, Complexity and Deduction Systems
D-76128 Karlsruhe, Germany
`i12www.ira.uka.de/~beckert`

Abstract. In this paper, I describe a Dynamic Logic for JAVA CARD and outline a sequent calculus for this logic that axiomatises JAVA CARD. The purpose of the logic is to provide a framework for software verification that can be integrated into real-world software development processes.

1 Introduction

Design principles and goals. The work that is reported in this paper has been carried out as part of the KeY project [1]. The goal of KeY is to enhance a commercial CASE tool with functionality for formal specification and deductive verification and, thus, to integrate formal methods into real-world software development processes. Accordingly, the design principles for the software verification component of the KeY system are:

- The programs that are verified should be written in a "real" object-oriented (OO) programming language.
- The logical formalism should be as easy as possible to use for software developers (that do not have years of training in formal methods).

Java Card. We use JAVA CARD [10, 5] (soon to be replaced by Java 2 Micro Edition, J2ME) as the target programming language. JAVA CARD is a "real" OO language and has, accordingly, features that are difficult to handle such as dynamic data structures, exceptions, and initialisation; but it lacks some crucial complications of the full JAVA language such as threads and dynamic loading of classes. JAVA smart cards are an extremely suitable application for software verification: (a) JAVA CARD applications are small (JAVA smart cards currently offer 32K memory for code); (b) at the same time, JAVA CARD applications are embedded into larger program systems or business processes which should be modeled (though not necessarily formally verified); (c) JAVA CARD applications are often security-critical, giving incentive to apply formal methods; (d) the high number of deployed smart cards constitutes a new motivation for formal verification, as arbitrary updates are not feasible.

Dynamic Logic. We use Dynamic Logic (DL) [6], which is an extension of Hoare logic [3], as the logical basis of the KeY system's software verification component. We believe that this is a good choice because deduction in DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of JAVA CARD.

DL is successfully used in the KIV software verification system [9] for a programming language that is not object-oriented; and Poetzsch-Heffter and Müller's definition of a Hoare logic for a JAVA subset [8] shows that there are no principal obstacles to adapting the DL/Hoare approach to OO languages.

DL can be seen as a modal predicate logic with a modality $\langle p \rangle$ for every program p (we allow p to be any legal JAVA CARD program); $\langle p \rangle$ refers to the successor worlds (called states in the DL framework) that are reachable by running the program p. In classical DL there can be several such states (worlds) because the programs can be non-deterministic; but here, since JAVA CARD programs are deterministic, there is exactly one such world—if p terminates—or there is no such world—if p does not terminate. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds. A formula $\phi \rightarrow \langle p \rangle \psi$ is valid if for every state s satisfying precondition ϕ a run of the program p starting in s terminates, and in the terminating state the postcondition ψ holds.

Thus, the formula $\phi \rightarrow \langle p \rangle \psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$. But in contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators: In Hoare logic, the formulas ϕ and ψ are pure first-order formulas, whereas in DL they can contain programs. DL allows to involve programs in the descriptions ϕ resp. ψ of states. For example, using a program, it is easy to specify that a data structure is not cyclic, which is impossible in pure first-order logic. Also, JAVA constructs such as `instanceof` are available in DL for the description of states. It is, therefore, not necessary to define an abstract data type *state* and to represent states as terms of that type (as has, for example, been done in [8]); instead DL formulas can be used to give a (partial) description of states, which is a more flexible technique and allows to concentrate on the relevant properties of a state.

In comparison to classical DL (that uses a simple "artificial" programming language), a DL for a "real" OO programming language like JAVA CARD has to cope with the following complications:

– A program state does not only depend on the value of program variables but also on the values of the attributes of all existing objects.
– The evaluation of a JAVA expression may have side effects; thus, there is a difference between an expression and a logical term.
– Language features such as built-in data types, exception handling, and object initialisation have to be handled.

2 Syntax of Java Card DL

The non-dynamic part of our DL is basically a typed first-order predicate logic. To define its syntax, we have to specify its sets of variables, its types, and its terms (which we often call "logical terms" in the following to emphasise that they are different from JAVA expressions). Then, we define what the programs of the DL are. In the programs that are part of a DL formula, we allow an extension of JAVA CARD, where logical terms may occur in place of expressions of the same type. Finally, the syntax of DL formulas and sequents is defined.

Context. We do not allow class definitions in the programs that are part of DL formulas, but define syntax and semantics of DL formulas w.r.t. a given JAVA CARD program (the context), i.e., a sequence of class definitions. With the following restrictions any syntactically legal JAVA CARD program may be used: A context must not contain occurrences of *local inner classes*; and `break` and `continue` must be used with (explicit) labels. These restrictions are "harmless" because any JAVA CARD program can easily be transformed accordingly.

We assume that the following methods and fields are implicitly defined for each class *Cls* in the context and can thus be used in DL formulas (but not in the context). They allow to access information about the program state that is otherwise inaccessible in JAVA: a list of all existing objects of a class and information on whether a class resp. its objects are initialised. The objects of a certain class are considered to be organised into an (infinite) ordered list; this list is used by `new` to "create" objects (intuitively, `new` changes the attributes `lastCreatedObj` of the class and sets the attribute `created` of the new object to `true`, see Section 4).

```
public static Cls firstObj;  // the first object in the list,
                             // whether already created or not
public static Cls lastCreatedObj;  // the last created object,
                                   // null if no object exists
public Cls prevObj;  // the previous object in the list,
                     // null if for the first object
public Cls nextObj;  // the next object in the list
public boolean beforeObj(Cls obj);  // returns true if this
                                    // is before obj in the list
public boolean created;  // true if the object has already been
                         // created with new, and false otherwise
public static boolean classInitialised;  // true if the class resp.
public boolean objInitialised;           // the object is initialised
```

Variables. In classical DL there is only one type of variables. Here however, to avoid confusion, we use two kinds of variables.

Program variables are denoted with `x`, `y`, `z`, ... Their value can differ from state to state and can be changed by programs. They occur in programs and can also be used in the non-program parts of formulas (there they behave like modal constants, i.e., constants whose value can differ from state to state). Program variables cannot be quantified and they cannot be instantiated with terms.

Logical variables are denoted with x, y, z, ... They are assigned the same values in all states; a statement such as "x `= 1;`", which tries to change the value of the logical variable x, is illegal. Logical variables must be bound by a quantifier, free occurrences are not allowed; they can be instantiated with terms (preserving syntactical correctness of a formula but not necessarily its satisfiability or validity).

Types. The set of types of our DL contains (a) the primitive types of Java Card (`boolean`, `byte`, `short`), (b) the classes (object types) defined in the context, (c) the built-in classes such as `String`, and (d) an array type for each of the types in (a)–(c). In addition, there are user-defined types; typically these are abstract data types. There is no type hierarchy, i.e., no sub-typing concept.

Terms. Logical terms are constructed as usual from program variables, logical variables, and the constant and function symbols of all types. The set of terms includes in particular all Java Card literals for the primitive types, string literals, and the `null` object reference literal.

In addition, (a) if o is a term of class type C (i.e., denotes an object) and `a` is a field (attribute) of class C, then o`.a` is a term. (b) If *`Class`* is a class name and *`a`* is a static field of *`Class`*, then *`Class.a`* is a term. (c) If a is an array type term and i is a term of type `byte`, then $a[i]$ is a term.

Programs. The programs in DL formulas are executable code; as said above, they are not allowed to contain class declarations. The (basic) programs are the legal Java Card statements, including: (a) expression statements such as "`x = 1;`" (assignments), "`m(1);`" (method calls), "`i++;`", "`new Cls;`", local variable declarations (which restrict the "visibility" of program variables); (b) blocks and compound statements built with `if-else`, `switch`, `for`, `while`, and `do-while`; (c) statements with exception handling using `try-catch-finally`; (d) statements that abruptly redirect the control flow (`throw`, `return`, `break`, `continue`); (e) labelled statements; (f) the empty statement.

The technique for handling method calls in a DL calculus is to syntactically replace the call by the method's implementation. To handle the `return` statement in the right way, it is necessary to record the program variable or attribute that the result is to be bound to and to mark the boundaries of the implementation when it is substituted for the method call. For that purpose, we allow statements of the form `call(`*`x`*`=`*`m`*`(`arg_1`,...,`arg_n`)){`*`prog`*`}` to occur in DL programs.

In addition, we allow programs in DL formulas (not in the context) to contain logical terms. Wherever a Java Card expression can be used, a term of the same type as the expression can be used as well. Accordingly, expressions can contain terms (but not vice versa).

Formulas. Formulas are built as usual from the (logical) terms, the predicate symbols of all the types and the equality predicate $\doteq$, the logical connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, the quantifiers $\forall$ and $\exists$ (that can be applied to logical variables but not to program variables), and the modal operator $\langle \mathtt{p} \rangle$, i.e., if $\mathtt{p}$ is a program and ϕ is a formula, then $\langle \mathtt{p} \rangle \phi$ is a formula as well.

If o is a variable of some class type C, then a quantification such as $(\forall o)\phi(o)$ ranges over the (infinite) set of all objects of type C whether they have been created or not. The fact that all *created* objects of class C have a certain property ϕ can be expressed using the formula $(\forall o)(o.\mathtt{created} \doteq \mathtt{true} \rightarrow \phi(o))$.

To simplify notation, we allow *updates* of the form $\{\mathtt{x} \leftarrow t\}$ resp. $\{o.\mathtt{a} \leftarrow t\}$ to be attached to terms and formulas, where $\mathtt{x}$ is a program variable, o is a term denoting an object with attribute $\mathtt{a}$, and t is a term. The intuitive meaning of an update is that the term or formula that it is attached to is to be evaluated after changing the state accordingly, i.e., $\phi^{\{\mathtt{x} \leftarrow t\}}$ has the same semantics as $\langle \mathtt{x}\texttt{=}t \rangle \phi$ (but is easier to handle because the evaluation of t is known to have no side effects).

Sequents. A sequent is of the form $\phi_1, \ldots, \psi_m \ \vdash \ \psi_1, \ldots, \psi_n$ $(m, n \geq 0)$, where the ϕ_i and ψ_j are DL formulas. The meaning of a sequent is that the conjunction of the ϕ_i's implies the disjunction of the ψ_j's.

3 Semantics of Java Card DL

To define the semantics of Java Card DL we use the semantics of the Java Card programming language. In case of doubt, we refer to the precise formal semantics of Java defined by Börger and Schulte [4] using Abstract State Machines.[1]

The models of DL are Kripke structures consisting of possible worlds that are called states. All states of a model share the same universe containing a sufficient number of elements of each type.

The function and predicate symbols that are not user-defined—such as the equality predicate and the function symbols of the primitive Java Card types—have a fixed interpretation. In all models they are interpreted according to their intended semantics resp. their meaning in the Java Card language.

Logical variables are interpreted using a (global) variable assignment; they have the same value in all states of a model.

States. In each state a (possibly different) value (an element of the universe) of the appropriate type is assigned to: (a) the program variables, (b) the attributes (fields) of all objects, (c) the class attributes (static fields) of all classes in the context, and (d) the special object variable `this`. Variables and attributes of object types can be assigned the special value *null*.

Note, that states do not contain any information on control flow such as a program counter or the fact that an exception has been thrown.

[1] Following another approach, Nipkow and von Oheimb have obtained a precise semantics of a Java sublanguage by embedding it into Isabelle/HOL; they also use an axiomatic semantics [7].

Programs and Formulas The semantics of a program p is a state transition, i.e., it assigns to each state s the set of all states that can be reached by running p starting in s. Since Java Card is deterministic, that set either contains exactly one state or is empty. The set of states of a model must be closed under the reachability relation for all programs p, i.e., all states that are reachable must exist in a model (other models are not considered).

The semantics of a logical term t occurring in a program is the same as that of an expression whose evaluation is free of side-effects and gives the same value as t.

For formulas ϕ that do not contain programs, the notion of ϕ being satisfied by a state is defined as usual in first-order logic. A formula $\langle p \rangle\phi$ is satisfied by a state s if the program p, when started in s, terminates in a state s' in which ϕ is satisfied. A formula is satisfied by a model M, if it is satisfied by one of the states of M. A formula is valid in a model M if it is satisfied by all states of M; and a formula is valid if it is valid in all models.

We consider programs that terminate abnormally to be non-terminating. Examples are a program that throws an uncaught exception and a `return` statement that is not within the boundaries of a method invocation. Thus, for example, $\langle$`throw x;`$\rangle\phi$ is unsatisfiable for all ϕ. Nevertheless, it is possible to express and (if true) prove the fact that a program p terminates abnormally (and, for example, throws an exception) using a sequence such as

$$\texttt{e} \doteq \texttt{null} \ \vdash\ \langle \texttt{try\{}p\texttt{\}catch\{Exception e\}}\rangle(\neg\, \texttt{e} \doteq \texttt{null}) \ .$$

Sequents. The semantics of a sequent $\phi_1, \ldots, \psi_m \ \vdash\ \psi_1, \ldots, \psi_n$ is the same as that of the formula $(\phi_1 \wedge \ldots \wedge \psi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n)$.

4 A Sequent Calculus for Java Card DL

In this section we outline the ideas behind the sequent calculus for Java Card DL, and we present some of the basic rules.[2]

The DL rules of our calculus operate on the first *active* command p of a program $\pi p\, \omega$. The non-active prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try`-`catch` blocks, and beginnings "`call(...){`" of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the commands `throw`, `return`, `break`, and `continue` that abruptly change the control flow can be handled appropriatly.[3]

[2] These are simplified versions of the actual rules. In particular, initialisation of objects and classes is not considered.

[3] In classical DL, where no prefixes are needed, any formula of the form $\langle p\, q \rangle\phi$ can be replaced by $\langle p \rangle\langle q \rangle\phi$. In our calculus, splitting of $\langle \pi p q \omega\rangle\phi$ into $\langle \pi p \rangle\langle q \omega\rangle\phi$ is not possible (unless the prefix π is empty) because πp is not a valid program; and the formula $\langle \pi p\, \omega\rangle\langle \pi\, q\, \omega\rangle\phi$ cannot be used either because its semantics is in general different from that of $\langle \pi p q\, \omega\rangle\phi$.

Assignment Rule. The assignment rule is the most important rule of the DL calculus:

$$\frac{\Gamma^{\{x \leftarrow c\}},\ x \doteq \mathit{expr}^{\{x \leftarrow c\}} \ \vdash\ \langle \pi\omega \rangle \phi,\ \Delta^{\{x \leftarrow c\}}}{\Gamma \ \vdash\ \langle \pi\ x\ \texttt{=}\ \mathit{expr}\,\texttt{;}\ \omega \rangle \phi,\ \Delta} \qquad c \text{ is a new constant} \tag{1}$$

In classical DL, rule (1) is always applicable; here however, we have to impose a restriction: this rule can only be used if the expression *expr* is a logical term. Otherwise, other rules have to be applied first to evaluate *expr* (as that evaluation may have side effects). For example, these rules replace the formula $\langle \texttt{x = ++i;} \rangle \phi$ by $\langle \texttt{i = i+1; x = i;} \rangle \phi$.

Moreover, the handling of updates is more difficult in Java Card DL: In classical DL $\phi^{\{x \leftarrow c\}}$ is equivalent to the formula that is constructed from ϕ by syntactically replacing the left side x of the update by the right side c. Now however, because several object variables may refer to the same object, more complex rules have to be used to simplify the result $\phi^{\{o.\texttt{a} \leftarrow c\}}$ of an update of an object (or class) attribute.

Rule for Creating Objects. The `new` statement is treated by the calculus as if it were implemented as follows (this implementation accesses the fields that are implicitly defined for all classes, see the explanation in Section 2):

```
public static Cls new() {
  if (lastCreatedObj == null)
      lastCreatedObj = firstObj;
  else
      lastCreatedObj = lastCreatedObj.nextObj;
  lastCreatedObj.created = true;
  return lastCreatedObj;
}
```

Rules for Loops. The following rules allow to "unwind" `while` loops. These are simplified versions that only work if (a) *cnd* is a logical term (and, thus, its evaluation does not have side effects), and (b) *prg* does not contain a `continue` statement. Similar rules are defined for `do-while` and `for` loops.

$$\frac{\Gamma \ \vdash\ \mathit{cnd} \doteq \texttt{true},\ \Delta \qquad \Gamma \ \vdash\ \langle \pi\ \mathit{prg}\ \texttt{while}\,\texttt{(}\mathit{cnd}\texttt{)}\,\mathit{prg}\ \ \omega \rangle \phi,\ \Delta}{\Gamma \ \vdash\ \langle \pi\ \texttt{while}\,\texttt{(}\mathit{cnd}\texttt{)}\,\mathit{prg}\ \ \omega \rangle \phi,\ \Delta} \tag{2}$$

$$\frac{\Gamma \ \vdash\ \mathit{cnd} \doteq \texttt{false},\ \Delta \qquad \Gamma \ \vdash\ \langle \pi\omega \rangle \phi,\ \Delta}{\Gamma \ \vdash\ \langle \pi\ \texttt{while}\,\texttt{(}\mathit{cnd}\texttt{)}\,\mathit{prg}\ \ \omega \rangle \phi,\ \Delta} \tag{3}$$

These rules allow to handle loops if used together with induction schemata for the primitive and the user defined types, such as:

$$\frac{\Gamma \ \vdash\ \phi(c),\ \Delta \qquad \Gamma \ \vdash\ (\forall x)(\phi(x) \rightarrow \phi(f(x))),\ \Delta}{\Gamma \ \vdash\ (\forall x)\phi(x),\ \Delta} \tag{4}$$

(where the type of x is generated by c and f).

Rules for Handling Exceptions. The following rules allow to handle `try-catch-finally` blocks and the `throw` statement. Again, these are simplified versions of the actual rules; they are only applicable if (a) *exc* is a logical term (e.g., a program variable), and (b) the statements `break`, `continue`, and `return` do not occur.

$$\frac{\Gamma \vdash \mathit{instanceof}(\mathit{exc}, T) \quad \Gamma \vdash \langle\pi\ \mathtt{try\{}e\mathtt{=}\mathit{exc}\mathtt{;}\ q\mathtt{\}finally\{}r\mathtt{\}}\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \langle\pi\ \mathtt{try\{throw}\ \mathit{exc}\mathtt{;}\ p\mathtt{\}catch(}T\ e\mathtt{)\{}q\mathtt{\}finally\{}r\mathtt{\}}\ \omega\rangle\phi,\ \Delta} \tag{5}$$

$$\frac{\Gamma \vdash \neg\mathit{instanceof}(\mathit{exc}, T) \quad \Gamma \vdash \langle\pi\ r\mathtt{;}\ \mathtt{throw}\ \mathit{exc}\mathtt{;}\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \langle\pi\ \mathtt{try\{throw}\ \mathit{exc}\mathtt{;}\ p\mathtt{\}catch(}T\ e\mathtt{)\{}q\mathtt{\}finally\{}r\mathtt{\}}\ \omega\rangle\phi,\ \Delta} \tag{6}$$

$$\frac{\Gamma \vdash \langle\pi\ r\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \langle\pi\ \mathtt{try\{\}catch(}T\ e\mathtt{)\{}q\mathtt{\}finally\{}r\mathtt{\}}\ \omega\rangle\phi,\ \Delta} \tag{7}$$

Rule (5) applies if an exception *exc* is thrown that is an instance of exception class T, i.e., the exception is caught; otherwise, if the exception is not caught, rule (6) applies. Rule (7) applies if the `try` block is empty and, thus, terminates normally.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. Technical Report 2000/4, University of Karlsruhe, Department of Computer Science, Jan. 2000.
2. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java.* LNCS 1523. Springer, 1999.
3. K. R. Apt. Ten years of Hoare logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 1981.
4. E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In Alves-Foss [2], pages 353–404.
5. U. Hansmann, M. S. Nicklous, T. Schäck, and F. Seliger. *Smart Card Application Development Using Java.* Springer, 2000.
6. D. Kozen and J. Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. Elsevier, Amsterdam, 1990.
7. T. Nipkow and D. von Oheimb. Machine-checking the Java specification: Proving type safety. In Alves-Foss [2], pages 119–156.
8. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Proceedings, Programming Languages and Systems (ESOP), Amsterdam, The Netherlands*, LNCS 1576, pages 162–176. Springer, 1999.
9. W. Reif. The KIV-approach to software verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, 1995.
10. Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Application Programming Interfaces, Draft 2, Release 1.3*, 1998.

A Type System for Controlling Representation Exposure in Java

P. Müller, A. Poetzsch-Heffter

A Type System for Controlling Representation Exposure in Java

Peter Müller and Arnd Poetzsch-Heffter

Fernuniversität Hagen, D-58084 Hagen, Germany
[Peter.Mueller,Arnd.Poetzsch-Heffter]@Fernuni-Hagen.de

1 Introduction

Sharing mutable objects is typical for object-oriented programs. As a direct consequence of the concept of object identities, it is one of the fundamentals of the OO-programming model. Furthermore, OO-programs gain much of their efficiency through sharing and destructive updates.

However, uncontrolled sharing leads to serious problems: Usually several objects work together to represent larger components such as windows, parsers, dictionaries, etc. Current OO-languages do not prevent references to objects of such components from leaking outside the components' boundaries, a phenomenon called *rep exposure*. Thus, arbitrary objects can use these references to manipulate the internal state of components without using their explicit interface. These manipulations can effect both the abstract value of components and their invariants. This makes OO-programs very hard to reason about. Furthermore, in systems with uncontrolled sharing, basically every object can interact with any other object. Therefore, such systems lack a modular structure and are difficult to maintain.

In this extended abstract, we present a type system for Java and similar languages that enforces a hierarchical partitioning of the object store into so-called *universes* and controls references between universes. The universe type system provides support for preventing rep exposure while retaining a flexible sharing model. It is easy to apply and guarantees an invariant that is strong enough for modular verification. Our type system is related to ownership types ([CPN98]), balloon types ([Alm97]), and islands ([Hog91]). However, it is capable of specifying certain implementation patterns (e.g., binary methods, several objects using a common representation) which cannot be handled by the other approaches.

Overview. Section 2 presents the universe programming model. The universe type system is informally described in Section 3. Section 4 demonstrates the application of universes. Our conclusions are contained in Section 5.

2 Structuring the Object Store

OO-languages in general allow for arbitrary references between objects. The universe type system enables the programmer to structure the object store according to a component-oriented programming model and provides support for sharing-control between components. It is a proper refinement of usual type systems; i.e., the programmer can use the additional power of the type system, but is not forced to do so.

The Universe Programming Model. Systems usually comprise several components. Components consist of one or more objects. Some of these objects are used to interact with other components. Their interfaces form the interface of the component. The other objects are the internal *representation* of the component. A component's representation should be modified only through the component's interface to control modification of the component's abstract value ([MPH00a]) and to guarantee data consistency. Therefore, references to objects of a component's representation must not be passed to other components (*rep exposure*), i.e., references to representation objects must be kept inside the component.

Representations and Universes. We associate every component with a partition of the object store that contains the component's representation, a so-called *universe*. Since a component's representation may contain other components which are in turn associated with a universe, universes form a hierarchical structure. A designated *root universe* corresponds to the whole object store and encloses all other universes. Two universes either enclose each other or are disjoint. The hierarchy of universes introduces a partial order of universes with the root universe as greatest element. We use the term *an object X belongs to universe U* if U is the least universe containing X.

The objects at the interface of a component are not part of the representation (and therefore not contained in the universe). We call them the *owner objects* of the corresponding universe. Owner objects of universe U belong to the universe directly enclosing U.

Consider a component for a doubly linked list of objects with iterators. The list header and the iterators are non-representation objects of the component. They are the owners of the component's universe which contains the nodes of the list.

Sharing Control. An owner object may reference objects belonging to its universe. All other references across universe boundaries are basically prohibited for the following reasons: (a) Objects outside a universe must not reference objects inside. Otherwise, they could use these references to manipulate the internal state of the component.[1] (b) Objects inside a universe must not reference objects outside. If the abstract value of the component depended on the state of objects outside its representation, it could be modified without using the component's interface.

These rules guarantee that objects belonging to universe U can only be referenced by objects belonging to U and U's owner objects. However, the above rules are too strong in two situations: (1) Components might want to pass parts

[1] In this context, local variables and formal parameters behave like instance variables of the `this` object. That is, universes control both static and dynamic aliasing.

of their representations to other components, provided that these components do not use the references for modifications. Such situations occur e.g., when a component needs to store a representation object in a container or when two components have to be tested for structural equality. (2) Objects inside a universe could contain references to objects outside if their abstract values did not depend on the states of the objects outside. To support both situations, we introduce so-called *read-only references*.

Read-only References. Read-only references cannot be used to perform field updates or method invocations on the referenced object[2]. Reading fields via read-only references in turn yields read-only references (or values of primitive types). Abstract values of components must not depend on *states* of objects referenced read-only (but can depend on their identities).

Read-only references can be used to pass references across universe boundaries. A read-only reference to an object belonging to universe U can be turned into a normal reference by objects of U and U's owner objects. For example, object X can pass a reference to object Y as read-only reference to a container. When this reference is retrieved later, X can cast it back to a normal reference and use it for method invocations, etc.

Fig. 1 shows the object structure of a doubly linked list of objects with two iterators. (Objects are depicted by boxes; solid and dashed arrows depict normal and read-only references, resp.; the universe is drawn as ellipse.) The nodes are the representation of the component and therefore inside the universe. Other components can interact with the list header and the iterators, which are the owner objects of the universe. The objects stored in the list are referenced read-only. Section 4 sketches the implementation of the list/iterator example.

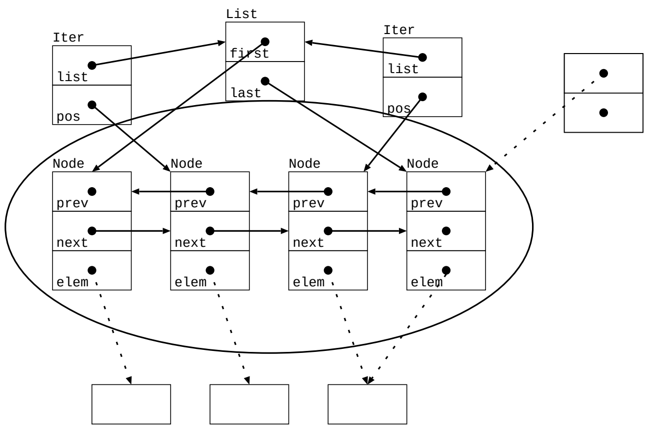


Fig. 1. Object Structure for List/Iterator Example

[2] To keep things simple, we do not consider read-only methods here (i.e., methods without side-effects). For practical applications, they would be helpful.

3 Static Checking of Representation Containment

In the last subsection, we sketched an ideal scenario for alias control for components. However, to check reference containment statically, we have to use a slightly weaker programming model. In this subsection, we present the refined programming model and informally describe a type system to enforce it.

Component Programming Model and Universes. We simplify the component programming model as follows: (1) We associate every object with its own *object universe*. That is, each object X is regarded as the interface of a component with a possibly empty representation. An object is the only owner object of its object universe. (2) We associate every type with a *type universe*. If T is a type declared in module M then every object of a type declared in M is an owner object of T's type universe. Due to inheritance, objects of subtypes of types declared in M may also contain references to objects in T's type universe. However, access control guarantees that subtype methods cannot manipulate objects via such references. Type universes allow objects of types declared in the same module to access a common representation. Thus, components with several owner objects can be realized by implementing them in one module.

The use of type universes reduces the amount of sharing control that can be done. For instance, type universes do not provide support for keeping the nodes of two lists disjoint if the lists' representations are stored in the same type universe. However, objects in T's type universe can only be manipulated by methods implemented in T's module. Therefore, type universes provide sufficient sharing control for modular reasoning, since all "dangerous" code is located in one module (cf. [MPH00a] for a discussion).

The Universe Type System. Reference containment for universes is statically checked by the universe type system. In this subsection, we present the basic ideas of a universe type system. A formalization of the type system and a sketch of the type safety proof can be found in [MPH99,MPH00b].

Universes and Types. There are three kinds of universes: The root universe, type universes, and object universes. Each class C introduces one type for read-only references (*read-only type*) and one type for every universe in a program execution (*reference types*); C is called the *base class* of these types. All types having the same base class share a common implementation, but are regarded as different types.

The subtype relation follows the subclass relation in Java. Two reference types are subtypes if they belong to the same universe and their base classes are subclasses. Two read-only types are subtypes if their base classes are subclasses. Each reference type with base class C is a subtype of the read-only type for C.

Since objects of a class in different universes have different types, objects of one universe cannot be assigned to variables expecting objects of another. All reference types are subtypes of the corresponding read-only type. Therefore, variables of read-only types can hold objects of any universe.

Type Schemes. A class introduces one reference type for each universe (in particular, for each object universe). Thus, the set of types is not fixed at compile time. To enable static type checking, we use so-called *type schemes* to statically type variables, methods, expressions, etc.

Since the universe of a type T is not known at compile time, the implementation of the base class of T can refer to other reference types only relatively to the universe T belongs to. To support the programming model described in Section 2, the universe type system provides three kinds of type schemes for reference types: (1) *Ground type schemes* of the form `C` to refer to the type for class `C` belonging to the same universe as T, (2) *object type schemes* of the form `C<obj>` to refer to the type for class `C` in the object universe owned by `this`, and (3) *class type schemes* (`C<S>`) to refer to the type for class `C` in the type universe associated with the type for class `S` in the universe T belongs to. Furthermore, there are type schemes for read-only types (`C<ro>`), and primitive types.

The subtype relation on type schemes resembles the subtype relation on types. Since read-only type schemes are supertypes of the corresponding reference type schemes, the cast operation can be used to downcast expressions of read-only type schemes to reference type schemes. As for ordinary casts, a dynamic check guarantees that the dynamic type of the right-hand-side object is a subtype of the type of the left-hand-side variable and therefore refers to the same universe.

Informal Type Rules. Three basic rules guarantee type safety of the universe type system (cf. [MPH00b] for a formalization): (1) A type scheme combinator (see appendix) is used to determine the type schemes for fields accesses and method invocations. The resulting type scheme must not be *undefined* to guarantee that an expression does not evaluate to a (non-read-only) reference that points "two steps down" in the universe hierarchy (e.g., by reading an object scheme field on an object scheme variable). (2) To keep object universes on the same level of the universe hierarchy disjoint (except for read-only references), all local variables/formal parameters of object type schemes refer to the object universe of `this`. To check this property statically, fields of object type schemes and methods with object type schemes as result/parameter type schemes may only be accessed/invoked on `this`. (3) Neither writing field access nor method invocation is allowed on read-only references

The Universe Invariant. In every well-typed state, each instance variable and each local variable/formal parameter holds a value of a subtype of the declared type of the variable. Thus, if object X references object Y exactly one of the following cases holds:[3] (1) X and Y belong to the same universe; (2) Y belongs to the object universe owned by X; (3) Y belongs to a type universe owned by X; (4) the reference is read-only.

This invariant guarantees the following *representation containment property*: All access paths from the root universe to a representation object X that do not contain read-only references pass through owners of X's universe.

[3] Again, local variables/formal parameters behave like instance variables of `this`.

4 Example

We illustrate the application of object and type universes, and of read-only types by two implementations of a doubly linked list. Our examples contain two patterns that cannot be handled in other type systems for alias control: Binary methods and cooperating objects that access a common representation.

Doubly Linked Lists. Our list implementation consists of a class `Node` for the node structure and a class `List` for the head of the list. Since the list is supposed to contain objects of any universe, `Node`'s `elem` field is declared read-only. Each node structure exclusively belongs to one list header. Therefore, the nodes are stored in the object universe of the list header (`first` and `last` use the object type scheme). The `equals` method in `List` takes a read-only parameter. Thus, it can access its representation and compare it to the representation of `this`.

```
class Node  { Object<ro> elem; Node prev; Node next; }

class List {
  Node<obj> first; Node<obj> last;
  public List() {
    Node<obj> f = new Node<obj>();    Node<obj> l = new Node<obj>();
    this.first = f;                   this.last = l;
    f.next = l;                       l.prev = f;                  }
  public void appFront(Object<ro> o) { ... }
  public boolean equals(List<ro> l)  {
    Node<obj>  n1 = this.first;       Node<ro>   n2 = l.first;
    Node<obj>  l1 = this.last;        Node<ro>   l2 = l.last;
    Object o1<ro> = n1.elem;          Object<ro> o2 = n2.elem;
    while (n1 != l1 && n2 != l2 && o1==o2) {
      n1 = n1.next;                   n2 = n2.next;
      o1 = n1.elem;                   o2 = n2.elem;  }
    return n1 == l1 && n2 == l2;                                   }
}
```

At first sight, the above example does not require the usage of universes since no `List` method returns a reference to a `Node` object. However, the universe type system guarantees that subclasses of `List` cannot introduce additional methods that violate representation containment. And, what is even more important, it prevents programmers from accidently writing classes that give away references to representation objects.

Lists with Iterators. By a variant of the above example, we demonstrate the use of type universes. The example shows how list iterators can be realized. Iterators allow one to remove elements from the list. Therefore, they must be able to modify the list representation and cannot be implemented via read-only references. To allow lists and iterators to access a common representation, we use type universes instead of object universes to store the node structure of the list. To do that, every `Node<obj>` in the above program has to be replaced by `Node<List>`. The same type scheme is used by the implementation of `Iter`:

```
class Iter {
  List list;  Node<List> position;     public Iter(List l)      { ... }
  public boolean hasNext() { ... }     public Object<ro> next() { ... }
  public void remove()     { ... }                                   }
```

5 Conclusion

We presented a flexible model for object-oriented programming that supports a hierarchical structure of the object store. It is a proper extension of the classical model in which all objects belong to one universe. It supports read-only references to express restricted access to objects. Read-only references increase the flexibility of the programming model and simplify the implementation of methods that need access to two representations. The programming model is realized by a type system that enforces a special representation containment property.

The representation containment property guarantees that modification of a representation is only possible by calling a method on a corresponding owner object. It can be considered as a further step towards "semantic encapsulation", simplifying program verification and optimization. In addition to this, the underlying programming model might be helpful for a better understanding of component-based programming approaches and distributed programming.

References

[Alm97] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In M. Akşit and S. Matsuoka, editors, *ECOOP '97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer-Verlag, 1997.

[CPN98] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, October 1998.

[Hog91] J. Hogg. Islands: Aliasing protection in object-oriented languages. In A. Paepcke, editor, *OOPSLA '91 Conference Proceedings*, pages 271–285, October 1991. SIGPLAN Notices, 26 (11).

[MPH99] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999. Technical Report 263, URL: `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[MPH00a] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[MPH00b] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. *Software Practice and Experience*, 2000. (submitted).

A Appendix

Type Scheme Combinator. To determine the type scheme of method invocations and field accesses, the following table is used, where the type scheme of the target of the method invocation/field access determines the line, and the result/parameter/field type scheme determines the column (all combinations not mentioned in the table yield *undefined*). For instance, the field access expression `v.f` with `v` and `f` having type schemes `D<obj>` and `C`, resp., has type scheme `C<obj>`.

	`C`	`C<obj>`	`C<T>`	`C<ro>`	`boolean`	`int`
`D`	`C`	`C<obj>`	`C<T>`	`C<ro>`	`boolean`	`int`
`D<obj>`	`C<obj>`	*undefined*	*undefined*	`C<ro>`	`boolean`	`int`
`D<S>`	`C<S>`	*undefined*	*undefined*	`C<ro>`	`boolean`	`int`
`D<ro>`	`C<ro>`	`C<ro>`	`C<ro>`	`C<ro>`	`boolean`	`int`

Composite Refactorings for Java Programs

M. Ó Cinnéide, P. Nixon

Composite Refactorings for Java Programs

Mel Ó Cinnéide[1] and Paddy Nixon[2]

[1] Department of Computer Science, University College Dublin, Dublin 4, Ireland.
mel.ocinneide@ucd.ie
http://www.cs.ucd.ie/staff/meloc/default.htm
[2] Department of Computer Science, Trinity College Dublin, Dublin 2, Ireland.
paddy.nixon@cs.ucd.ie

Abstract. There has been much interest in refactoring recently, but little work has been done on tool support for refactoring or on demonstrating that a refactoring does indeed preserve program behaviour. We propose a method for developing composite refactorings for Java programs in such a way that a rigorous demonstration of behaviour preservation is possible.

1 Introduction

A refactoring is a change made to the internal structure of software that improves it in some way but does not alter its observable behaviour [2]. Refactoring has increased in importance as a technique for improving the design of existing code, especially with the advent of methodologies such as Extreme Programming [1] that involve little up-front design and multiple iterations through the software lifecycle. The earliest significant work on refactoring was the suite of C++ refactorings developed by William Opdyke [4]. This work was hampered by the low-level complexities of the C++ language and was never developed into a practical tool. It did however form the basis for the development of the Smalltalk Refactory Browser [6]. Smalltalk is a much cleaner language than C++ and this refactoring tool has been very successful. Its principal limitation is probably that Smalltalk is not a very widely-used language outside of academia. These experiences suggest that the Java programming language may be a promising language to serve as a domain for refactoring development. In spite of the obvious syntactic similarities, it is a much simpler language than C++ and has become extremely popular in the past number of years.

The ultimate aim of our work is the development of a methodology for the construction of program transformations that introduce design patterns to a Java program [3]. The algorithm that describes a design pattern transformation is expressed as a composition of refactorings using sequencing, iteration and conditional constructs. This type of transformation must not change the behaviour of the progam, and so it is necessary to be able to calculate if a given composition of primitive refactorings is itself behaviour preserving. Each primitive refactoring has a precondition and a postcondition. When applied to a program for which the precondition holds, the resulting transformed program exhibits the same behaviour as the original, and the postcondition holds for this program. This paper describes a technique that, given a composition of refactorings, computes whether it is a refactoring, and what its pre- and postconditions are.

2 Preliminaries

Is this section we describe the mathematical notation we use and outline the basic elements of our approach.

2.1 Notation

We use the same notation as Roberts in his refactoring work [5]:

- P: This is the program to be refactored.
- $\mathcal{I}_P$: Denotes an interpretation of first-order predicate logic where the domain of discourse is the program elements of P.
- $\models_{\mathcal{I}_P} pre_R$: Denotes the evaluation of the precondition of the refactoring R on the interpretation $\mathcal{I}_P$.
- $post_R(\mathcal{I}_P)$: Denotes the interpretation $\mathcal{I}_P$, rewritten with the postcondition of the refactoring R.
- $f[(x, y)/true]$: Denotes the function f, extended with the new element (x, y). This syntax is used in postconditions to describe the effect of the refactoring on the analysis functions.

2.2 Analysis Functions

We do not explicitly build an internal representation of the program; rather the required information is extracted when needed. Analysis functions are used to extract this information. They serve a dual role in that they are used both in specifying the preconditions to the refactorings and as an transformation designer's view of the program being transformed. An example of the specification of an analysis function is as follows:

Boolean **contains**(c:*Class*, m:*Method*): Returns true iff the class c contains the method m.

2.3 Helper Functions

In describing a refactoring it may be necessary to extract richer content from the program code than is provided by the analysis functions. Helper functions are used to perform this type of task. As they are not at the primitive level of the analysis functions, we provide them with pre- and postconditions. Helper functions are proper functions without side-effects on the program, so the postcondition invariably involves the return value of the helper function itself. For example the *makeAbstract* helper function is specified as follows:

Method **makeAbstract**(*c*:*Constructor*, *newName*:*String*): Returns a method called *newName* that, given the same arguments, will create the same object as the constructor *c*.

pre: None.
post: createsSameObject′ = createsSameObject[(*c*,*m*)/true] $\wedge$
nameOf′ = nameOf[(*m*,*newName*)/true], where *m* is the returned method.

2.4 Primitive Refactorings

Composite refactorings are built upon a layer of primitive refactorings. Each primitive refactoring is given a precondition written in first-order predicate logic and a postcondition that describes the effect of applying this refactoring in terms of changes to the relevant analysis functions. An argument that behaviour is preserved by this transformation is also provided. This is not formal, but it is at least as strong as the argument a programmer would make internally were they to perform the refactoring by hand. Also, this argument is only made once by the designer of the primitive refactoring and is effectively reused each time a new composite refactoring uses the primitive refactoring. As an example, the *addMethod* refactoring is specified as follows:

addMethod(*c*:*Class*, *m*:*Method*): Adds the method *m* to the class *c*. A method with this signature must not already exist in this class or its superclasses. This refactoring extends the external interface of the class.

pre: isClass(*c*) $\wedge$ ¬defines(*c*, *nameOf(m)*, *sigOf(m)*)
post: contains′ = contains[(*c*,*m*)/true] $\wedge$
∀ a:Class, a≠c, if equalInterface(*a*,*c*) then
equalInterface′ = equalInterface[(*a*,*c*)/false].

behaviour preservation: Since a method with the same name and signature as the method being added is not defined in the class, there can be no name clashes and no existing invocations of this method.

3 Composite Refactorings

In this section we describe the way in which refactorings are composed, and present a technique for deriving the pre- and postconditions of a composite refactoring. The importance of this technique lies in the fact that it allows us to build complex transformations as a composition of primitive refactorings and then to check the legality of the composition and calculate its pre- and postconditions. Here we consider two ways in which refactorings are composed, namely *chaining* and *set iteration*.

Chaining is where a sequence of refactorings are applied one after the other. For example, the following chain adds methods *foo* and *foobar* to the class *c*.

```
addMethod(c,foo)
addMethod(c,foobar)
```

Set iteration is where a refactoring or a refactoring chain is performed on a set of program elements. For example, the following set iteration copies all the methods of the class a to the class b.

```
ForAll m:Method, classOf(m)=a {
      addMethod(b,m)
}
```

A selection statement can also be used. For space reasons we omit it here.

3.1 Computing Pre- and Postconditions for a Chain of Refactorings

A chain of refactorings may be of any length, but we can simplify the computation of its pre- and postconditions by observing that we need only solve the problem for a chain of length 2. This procedure can then be repeatedly applied to the remaining chain until the full pre- and postconditions have been computed.

The two refactorings to be composed are referred to as R_1 and R_2. For a general refactoring R_i, its precondition and postcondition are denoted by pre_{R_i} and $post_{R_i}$ respectively. Figure 1 presents a graphical depiction of this. The

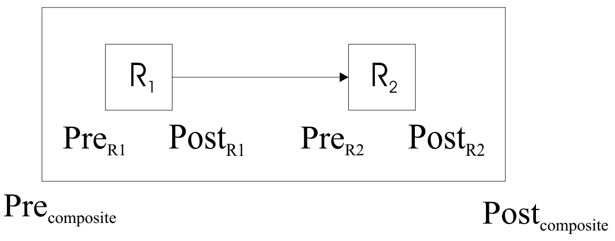


Fig. 1. A Refactoring Chain

precondition of this chain is not simply the conjunction of pre_{R_1} and pre_{R_2}. Firstly, $post_{R_1}$ may guarantee pre_{R_2} which means that an unnecessarily strong precondition would result. Secondly, although the precondition for R_2 may be made part of the precondition for the chain, the refactoring R_1 may break it meaning that this composition of refactorings can never be legal.

The technique we present first attempts to compute the precondition of the chain. During this computation it may emerge that the chain is illegal. Assuming the chain is indeed legal, its postcondition is computed.

1. *Legality test and precondition computation*: First we compute the parts of pre_{R_2} that are not guaranteed by $post_{R_1}$:
   $\models_{post_{R_1}(\mathcal{I}_P)} pre_{R_2}$
   If a contradiction arises in this evaluation, the chain is illegal. The postcondition of the first refactoring creates a condition that contradicts the precondition to the second refactoring.
   The precondition of the complete chain is obtained by evaluating:

$pre_{R_1} \wedge \models_{post_{R_1}(\mathcal{I}_P)} pre_{R_2}$
A contradiction can arise in this evaluation as well, and this also means that the chain is illegal. In this case the precondition to the first refactoring demands a certain condition that contradicts the precondition to the second refactoring, and the first refactoring does not change this condition.
2. *Postcondition computation*: The postcondition is obtained by "sequentially ANDing" the postconditions. By this we mean that if $post_{R_1} \wedge post_{R_2}$ leads to a contradiction, the part of $post_{R_1}$ that causes the problem is dropped. So if $post_{R_1}$ contains the mapping:
$classOf' = classOf[(foo, c)/true]$
and $post_{R_2}$ contains the mapping:
$classOf' = classOf[(foo, c)/false]$
then it is $classOf' = classOf[(foo, c)/false]$ that becomes part of the postcondition of the chain. Denoting this operator as $\wedge_{seq}$ we state the postcondition of the chain to be:
$post_{R_1} \wedge_{seq} post_{R_2}$

3.2 Computing Pre- and Postconditions for a Set Iteration

A set iteration has the following format:

```
ForAll x:someProgElement, somePredicate(x,...) {
    someRefactoring(x, . . . )
}
```

where "..." denotes the program entities that are arguments to the predicate and/or arguments to the refactoring. If the set of x of type *someProgElement* that satisfies *somePredicate*$(x, \ldots)$ is given as $\{x_1, x_2 \ldots, x_n\}$, then this iteration may be viewed as the following chain:

```
someRefactoring($x_1, \ldots$)
someRefactoring($x_2, \ldots$)
...
someRefactoring($x_n, \ldots$)
```

However this is a *set* iteration, so the refactorings can take place in any order. In particular any of them can be first and this fact enables us to define when a set iteration is legal and what its pre- and postconditions should be.

1. *Legality test*: A set iteration is illegal if the precondition of any component refactoring depends on the postcondition of another component refactoring. It is also illegal if the postcondition of any component refactoring contradicts the precondition of another component refactoring.
2. *Precondition computation*: The precondition of the first refactoring of a chain must form part of the precondition for the whole chain, so the precondition of the set iteration must be at least the ANDing of the preconditions of each of the component refactorings. Nothing stronger is required, so the precondition for the above chain can be expressed as:

$$\bigwedge_{i=1}^{i=n} pre_{someRefactoring(x_i,...)}$$

3. *Postcondition computation*: By a similar argument, the postcondition for the above chain can be expressed as:

   $$\bigwedge_{i=1}^{i=n} post_{someRefactoring(x_i,...)}$$

The legality test for a set iteration is not as prescriptive as in the chaining case. It is usually necessary to study the general postcondition carefully to ensure that it has no impact on the precondition on another iteration. It has nevertheless proved to be useful in the cases we have examined.

4 Conclusions

We have described a method for building composite refactorings based on a set of primitive refactorings. Our layer of primitive refactorings are Java-specific, though in principle they could be defined for another language as well. The method of computing the pre- and postconditions of a composite refactoring is similar to Roberts' approach [5], but our work improves on this in several ways. We explicitly calculate whether or not a composite refactoring is legal and we also compute its postcondition as we want to be able to use this composite as a component in future composite refactorings. Roberts also only permits chains of refactorings and does not consider any type of iteration.

We have successfully designed and implemented several composite refactorings that introduce design patterns to a Java program. The techniques described here improved our confidence greatly that the transformations are behaviour preserving. The computation of pre- and postconditions is currently performed by hand by the designer of the design pattern transformation, but our aim is to automate much of this work in the future. We believe that the layer of primitive refactorings we have built coupled with the composition method provides a useful support for further work in the area of refactoring of Java programs.

References

1. Kent Beck. *Extreme Programming.* Addison Wesley Longman, Reading, Massachusetts, first edition, 2000.
2. Martin Fowler. *Refactoring: improving the design of existing code.* Object Technology Series. Addison-Wesley Longman, Reading, Massachusetts, first edition, 1999.
3. Mel Ó Cinnéide and Paddy Nixon. A methodology for the automated introduction of design patterns. In Hongji Yang and Lee White, editors, *Proceedings of the International Conference on Software Maintenence*, pages 463–472, Oxford, September 1999. IEEE Press.
4. William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, 1992.
5. Donald Roberts. *Eliminating Analysis in Refactoring.* PhD dissertation, University of Illinois at Urbana-Champaign, Department of Computer Science, 1999.
6. Donald Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.

Java Access Protection through Typing

E. Rose, K. Høgsbro Rose

Java Access Protection through Typing[⋆]

Eva Rose[1] and Kristoffer Høgsbro Rose[2]

[1] INRIA-Rocquencourt (GIE Dyade); Domaine de Voluceau, Rocquencourt B.P. 105; F–78153 Le Chesnay (France)
[2] ENS-Lyon (LIP); 46, Allée d'Italie; F–69364 Lyon 7 (France)

Abstract. We propose integrating field access in general, and dedicated read-only field access in particular, into the Java type system. The principal gain is that "getter" methods can be eliminated such that

– fast static lookup can be used instead of dynamic dispatch for field access (without requiring a sophisticated inlining analyses),
– the (noticeable) space required by getter methods is avoided,
– denial-of-service attacks on field access is prevented, and
– access protection violations can be discovered by the bytecode verifier thus further simplifying the required run-time support.

We obtain this by extending a formalization of the Java bytecode verifier with access control so we can prove that the change is safe and backwards compatible.

1 Introduction

Object-oriented programming languages in general, and Java in particular, do not distinguish between read- and write-access to fields. Instead the recommended method to only permit read access to a field is to make the field private and write a "getter" method that accesses the field and returns the stored value.

For Java, the semantics of field access states that the actual field location accessed in an object can be determined statically (at compile-time), whereas the actual getter method invocation is determined dynamically (at run-time) [1, §15.10.1]. This has the following consequences:

– Using a getter method is significantly slower (at run-time) than using a direct field access. (The traditional remedy for this is to declare getter methods `final` which permits the compiler to *inline* its body, *i.e.*, insert the field access instruction directly at the invocation place. In Java this is frequently not feasible because Java employs *dynamic class loading* which means that often a class to inline from is not available when installing a class using a getter method.)

⋆ **Extended abstract .**

- It is possible to access the field belonging to a particular (super)class of an object by simply casting the object of the field access to the appropriate class. One cannot obtain a similar effect with a getter method. (One may see this as a feature rather than an inconvenience.)
- "Denial-of-service" attacks are possible in that a getter method can be overriden by a subclass. (This can also be avoided by declaring the method `final`.)
- Finally, getter methods may add a significant space overhead to class files since they must be declared and their code given. For example, getter methods account for about one fourth of the total number of methods in the standard Java "`java.*`" package source classes.[1]

Furthermore the *Java virtual machine* (JVM) specifies that field access control is performed through (dynamic) load and run time checks. This seems a shame since everything else about fields is static.

Here is a traditional example with a getter method: an object that simply contains an integer value that should be publicly readable.

```
class CrCardRd1 {
  int it;
  public int getIt() {return it;}
}
```

Access to the `it` field value of an object `cc` of type `CrCardRd1` requires the method invocation `cc.getIt()` with the problems discussed above.

In this paper we propose a simple modification in two steps that eliminates the problem altogether:

1. add a special *get-specific* access modifier that permits making the reading of a field "more public" than the modification of it, and
2. integrate field access checks into the type system.

In effect we propose replacing the above code with

```
class CrCardRd2 {
  read public int it;
}
```

which explicitly permits everyone to read off the field value with the usual field access syntax `cc.it` (but not to assign to it).

[1] This measure obtained for Sun's JDK 1.1 [4] with the unix commands "`find jdk1.1 -name '*.java' -exec grep ' +public .*(' '{}' ';' | wc -l`" to get the total number of public methods (4317), and "`find jdk1.1 -name '*.java' -exec egrep ' +public .* get.*(' '{}' ';' | wc -l`" to get the number of getter methods (999).

Accessibility from / Modifier	`private`	`package`	`protected`	`public`
same class	✓	✓	✓	✓
other class, same package	×	✓	✓	✓
subclass outside package	×	×	✓	✓
other class outside package	×	×	×	✓

Table 1. Java Access Modifiers.

Plan. In section 2 we propose a mininimal extension of the Java language [1] with the desired semantics, and since the Java runtime environment is centered around the JVM [2], we exlain how the modification could be specified for the JVM. In section 3 we then explain how we can integrate field access into the type system of the JVM to be performed by the JVM *bytecode verifier*. Finally, we conclude in section 4 with some remarks on future work.

2 Read-only Field Access in Java

Recall that the Java access "modifiers" change the access rights as shown in table 1. (The "package" modifier is the default assumed when no modifier keyword is present and the table has ✓ when access is permitted and × when it is not.) Notice that the permissions are strictly included in each other and, in fact, statically checkable, since both the class hosting the field and the method attempting to access it are statically known.

We propose to extend the Java language with syntax for specifying an access modifier specific to *reading* a field value. This can be done with the following syntax extension to the Java Language Specification [1, §8.3.1]:

FieldDeclaration:
  FieldModifiers$_{op}$ *ReadModifier*$_{op}$ *Type* *VariableDeclarators* `;`
  ...
ReadModifier:
  `read` *AccessModifier*$_{op}$

The semantics of the new construction is that we must separate field *access* from field *assignment*: an access that is not an assignment, *i.e.*, is not a Java *LeftHandSide* [1, §15.25], is permitted if either of the (original) field access modifier or the specific read access modifier (if any) permits it.

By using "either" we ensure that our extension is conservative in that old systems ignoring the read modifier will always be strictly less permissive than new ones.

This change permeates through to the JVM [2] where it could be realized directly by extending the `access_flags` item of the `field_info` structure and modifying the getfield semantics correspondingly, however, we will prefer to integrate it into the type system as described in the next section.

3 Field Access Types

At present field access rights are checked for the getfield instruction both at Java verification time, for the declared "static" type [2, §4.8.2, p.138], and again at run-time, using the real "dynamic" type [2, §6.4, p.248].

Our idea is the following: if access information is integrated into the type system then

1. the bytecode verifier can check for access violations assuming that the provided access information (in the type) is correct, and
2. resolution requires equality of the used and actual type.

Thus once resolution has happened the system has checked that no access violation can happen.

Formalizing this is based on the ordering

$$\texttt{private} < \text{package} < \texttt{protected} < \texttt{public}$$

(where larger access rights define more accessible fields).

Assume a field is declared like

```
class c₁ {
  w read r t f;
}
```

with c_1 the class hosting the field, w the "write" access modifiers, $\mathtt{r}$ the "read" access modifiers, $\mathtt{t}$ the field (value) type, and $\mathtt{f}$ the field name. Consider an access in a class c from within some method with code like

```
  c₂ x;
  ...  x.f ...
```

(with c_2 a subclass of c_1). When we include the access modifiers in the type information this means that the field access generates the JVM instruction

$$\text{getfield}(c_1,\ w\ \texttt{read}\ \mathtt{r}\ \mathtt{t}\ ,\mathtt{f})$$

where we note that the bytecode (as usual) contains

- the class where the field is declared: $\mathtt{c}_1$,
- a copy of the (complete) type of the field extracted from the original definition: w `read` r t, and
- the field name: f.

We will express the static check for access rights of the above situation by extending the JVM type checking (verification) rules with a judgment like

$$\mathrm{c} \vdash \mathsf{getfield}(\mathrm{c}_1,\ w\ \mathtt{read}\ \mathrm{r}\ \mathrm{t}\ ,\mathrm{f})$$

defined by

$$\frac{}{\mathrm{c} \vdash \mathsf{getfield}(\mathrm{c}_1,\ w\ \mathtt{read}\ \mathtt{public}\ \mathrm{t}\ ,\mathrm{f})}$$

$$\frac{\mathrm{r} \geq \mathtt{protected} \quad \mathrm{c} \leq: \mathrm{c}_1}{\mathrm{c} \vdash \mathsf{getfield}(\mathrm{c}_1,\ w\ \mathtt{read}\ \mathrm{r}\ \mathrm{t}\ ,\mathrm{f})}$$

$$\frac{\mathrm{r} \geq \text{package} \quad \text{same-package}(\mathrm{c},\mathrm{c}_1)}{\mathrm{c} \vdash \mathsf{getfield}(\mathrm{c}_1,\ w\ \mathtt{read}\ \mathrm{r}\ \mathrm{t}\ ,\mathrm{f})}$$

$$\frac{}{\mathrm{c} \vdash \mathsf{getfield}(\mathrm{c},\ w\ \mathtt{read}\ \mathtt{private}\ \mathrm{t}\ ,\mathrm{f})}$$

Notice that it is the fact that the checks in table 1 are static that makes this possible since the verifier merely needs to be able to determine whether two classes belong to the same package and whether the current class is a subclass of the class owning the field; both can be checked with information readily available. There are similar rules for putfield checking the w component of the type, of course.

All that remains is to encode the access modifiers into the JVM *FieldDescriptor* encoding [2, §4.3.2]: the existing type equality test at resolution time will ensure that the verifier has not made false assumptions. (The encoding is not difficult but outside the scope of this paper.)

In the full paper we integrate the above rules into our formalization of a Java bytecode verifier [3], and show that “well-access-typed” programs cannot violate field permissions.

4 Conclusion

We have outlined how access rights to fields, and specifically *read-only* access rights, can be encoded in the Java type system as implemented by a (slightly modified) Java bytecode verifier, thus eliminating all access right checks at run-time.

One very interesting further venue of research is that using “access types” could be used to implement “sticky” access rights such as “private objects” where the *value* cannot be passed out of the current method, for example.

One may comment that “static is bad because everything should be run-time configurable.” This possibility remains (using setter and getter methods) but we believe it is important to give the programmer of a class the choice of permitting (efficient) build-in static field access even for read-only fields, specifically for the variants of Java targeted at devices with limited resources [5].

Another question that one could ask is “why not for ‘setter’ methods?” This can be done but is complicated by the fact that “setter” methods usually also check the value to be stored for validity to ensure that the object is (internally) consistent. One could introduce special “validity checks” such that our getter example could be extended, for example, with a

```
class CrCardRd raises IllegalCreCaIt {
  protected read public int it
    {if (it<0) raise IllegalCreCaIt;}
}
```

with the semantics that any assignment to `it` would execute the additional “assertion” code. Such an addition may be worth considering, however, in contrast to the read-only case it complicates the Java language considerably.

Finally we remark that the above can without problems be integrated with *lightweight* bytecode verification [3], as used by Sun’s KVM [6], to permit static access control even in sparse resources.

References

1. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.
2. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, second edition, 1999.
3. Eva Rose and Kristoffer Høgsbro Rose. Lightweight java bytecode verification. Technical report, INRIA, 2000. To appear.
4. Sun. *JDK 1.1 Documentation*, 1997. Available from http://java.sun.com/products/jdk/1.1/docs/.
5. Sun. Java 2 platform, micro edition. `http://java.sun.com/j2me`, 1999.
6. Sun. The K virtual machine (KVM). `http://java.sun.com/products/kvm`, 1999.

Escape Analysis for Stack Allocation in Java

E. Cho, K. Yi

Escape Analysis for Stack Allocation in Java

Eun-Sun Cho[1] and Kwangkeun Yi[2]

[1] eschough@ropas.kaist.ac.kr
Dept. of Information & Computer Engineering
Ajou University
San 5 Wonchon-dong Paldal-gu
Suwon Kyunggi-do, Korea

[2] kwang@ropas.kaist.ac.kr
Research on Program Analysis System (ROPAS)
Dept. of Electrical Engineering & Computer Science
Korean Advanced Institute of Science & Technology
373-1 Kusong-dong Yusong-gu
Taejon, Korea

Abstract. We suggest an escaping analysis for Java programs, to identify stack-allocatable objects. This analysis considers an object escaping if it is used after the deactivation of the method that created it. For each object, this analysis records the interprocedural movement of the method of its creation. An object is considered escaping the method that created it, if the method of its creation has been already deactivated. Our approach is different from prior works, in that special cares of non-local variables need not be taken. This enables us to handle some cases that are missed in the previous approach. The whole analysis is done in a single phase.

1 Introduction

Garbage collecting objects in Java makes memory management easier for the programmer, but it is time consuming – stack allocation may be an alternative [1, 3, 5]. This article presents a static analysis to identify the objects that never escape the method of their creation.

The analysis is based on the fact that an object escapes the method that created it, if and only if it is used after the method is deactivated[4]. To identify such cases, we record for each object its interprocedural behaviors of the method that created it. At each expression using the object, the analysis checks if its record contains the sign for the deactivation of the method of its creation. If so, the object is deemed escaping the method; otherwise, it can be allocated in the

⋆ This work is done while the first author was associated in ROPAS (http://ropas.kaist.ac.kr), KAIST in 1999 – 2000.

⋆⋆ This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology, and by Brain Korea 21 project.

method stack. This approach is quite different from conventional analyses, which consider objects escaping as long as they are assigned to non-local variables, irrespective of whether they are actually used or not.

The analysis is done in a single-phase, based on abstract interpretation[2]. We consider a core of Java for presentation simplicity. The abstract syntax is shown in Figure 1. We assume one formal parameter for each method, one field variable for each object and no arrays. We do not consider loops, which can be simulated by recursions. Neither do we consider local variables since they make no important effects on our analysis. And '`this`' cannot be abbreviated in the expressions for field access and method invocations upon it.

Except the case that `finalize` methods destroy the objects in sequence, no ordering on the destruction of objects is assumed, such as the one forced by run-time garbage collection.

$$\begin{array}{rcl}
P & ::= & C^{*}e \\
C & ::= & \texttt{class}\ c\ \texttt{ext}\ c\ \{\texttt{var}\ x^{*}\ M^{*}\} \\
M & ::= & \texttt{method}\ m(x){=}e \\
 & & \\
c & & \textit{class name} \\
m & & \textit{method name} \\
x & & \textit{variable name}
\end{array}
\qquad
\begin{array}{rcl}
e & ::= & x \\
 & | & \textit{head}.x \\
 & | & x := e \\
 & | & \textit{head}.x := e \\
 & | & \texttt{new}\ c \\
 & | & e\ ;\ e \\
 & | & \texttt{this} \\
 & | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & | & e.m(e) \\
\textit{head} & ::= & \textit{head}.x \\
 & | & \texttt{this} \\
 & | & \texttt{x}
\end{array}$$

Fig. 1. Syntax of the language

2 Analysis

To safely estimate the information of interprocedural behavior, we build an abstract interpreter. Iterative evaluation of the interpretation function reaches to a fixed point that is our analysis result.

At the creation of each object, we start recording the interprocedural movements of the method that created it by encoding the movements into seven types of abstract entities: $\hat{\mathcal{P}} = \{\bot, \epsilon, u, uu^{+}, dd^{+}, ud, ud^{+}\}$. For the object `o`, let the method that created it be `m`; ϵ indicates that no activation/deactivation of `m` has happened; u ($u$pward movement of the stack pointer) means that an instance of `m` is deactivated (returned); uu^{+} and dd^{+} respectively means that multiple instances of `m` has been deactivated and activated; ud represents one call after one return from `m`; ud^{+} stands for one or more activation after one or more deactivation of `m`. The matched pairs of activation and deactivation of the same instance of `m` is ignored, which is useless in determining if it is "outside of `m` or not". The four interprocedural movements u, uu^+, ud, and ud^+ of `m` for object `o` imply that that the object *may-escape* `m`. These seven abstract symbols cover all the cases of interprocedural movement of `m` after the creation of `o`.

This record of the interprocedural behavior is based on the concept of *procedure string* in Harrison's work[4]. He proposed an escaping analyses for functional languages, to handle implicitly created objects such as cons cells and closures. Procedure string is the sequence of α^u's and α^d's to record the interprocedural behavior of running programs, where α is a function (method) name. An abstract value of a procedure string is a mapping from the function (method) names to subsets of the set $\{\epsilon, u, uu^+, d, dd^+, ud, ud^+\}$.

Note that we do not use d, which would indicate a single call of method `m` that created the object `o`. It would happen only if another instance of `m` is invoked before the existing instance of `m`, which created `o`, is deactivated. This means that the program part responsible for d must include recursive call of `m`, which would always result in dd^+. In addition, we have a new type ud, representing a single pair of deactivation and invocation. This enables us to handle the repeated but non-recursive invocations of `m`, which were abstracted in the same way as recursive invocations of `m` in [4].

An abstract memory object is identified by the `new` expression together with the record of interprocedural movement of the method of its creation. The former one ensures that for each `new` expression, at least one abstract object is identified. The latter one is useful when a `new` expression creates more than one object, as in loops or in recursive functions. $2^{O\hat{R}ef}$ in heap, environment and escaping is the set of ideals ordered by set inclusion.

The interpretation function $\hat{\mathcal{E}}$ takes a pre-state and returns a post-state, where a pre-state is an element of $E\hat{n}v \times He\hat{a}p \times Esc\hat{a}pe$, and a post-state is an element of $2^{O\hat{R}ef} \times E\hat{n}v \times He\hat{a}p \times Esc\hat{a}pe$. The first element of a post-state represents a set of abstract objects, the possible result of a given expression. Each pre-state and post-state describes the current heap, the current environment and escaping information. A heap maps each object to its possible field values, which are again objects. *head.x* refers the heap; *head.x*:=e and $\texttt{new}_\beta\ c$ change the heaps. An environment consists of the pairs of sets of objects for the variable `this` and those for the parameter x. It is accessed by the expressions x and `this`, and modified by x:=e. Modifying `this` is impossible.

Domains

labels for new expressions	$\beta \in B$
interprocedural behavior	$\hat{p} \in \hat{\mathcal{P}} = \{\epsilon, u, dd^+, uu^+, ud, ud^+\} \cup \bot$
object reference	$\hat{r} \in O\hat{R}ef = B_\bot \times \hat{\mathcal{P}}_\bot$
heap	$\hat{h} \in He\hat{a}p = O\hat{R}ef \rightarrow 2^{O\hat{R}ef}$
environment	$\hat{\sigma} \in E\hat{n}v = 2^{O\hat{R}ef} \times 2^{O\hat{R}ef}$
escaping	$\hat{e} \in Esc\hat{a}pe = B_\bot \rightarrow \{\top, \bot\}$

$$
\begin{array}{l}
\hat{\mathcal{E}} \in \ Expr \to (\widehat{Env} \times \widehat{Heap} \times \widehat{Escape}) \to (2^{\widehat{ORef}} \times \widehat{Env} \times \widehat{Heap} \times \widehat{Escape}) \\
\hat{\mathcal{F}} \in (Expr \to (\widehat{Env} \times \widehat{Heap} \times \widehat{Escape}) \to (2^{\widehat{ORef}} \times \widehat{Env} \times \widehat{Heap} \times \widehat{Escape})) \to \\
\qquad\qquad (Expr \to (\widehat{Env} \times \widehat{Heap} \times \widehat{Escape}) \to (2^{\widehat{ORef}} \times \widehat{Env} \times \widehat{Heap} \times \widehat{Escape}))
\end{array}
$$

$$
\begin{array}{lll}
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ x\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & = & \langle\ snd(\hat{\sigma}), \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ \mathtt{this}\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & = & \langle\ \sigma(\mathtt{this}), \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ head.x\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & = & let\ \langle\ \hat{rs}, \hat{\sigma}, \hat{h}, \hat{e_1}\ \rangle = \hat{\mathcal{E}}\ [\![\ head\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & & in\ \ \langle\ ReadH\ \hat{h}\ \hat{rs}, \hat{\sigma}, changeE\ \hat{e_1}\ \hat{rs}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ x{:=}e\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & = & let\ \langle\ r\hat{s}_1, \hat{\sigma_1}, \hat{h_1}, \hat{e_1}\ \rangle = \hat{\mathcal{E}}\ [\![\ e\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & & in\ \ \langle\ r\hat{s}_1, \langle\ fst(\hat{\sigma_1}), snd(\hat{\sigma_2}) \sqcup r\hat{s}_1\ \rangle, \hat{h_1}, \hat{e_1}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ head.x{:=}e\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & = & let\ \langle\ r\hat{s}_1, \hat{\sigma}, \hat{h}, \hat{e_1}\ \rangle = \hat{\mathcal{E}}\ [\![\ head\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & & \qquad \langle\ r\hat{s}_2, \hat{\sigma_2}, \hat{h_2}, \hat{e_2}\ \rangle = \hat{\mathcal{E}}\ [\![\ e\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e_1}\ \rangle \\
 & & in\ \ \langle\ r\hat{s}_2, \hat{\sigma_2}, WriteH\ \hat{h_2}\ r\hat{s}_1\ r\hat{s}_2, \\
 & & \qquad\qquad\qquad changeE\ \hat{e_2}\ r\hat{s}_1\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ \mathtt{new}_\beta\ c\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & = & let\ \hat{r} = \langle\ \beta, \epsilon\ \rangle\ \ \hat{rs} = \{\hat{r}\} \\
 & & in\ \ \langle\ \hat{rs}, \sigma, WriteH\ \hat{h}\ \hat{rs}\ \{InitField(c)\}, \hat{e}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ e_1.m(e_2)\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & = & let\ \langle\ r\hat{s}_1, \hat{\sigma_1}, \hat{h_1}, \hat{e_1}\ \rangle = \hat{\mathcal{E}}\ [\![\ e_1\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & & \qquad \langle\ r\hat{s}_2, \hat{\sigma_2}, \hat{h_2}, \hat{e_2}\ \rangle = \hat{\mathcal{E}}\ [\![\ e_2\ ]\!]\langle\ \hat{\sigma_1}, \hat{h_1}, \hat{e_1}\ \rangle \\
 & & in\ \ \left\{ \begin{array}{ll}
\langle\ r\hat{s}_1, \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & if\ r\hat{s}_1 = \bot \\
\bigsqcup_{pairwise}\ \{\ \langle\ \widehat{UpRs}\ r\hat{s}_3\ \alpha, \sigma_2, \widehat{UpH}\ \hat{h_3}\ \alpha, \hat{e_3}\ \rangle\ | & \\
\qquad \langle\ r\hat{s}_3, \hat{\sigma_3}, \hat{h_3}, \hat{e_3}\ \rangle = \hat{\mathcal{E}}\ [\![\ e\ ]\!] & \\
\qquad\quad \langle\ \widehat{DownEnv}\ \langle\ \{\hat{r}\}, r\hat{s}_2\ \rangle\ \alpha, \widehat{DownH}\ \hat{h_2}\ \alpha, & \\
\qquad\quad changeE\ (changeE\ \hat{e_2}\ r\hat{s}_2)\{\hat{r}\}\ \rangle & \\
\qquad \wedge\ \lambda_\alpha x.e = \widehat{Method}(snd(\hat{r}), m) \wedge\ \hat{r} \in r\hat{s}_1\ \} & \\
 & otherwise
\end{array} \right. \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ e_1;\ e_2\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & = & let\qquad \langle\ r\hat{s}_1, \hat{s_1}, \hat{h_1}, \hat{e_1}\ \rangle = \hat{\mathcal{E}}\ [\![\ e_1\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & & in\ \ \hat{\mathcal{E}}\ [\![\ e_2\ ]\!]\langle\ \hat{\sigma_1}, \hat{h_1}, \hat{e_1}\ \rangle \\
\hat{\mathcal{F}}\hat{\mathcal{E}}\ [\![\ \mathtt{if}\ e_1\ e_2\ e_3\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle & = & let\qquad \langle\ r\hat{s}_1, \hat{\sigma_1}, \hat{h_1}, \hat{e_1}\ \rangle = \hat{\mathcal{E}}\ [\![\ e_1\ ]\!]\langle\ \hat{\sigma}, \hat{h}, \hat{e}\ \rangle \\
 & & in\ \ \hat{\mathcal{E}}\ [\![\ e_2\ ]\!]\langle\ \hat{\sigma_1}, \hat{h_1}, changeE\ \hat{e_1}\ r\hat{s}_1\ \rangle \\
 & & \qquad \bigsqcup \hat{\mathcal{E}}\ [\![\ e_3\ ]\!]\langle\ \hat{\sigma_1}, \hat{h_1}, changeE\ \hat{e_1}\ r\hat{s}_1\ \rangle
\end{array}
$$

Fig. 2. Definition of abstract evaluation function

The method is invoked with a new environment that consists of the receiver and the parameter as its `this` and `x`; the previous environment is restored after the method is returned. The evaluation of method invocation updates the record of the interprocedural behavior for the objects related. This assigns new identifiers to the objects, since the record is a part of the object identifier. To hide the complexity of this operation, auxiliary functions prefixed with *Up* and *Down* are introduced. They also adjust the heap and the environment to the new object identifiers. The definitions of auxiliary functions are in the appendix.

The result of the analysis is collected in $\hat{e}$, a mapping from each `new` expression to $\{\top, \bot\}$, where $\top$ means *may-escaping* and $\bot$ means *never-escaping*. At the beginning of the analysis, the $\bot$ value is assumed for every `new` expression. The value is updated to $\top$ whenever one of the objects created by the `new` expression is possibly used. Here, "using" an object means that the object is accessed. Thus, *changeE*, updating operation for $\hat{e}$, is applied whenever the objects in heap are referenced: reading/writing the objects, passing the objects to the methods as actual parameters are passed, and so on. Note that, during the abstract evaluation the escaping value is only increasing from $\bot$ to $\top$, which makes the iterative application of the abstract monotonic interpreter (collecting the escaping information) reach a fixed point within a finite time.

The correctness of our analysis is shown by the following theorem. $fix\ \mathcal{F}[\![\ expr\ ]\!]$ is a concrete semantics of the language. We abuse the terms 'pre-state' and 'post-state' in the theorem for the ones before the abstraction.

Theorem 1. *For any expression expr,*

$$Abs_{post\text{-}state} \circ fix\ \mathcal{F}[\![\ expr\ ]\!] \sqsubseteq fix\hat{\mathcal{F}}[\![\ expr\ ]\!] \circ Abs_{pre\text{-}state}$$

3 Related Works

Unlike our analysis, previous escaping analyses for Java[1, 3, 5] consider an object escaping the method that created it, as soon as it is assigned to global variables, parameters, and/or return variables. Thus, they miss the objects that are assigned to such variables, but not actually used outside the methods of their creation. As far as we know, this analysis, considering actual using points of objects, is the only one to cover such cases.

Blanchet[1] proposed a type-based and two-phase (a backward phase and a forward phase) escaping analysis. Like our work, he build an abstract interpretation and defined more than one abstract objects for each `new` expression.

4 Conclusions

This paper introduces a single phase escape analysis based on abstract interpretation to determine if an object is stack-allocatable. By recording the interprocedural movement of the method of creation for each object, this analysis considers an object stack allocatable as long as it is not actually used after the method of its creation is deactivated. We are now concentrating on the cases with the unknown methods, and implementing the analysis to obtain experimental result.

Appendix : Auxiliary functions

$$changeE = \lambda\hat{e}.\lambda\hat{rs}\begin{cases}\bot & if\ \hat{e} = \bot \\ \lambda\beta.\left\{\begin{array}{ll}\hat{e}(\beta) \sqcup \top & if\ \hat{p} \in \{u, uu^+, ud, ud^+\} \\ & \quad for\ some\ \langle\ \beta, \hat{p}\ \rangle \in \hat{rs} \\ \hat{e}(\beta) & otherwise\end{array}\right\} & otherwise\end{cases}$$

$$ReadH = \lambda\hat{h}.\lambda\hat{rs}.\bigsqcup_{\hat{r}\in\hat{rs}} \hat{h}(\hat{r}),\ WriteH = \lambda\hat{h}.\lambda r\hat{s}_1.\lambda r\hat{s}_2.\lambda\hat{r}.\begin{cases}\bot & if\ \hat{r} = \bot \\ \hat{h}(\hat{r}) & if\ \hat{r} \notin r\hat{s}_1 \\ \hat{h}(\hat{r}) \sqcup r\hat{s}_2 & otherwise\end{cases}$$

	ϵ	u	d^+	uu^+	ud	ud^+	$\bot$
$\oplus d$	$\{d\}$	$\{ud\}$	$\{d^+\}$	$\{ud^+\}$	$\{ud^+\}$	$\{ud^+\}$	ϕ
$\oplus u$	$\{u\}$	$\{uu^+\}$	$\{\epsilon, d^+\}$	$\{uu^+\}$	$\{u\}$	$\{u, uu^+, ud^+\}$	ϕ
$\ominus d$	ϕ	ϕ	$\{\epsilon\}$	ϕ	$\{u\}$	$\{u, uu^+, ud^+\}$	ϕ
$\ominus u$	ϕ	$\{\epsilon\}$	ϕ	$\{u, uu^+\}$	ϕ	ϕ	ϕ

$$Do\hat{w}nR = \lambda\langle\ \beta, \hat{p}\ \rangle.\lambda\alpha.\begin{cases}\bot = \phi & if\ \langle\ \beta, \hat{p}\ \rangle = \bot \\ \{\langle\ \beta, \hat{p'}\ \rangle | \hat{p'} \in \hat{p} \oplus d\} & if\ birthmethod(\beta) = \alpha \\ \{\langle\ \beta, \hat{p}\ \rangle\} & \end{cases}$$

$$DownRs = \lambda\hat{rs}.\lambda\alpha.\begin{cases}\phi & if\ \hat{rs} = \phi \\ \{DownR\ \hat{r}\ \alpha | \hat{r} \in \hat{rs}\} & otherwise\end{cases}$$

$$Do\hat{w}nH = \lambda\hat{h}.\lambda\alpha.\lambda\langle\ \beta, \hat{p}\ \rangle.\begin{cases}\bot & if\ \hat{h} = \bot = \phi \\ \sqcup\{Do\hat{w}nR\ \hat{h}\langle\ \beta, \hat{p'}\ \rangle\ \alpha | \hat{p'} \in \hat{p} \ominus d\} & if\ birthmethod(\beta) = \alpha \\ Do\hat{w}nR\ \hat{h}\ \langle\ \beta, \hat{p}\ \rangle\ \alpha & otherwise\end{cases}$$

$$Dow\hat{n}Env = \lambda\hat{\sigma}.\lambda\alpha.\begin{cases}\langle\ \phi, \phi\ \rangle & if\ \hat{\sigma} = \bot \\ \langle\ \bigsqcup\{DownR\ \hat{r}\ \alpha\ |\ \hat{r} \in fst(\hat{\sigma})\}, & \\ \qquad \bigsqcup\{DownR\ \hat{r}\ \alpha\ |\ \hat{r} \in fst(\hat{\sigma})\}\ \rangle & otherwise\end{cases}$$

References

1. B. Blanchet. “Escape analysis for object oriented languages application to java”. In *Proc. of the ACM OOPSLA Conf.*, 1999.
2. P. Cousot and N. Cousot. “Abstract interpretation : a unified lattice model for static analysis of programs by construction of approximation”. In *Proc. of SIGPLAN Conf. on Principle of Programming Languages*, 1977.
3. J-D. Choi et. al. “Escape Analysis for Java”. In *Proc. of the ACM OOPSLA Conf.*, 1999.
4. W. L. Harrison. “The Interprocedural Analysis and Automatic Parallelization of scheme Programs”. *Lisp and Symbolic Computation*, 2(3):179–396, 1989.
5. J. Whaley and M. Rinard. “Compositional Pointer and Escape Analysis for Java Programs”. In *Proc. of the ACM OOPSLA Conf.*, 1999.

Overloading and Inheritance in Java

D. Ancona, E. Zucca, S. Drossopoulou

Overloading and Inheritance in Java
(Extended Abstract)

Davide Ancona[1], Elena Zucca[1], and Sophia Drossopoulou[2]

[1] DISI, University of Genova
[2] Department of Computing, Imperial College
{zucca,davide}@disi.unige.it   sd@doc.ic.ac.uk

Abstract. The combination of overloading and inheritance in Java introduces questions about function selection, and makes some function calls ambiguous. We believe that the approach taken by Java designers is counterintuitive. We explore an alternative, and argue that it is more intuitive and agrees with the Java rules for the cases where Java considers the function calls unambiguous, but gives meaning to more calls than Java does.

1 Overloading and Inheritance

Overloading, already present in the seventies (LIS, Ada, Hope), allows the definition of several, different, functions with the same name and different parameter types. Thus, the programmer is freed from the burden of dreaming different identifiers for functions which perform essentially the same operation, but on different types of parameters. Overloading is usually resolved statically[1], namely, the function that fits the actual parameter types is selected. Thus, overloading resolution corresponds to consistent renaming of the function definitions and the corresponding function calls.

Inheritance, already present in the sixties (Simula), allows classes to be organized in a class hierarchy, and either to inherit functions from their superclass or to redefine these functions. When a function is called for a certain object, the function from the most specific superclass is called. Resolution for inheritance can only take place at run-time, and depends on the dynamic class of the receiver. Inheritance introduces subtyping, namely an object of a subclass may be used where an object of a superclass is expected.

The combination of subtyping and overloading is not straight-forward, since now more than one method may fit the types of the arguments of a function call.

Java overloading resolution Assume that Oyster is a subclass of Food, and that aPhl, pascal, aFood and anOyster are variables of type Phil, FrPhil, Food and Oyster respectively. Consider the following example:

[1] Dynamic resolution is also possible, as it actually happens in object-oriented languages for the first argument and for all arguments in languages with *multimethods*; see [1] for a deep analysis of the difference. In this paper, we only consider overloading with static resolution.

```
class Phil extends Object {
   int eat ( Food x ) { return 1; }
   int eat ( Oyster x ) { return 2 ;}
}

class FrPhil extends Phil {
   int eat ( Food x ) { return 3; }
}
```

For the method call aPhl.eat (anOyster), two methods are applicable, both declared in the class Phil: one with parameter type Food (returning 1) and one with parameter type Oyster (returning 2).

In cases where several methods are applicable, Java (and C++ before) took the approach of selecting the method that "fits" best, called *most specific.*[2] In the call from above, it is clear that the method with parameter Oyster fits better than the method with parameter Phil, therefore this method is selected. In general, if many methods declared in the same class are applicable, then that with most specific argument's type is selected, if any, otherwise[3] the call is ambiguous.

Much less obvious is the case when we have to compare methods declared in different classes, like in the method call pascal.eat (anOyster). Here, both the method returning 2 and that returning 3 are applicable, and in Java's view *none* fits better than the other, hence the method call is ambiguous.

In our experience, many people (even with a deep knowledge of Java) are unaware of the implication of the Java overloading resolution in this case, and expect on the contrary that the method returning 2 is selected. In our opinion this is so, because the latter solution corresponds to an intuitive understanding of inheritance semantics. This is explained in the next section.

Alternative overloading resolution In our view the method call pascal.eat (anOyster) should *not* be ambiguous, and should return 2. This view is based both on methodological and language semantics reasons.

On the methodological side, an implication of the Java rule is that programmers who use a class and want to be aware of how method overloading will be resolved need to know not only which methods are inherited, but also the exact class containing the definition of these inherited methods. This conflicts with a modular approach to software development where all the information needed for the correct use of a module (class in this case) should be provided by its specification alone.

For instance, the specification of class FrPhil states that this class has two methods int eat (Food x) and int eat (Oyster x). However, with this information only, users *cannot* know which will be the effect of the call pascal.eat (anOyster).

[2] Another solution would be to avoid the occurrence of such situations where several methods are applicable, and forbid the definition of overloaded methods with parameter types with non-empty intersections of sets of values. This would make the class Phil from above illegal. However, such a solution would restrict overloading only to non-class, non-interface parameter types, since the value **null** belongs to all classes.

[3] For instance, if there are two applicable methods with two arguments with types Food,Oyster and Oyster,Food, respectively.

The counterpart at the level of language semantics is that inheritance should be explainable as a mechanism for code sharing. In other words, a natural intuitive understanding of inheritance is as a linguistic mechanism allowing to get for free the same effect that one could obtain "by hand" by duplicating parent's code in the heir. Thus, the subclass FrPhil should be equivalent to a copy of Phil, where the overriden methods of Phil are replaced by the corresponding methods from FrPhil. That is, FrPhil should be equivalent to FrPhil_by_Copy, defined as:

```
class FrPhil_by_Copy extends Phil {
    int eat ( Oyster x ) { return 2; }
    int eat ( Phil x ) { return 3; }
}
```

Then, for a variable rousseau of class FrPhil_by_Copy, the call rousseau eat (anOyster), would be unambiguous, and would return 2. Therefore, by analogy, the call pascal.eat (anOyster) should return 2 as well!

2 The alternative approach "subsumes" the Java approach

The alternative approach corresponds to the Java approach for all cases where Java considers the method call unambiguous. As we have seen, in some cases where Java considers the call ambiguous the alternative gives it an unambiguous meaning.

The rest of the paper is devoted to the proof of this claim.

For simplicity, and without restricting the applicability of our result, we assume that all methods have one parameter.

Moreover, we start from considering only non-abstract classes. The generalization to interfaces and abstract classes requires more involved definitions and will be considered in the full paper [2].

Both the Java approach and the alternative approach start from the same set of *applicable* methods, which are all the methods of the receiver's class, either directly declared or inherited (that is, declared in a superclass and not overidden, *cf.* [3] 15.11.1) , which are type compatible with the given method call, *cf.* [3] 15.11.2.1), but they differ in the way they consider methods to "fit better" than others.

Sets of applicable methods are denoted by $\mathcal{A}$, $\mathcal{A}'$ *etc.*, and contain method *types*. Method types consist of the class containing the method declaration, the argument type and the result type.

For example, the applicable methods for the call aPhl.eat (anOyster) are $\mathcal{A}_1 = \{< \mathsf{Phil}, \mathsf{Food}, \mathsf{int} >, < \mathsf{Phil}, \mathsf{Oyster}, \mathsf{int} >\}$.
Also, the applicable methods for the method call pascal.eat (anOyster), are $\mathcal{A}_2 = \{< \mathsf{Phil}, \mathsf{Oyster}, \mathsf{int} >, < \mathsf{FrPhil}, \mathsf{Food}, \mathsf{int} >\}$.

In Java, a method "fits better" than another one if the former is defined in a subclass of where the latter is defined and the argument types of the first widen[4] to the corresponding argument types of the second, *cf.* [3] 15.11.2.2.

[4] A type t widens to another type t′ if they are identical, or t is a subclass of t′, or t is a subinterface of t′, or t is a subclass of a class that implements a subinterface of t′, *cf.* [3] 5.1.4 .

On the other hand, for the alternative definition, a method "fits better" than another one if the argument types of the former widen to the corresponding argument types of the latter.

So, we define the following two ordering relationships on method types:

- $< \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3 > \ \ll_{\mathsf{j}} \ < \mathsf{t}'_1, \mathsf{t}'_2, \mathsf{t}'_3 >$ iff t_1 subclass of t'_1 and t_2 widens to t'_2
- $< \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3 > \ \ll_{\mathsf{a}} \ < \mathsf{t}'_1, \mathsf{t}'_2, \mathsf{t}'_3 >$ iff t_2 widens to t'_2

Notice that, by definition of applicable methods, there can be at most *one* applicable method with a given name and argument type, hence the ordering relationship $\ll_{\mathsf{a}}$ can be equivalently defined as follows:

$< \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3 > \ \ll_{\mathsf{a}} \ < \mathsf{t}'_1, \mathsf{t}'_2, \mathsf{t}'_3 >$ iff t_2 widens to t'_2, $\mathsf{t}_2 \neq \mathsf{t}'_2$, or
$\mathsf{t}_2 = \mathsf{t}'_2$ and t_1 subclass of t'_1

The equivalence follows from the fact that if $\mathsf{t}_2 = \mathsf{t}'_2$ then also t_1 subclass of t'_1 for the reason explained above. This formulation point out that the two ordering relationships correspond to two different ways of combining the ordering relationships existing on the first and second component of method types, that is, componentwise and lexicographical from right to left.

For example, $< \mathsf{Phil}, \mathsf{Oyster}, \mathsf{int} > \ \ll_{\mathsf{j}} \ < \mathsf{Phil}, \mathsf{Food}, \mathsf{int} >$, and similarly, in the alternative approach, $< \mathsf{Phil}, \mathsf{Oyster}, \mathsf{int} > \ \ll_{\mathsf{a}} \ < \mathsf{Phil}, \mathsf{Food}, \mathsf{int} >$. This makes the call aPhl.eat (anOyster) unambiguous in both approaches.

On the other hand, $< \mathsf{Phil}, \mathsf{Oyster}, \mathsf{int} > \ \ll_{\mathsf{a}} \ < \mathsf{FrPhil}, \mathsf{Food}, \mathsf{int} >$, but the method types are incomparable in the sense of $\ll_{\mathsf{j}}$. So, pascal.eat (anOyster) is unambiguous in the alternative approach and ambiguous in Java.

It is easy to see that both $\ll_{\mathsf{j}}$ and $\ll_{\mathsf{a}}$ are reflexive and transitive. The relation $\ll_{\mathsf{j}}$ is antisymmetric if the program is well-formed (*i.e.,* if the subclass relationship is acyclic), whereas the relation $\ll_{\mathsf{a}}$ is *not* antisymmetric: a counterexample would be a method defined in class c_1 and defined with the same parameter type in whichever different class c_2. Also, one can immediately see that $\ll_{\mathsf{j}}$ is stronger than $\ll_{\mathsf{a}}$, *i.e.,* that:

$$< \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3 > \ll_{\mathsf{j}} < \mathsf{t}'_1, \mathsf{t}'_2, \mathsf{t}'_3 > \ \Rightarrow \ < \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3 > \ll_{\mathsf{a}} < \mathsf{t}'_1, \mathsf{t}'_2, \mathsf{t}'_3 >$$

Finally, the type of a method call with applicable methods $\mathcal{A}$ is defined as the return type of the least method type from $\mathcal{A}$ in the sense of the ordering either $\ll_{\mathsf{j}}$ (in Java) or $\ll_{\mathsf{a}}$ (in the alternative). If this minimum does not exist, then the method call is ambiguous.

It only remains to be shown that if a set $\mathcal{A}$ of applicable methods has a least element in the sense of $\ll_{\mathsf{j}}$ then this is also the least element in the sense of $\ll_{\mathsf{a}}$. This is easy, because $\ll_{\mathsf{j}}$ implies $\ll_{\mathsf{a}}$.

This completes the proof that the alternative approach is a conservative extension of the Java approach, in the sense that it gives the same meaning to all the method calls which are unambiguous for Java.

Another alternative As stated above, the two ordering relationships correspond to two different ways of combining the ordering relationships existing on the first and second component of method types. It is natural therefore to also consider a third possibility, which corresponds to lexicographical order from left to right, that is:

$$< t_1, t_2, t_3 > \ll_{a2} < t'_1, t'_2, t'_3 > \quad \text{iff} \quad t_1 \text{ subclass of } t'_1, \quad \text{or} \quad t_1 = t'_1 \text{ and } t_2 \text{ widens to } t'_2$$

Such a rule resolves overloading by selecting the method that fits *first*. It searches from the most specific subclass following the superclass hierarchy upwards, and only takes those overloaded methods into account which were declared in the first superclass that contains applicable methods. With this rule the method call pascal.eat (anOyster) would select the method returning 3. However, this does not influence of course the Java rule for overriding; so, for instance, in the call aPhil.eat(anOyster) the method declared in Phil with argument type Oyster is selected (and kept at run time) even in the case aPhl has dynamic type FrPhil.

It is easy to see that this alternative too is conservative (and less restrictive) w.r.t. Java rule; indeed, also $\ll_{a2}$ is implied by $\ll_j$. We will further investigate the methodological implications of this third possibility.

3 Outline of the full paper

We have argued that the Java approach to overloading resolution considers ambiguous some method calls which would have a meaning if we had taken a more intuitive view of inheritance, based on copying. We have given an alternative rule for overloading resolution which gives meaning to more calls than Java does and gives the same meaning as Java when Java considers a method call unambiguous, and have proven this result.

We briefly illustrate now which are the further topics which will be developed in the full paper [2].

Methodological aspects We will include a survey of the design space of the semantics of method overloading in the presence of subtyping. In particular we will discuss more extensively advantages and disadvantages of the different possible choices and analyze what happens in other object-oriented languages, notably in C++.

Extension to abstract classes and interfaces The main difference w.r.t what has been previously presented is that when considering also abstract classes and interfaces more than one method with the same signature an be inherited, as explicitly stated in [3] 8.4.6.4.

Hence, now in applicable methods there can be two methods with the same name and argument type.

For instance in the following example

```
interface I1 {
    void m ( ) ;
}

interface I2 {
    void m ( ) ;
}
```

```
interface I extends I1, I2 {
}
```

the applicable methods for a call i.m() with i of type I are

$$\{< \mathsf{I1}, \mathsf{void}, \mathsf{void} >, < \mathsf{I2}, \mathsf{void}, \mathsf{void} >\}.$$

Hence this call should be ambiguous following the Java rule in [3] (even though different Java compilers have different, and sometimes obscure, behaviour on this and similar examples), while the alternative approach corresponds to assume that the interface I has just *one* method void m(), regardless of how many copies have been inherited, hence the call is not ambiguous.

We will show that it is possible to generalize the previous formalization of the two rules and that the result that $\ll_{\mathsf{a}}$ is a conservative extension of $\ll_{\mathsf{j}}$ still holds.

An interesting remark is that the above situation shows that there is a contradiction in [3] between the *definition* of overloading given in 8.4.7 and the rule for overloading resolution, and that this contradiction disappears if one considers the alternative rule.

Copy semantics of inheritance We have repeatedly stated that the alternative approach corresponds to an intuitive interpretation of inheritance as a mechanism achieving the same effect one would have by copying parent's code in the heir. In some more detail, that means that a simple way for expressing inheritance semantics is to translate a Java program in an intermediate representation (which we call Flat Java) consisting, roughly speaking, of a subtype-hierarchy part and a collection of "flat" classes (that is, without any extends or implements clause). Methods of one of these classes are all the methods (either directly declared or inherited) of the original Java class.

In Flat Java there is no longer notion of inheritance, hence in method calls there is no need for method look-up. Of course, the information about the subtyping relation is still needed, *but only* for type-checking bodies of methods, while no information about that is needed at run time.

This model corresponds to a natural intuitive understanding of the basic inheritance mechanism in object-oriented languages; some more sophisticated features of Java, like the super mechanism and the possibility of hiding fields, do not have a direct copy semantics but can be easily simulated.

In [2] we will provide a formal definition of copy semantics by means of a translation from Java into Flat Java and show that, for what concerns overloading, copy semantics leads to the alternative approach we proposed, while the Java rule is not directly expressible.

References

1. G. Castagna. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science. Birkhäuser, 1997.
2. D. Ancona, E. Zucca, and S. Drossopoulou. Overloading and inheritance in Java. Technical Report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 2000. In preparation.
3. J. Gosling, B. Joy, and G. Steele. *The Java$^{\text{TM}}$ Language Specification.* Addison-Wesley, 1996.

Testing Java

S. Eisenbach, T. Valkevych

Testing Java

Position Paper

Susan Eisenbach and Tatyana Valkevych
Department of Computing,
Imperial College of Science, Technology and Medicine
email: {se,tanya}@doc.ic.ac.uk

The Java programming language is evolving. Sun releases new versions of their compilers, sometimes clarifying and amending features of the language [9]. They withdrew the language from the standardization procedure *"to ensure compatibility and continued rapid pace of innovation"* encouraging the world-wide community *"to compete on implementation"* [11]. Their latest language description *"The Java Language Specification"* (JLS) is currently only a draft document [10].

What this means to programmers, compiler writers and formalizers, is that on features where the JLS is ambiguous or out-of-date one has to test a feature on the latest Sun compiler to see what the expected behaviour should be. So the de-facto specification of Java is obtained by running Sun's latest compiler. Therefore, we have been developing the Java Test Suite for online, automated, classified experimenting over Java features as specified in [7].

The Java Test Suite is a web based tool that enables the user to submit test programs, to specify the expected results for compilation and execution, to select individual or groups of test programs for features testing, to select compilers, to run the tests on one or more compilers, and to compare the results from two points of view: (1) actual and expected behaviour, and (2) programs' behaviour under various compilers. The test programs and test engine can also be downloaded for running in the user's own environment.

When working on the operational semantics of Java [4] occasionally we are told that our semantics are incorrect [2]. Usually we use the test suite to check the behaviour of different compilers exercising the specific feature in question. Normally what has happened is the language has changed and the semantics has not kept up. So, old versions of the compiler produce results compatible with our semantics and the latest compiler produces something different.

Java is a language that loads its classes at runtime. To our knowledge other test suites for other languages run in batch mode. This would not enable the testing of behaviours that come from dynamically loading classes. The Java Test Suite can deal with dynamic behaviours. Classes can be loaded separately, some changed and then reloaded. In most languages, one can recreate a binary of a program by compiling all the sources together, linking and loading. This is not necessarily true in Java [5, 6, 3]. It is possible to create an executable over time where no such sources can exist. This feature can also be tested.

Having a web based application leads to all sorts of potential security problems. The Test Suite enables users to run programs on our machines. It therefore could not let people put test programs in directly. Rather they go into a submission database, and an administrator checks them and moves them into the actual database of test programs. This is time consuming and we welcome alternative solutions to the security problem. We wonder whether this precaution is actually necessary.

There exist two versions of the suite: the first implementation was as CGI scripts, Perl, SQL [1], the second one as Java servlets [8].

The suite is populated with a wide range of test programs to give a broad coverage of all Java features. Programs need to be concise and accurate and should test only one feature. When fully populated we expect it should contain thousands of programs covering various aspects of Java features. Therefore, we wish to disseminate the Java Test Suite widely to help populate it quickly. We are especially interested in Java programs that show behaviour that is surprising.

References

[1] Eileen Dreyer. A Java Test Suite. Master's thesis, Imperial College of Science, Technology and Medicine, 1999.

[2] Sophia Drossopoulou. Private communication, March 2000.

[3] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A Fragment Calculus – Towards a Model of Separate Compilation, Linking and Binary Compatibility. In *LICS'99 Proceedings*, 1999. 147-156.

[4] Sophia Drossopoulou, Tatyana Valkevych, and Susan Eisenbach. Java Type Soundness Revisited. Technical report, Imperial College, April 2000. Also available from: http://www-doc.ic.ac.uk/project/slurp/.

[5] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java Binary Compatibility? In *OOPSLA'98 Proceedings*, pages 341–358, 1998.

[6] Susan Eisenbach and Chris Sadler. Ephemeral Java Source Code. In *IEEE Workshop on Future Trends in Distributed Computing Systems*, December 1999.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996.

[8] Julian Hatchwell, Melissa Rafael, Silva Lau, Tarik Baig, and Don Vethanayagam. Java Test Suite, 2000. MSc group project, Imperial College of Science, Technology and Medicine.

[9] Clarification and Amedments to *Java Language Specification*. http://java.sun.com/docs/books/jls/clarify.html.

[10] Java Language Specification, 2nd Edition, Draft. http://java.sun.com/aboutJava/communityprocess/maintenance/JLS/index.html.

[11] Sun Microsystems Withdraws Java^{TM} 2 Platform Submission from ECMA. http://java.sun.com/pr/1999/12/pr991207-08.html.