ECOOP;2001 Workshop 4

3rd ECOOP Workshop on Formal Techniques for Java Programs

Informal proceedings, Budapest, 18th June 2001

Program Committee:

Sophia Drossopoulou (Imperial College, Great Britain)
Susan Eisenbach (Imperial College, Great Britain)
Gary T. Leavens, Iowa State University, USA
Peter Müller, Fernuniversität Hagen, Germany
Arnd Poetzsch-Heffter, Fernuniversität Hagen, Germany
Gilad Bracha (Sun Microsystems, USA)
Doug Lea (State University of New York at Oswego, USA)
Rustan Leino (Compaq Computer Corporation, USA)
Erik Poll (University of Nijmegen, The Netherlands)

Organizers:

Susan Eisenbach (Imperial College, Great Britain)
Gary Leavens (Iowa State University, USA)
Peter Müller (FernUniversität Hagen, Germany)
Arnd Poetzsch-Heffter (FernUniversität Hagen, Germany)
Erik Poll (University of Nijmegen, The Netherlands)

3rd ECOOP Workshop on
Formal Techniques for Java Programs

Budapest, 18th June 2001

Programme

9:00 – 10:00 Opening Session, and Invited Talk

Gilad Bracha Adventures in Computational Theology:
Selected Experiences with the Java(tm) Programming Language

10:15 – 11:15

Alessandro Coglio
Improving the Official Specification of Java Bytecode Verification

Kees Huizing and Ruurd Kuiper:
Reinforcing Fragile Base Classes

11:25 –12:25

Davide Ancona, Giovanni Lagorio, and Elena Zucca
Java Separate Type Checking is not Safe

Mirko Viroli
From FGJ to Java according to LM translator

Mats Skoglund and Tobias Wrigstad
A mode system for read-only references in Java

12:25 – 13:45 Lunch

13:45 – 15:45

Pierre Boury and Nabil Elkhadi
Static Analysis of Java Cryptographic Applets

Peter Mueller, Arnd Poetzsch-Heffter, Gary T. Leavens :
Modular Specification of Frame Properties in JML

Gilles Barthe, Dilian Gurov, and Marieke Huisman
Compositional specification and verification of control flow based security properties of multi-application programs

J Strother Moore, Robert Krug, Hanbing Liu, and George Porter
Formal Models of Java at the JVM Level: A Survey from the ACL2 Perspective

16:00 – 17:30

Peter Mueller and Arnd Poetzsch-Heffter
A Type System for Checking Applet Isolation in Java Card

John Boyland
The Interdependence of Effects and Uniqueness

Ana Cavalcanti and David Naumann
Class Refinement for Sequential Java

Joachim van den Berg, Cees-Bart Breunesse, Bart Jacobs, Erik Poll
On the Role of Invariants in Reasoning about Object-Oriented Languages

17:45 – 18:30 Short presentations and closing session

Claus Pahl:
Formalising Dynamic Composition and Evolution in Java Systems

M. Carbone, M. Coccia, G. Ferrari and S. Maffeis
Process Algebra-Guided Design of Java Mobile Network Applications

Peep Kungas, Vahur Kotkas, and Enn Tyugu
Introducing Meta-Interfaces into Java

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 15th European Conference on Object-Oriented Programming, ECOOP 2001. Papers in the proceedings are included here based on the reviews of the workshop organizers.

The papers are also available from:

http://www.informatik.fernuni-hagen.de/import/pi5/workshops/ecoop2001_papers.html

As in the two previous years, we also plan to organize a special issue of an appropriate journal with long versions of selected papers from the workshop and additional invited papers on the topic.

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 , Gary T. Leavens , Peter Müller
Arnd Poetzsch-Heffter, Gilad Bracha, Doug Lea, Rustan Leino, Erik Poll

Invited Talk

Adventures in Computational Theology: Selected Experiences with the Java(tm) Programming Language

Gilad Bracha (SUN Microsystems)

Improving the Official Specification of Java Bytecode Verification

Alessandro Coglio

Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304
http://www.kestrel.edu

coglio@kestrel.edu

Abstract

Bytecode verification is the main mechanism to enforce type safety in the Java Virtual Machine. Since Java security is based on type safety, inadequacies and ambiguities in the official specification of bytecode verification can lead to incorrect implementations where security can be broken. This paper analyzes the specification and proposes concrete improvements. The goal of this work is to increase the understanding, assurance, and usability of the Java platform.

1 Introduction

Bytecode verification is the main mechanism to enforce type safety in the Java Virtual Machine (JVM)[1]. Its purpose is to establish that certain type safety properties are always satisfied when bytecode is run. Therefore, the interpreter or just-in-time compiler can omit checks of such properties, resulting in better performance.

Because Java security is based on type safety [Gon99], correct implementation of bytecode verification is of paramount importance to the security of an implementation of the JVM. Holes in bytecode verification constitute potential points of attack exploitable by malicious programs (e.g., applets from remote sites of the Internet).

The official specification of the JVM [LY99], which includes a description of bytecode verification, is written in informal English prose. While this specification is mostly rather clear, it does contain some inadequacies and ambiguities. Their presence is particularly problematic for security-critical features such as bytecode verification, because erroneous interpretation can lead to erroneous implementation.

This paper exposes and analyzes the inadequacies and ambiguities in the official specification of bytecode verification in Section 2. It then proposes concrete improvements in Section 3. Concluding remarks are given in Section 4. For brevity, the official JVM specification is denoted by "JS". Individual chapters or (sub)sections of it are denoted by appending their number, e.g., "JS3.1". Some knowledge of JS is assumed.

[1] The other mechanisms, which complement bytecode verification, are resolution, class loading, as well as residual run-time checks (e.g., of array indices).

The goal of this paper, like others in the field, is to improve the understanding, assurance, and usability of the Java platform.

2 Analysis

JS4 describes the format of valid class files. Class verification is the process of checking that a byte sequence constitutes a valid class file. Class verification is described in JS4.9.1 as consisting of four passes. Passes 1 and 2 are in charge of checking the format of the class file, excluding those byte subsequences that constitute methods' code. The boundaries of these subsequences are identified during passes 1 and 2, but it is pass 3's responsibility to verify that each of them encodes valid code for an individual method. Pass 4 consists of the checks performed by resolution, i.e., that symbolic references to classes, fields, and methods are correct; even though it may not take place before execution, pass 4 is considered logically part of class verification.

Bytecode verification is pass 3. So, it is a part of class verification. In fact, it is the most interesting and delicate part; the rest of class verification is relatively straightforward and does not present major difficulties.

Bytecode verification is described in JS4.8 and JS4.9. JS4.8 presents constraints that a byte sequence must satisfy in order to represent valid method code. They are divided into *static* and *structural* constraints. JS4.9 explains, in a quite algorithmic way, how such constraints are checked. These descriptions are now analyzed.

2.1 Static Constraints

JS4.8.1 presents static constraints, which a byte sequence must satisfy in order to represent a sequence of correct bytecode instructions with correct opcodes and operands. All the instructions' operands are covered by the list of constraints in JS4.8.1, and the requirements are quite straightforward.

There is one requirement, though, that seems out of place. It states that the `new` instructions cannot be used to create an instance of an interface or of an abstract class.

This is true, but it is given in the context of restrictions on `new`'s operand. Since `new` references a class by name, the class should be resolved in order to determine whether it is an interface, an abstract class, or a non-abstract class. So, unlike all the other constraints given in JS4.8.1, this one cannot be checked on the operand alone.

Further evidence that the requirement is out of place is gathered from the following two observations. The first is that the specification of the `new` instruction in JS6 states that an exception is thrown if the resolved class is abstract or if it is an interface. This means that this check is done at run time.

The second observation is that other instructions that reference classes, fields, or methods in their operands also require checks similar to this one. For example, the `getfield` instruction requires the resolved field to not be static. This requirement is given as a run-time check in the specification of `getfield` in JS6, not as a static constraint in JS4.8.1. The same holds for the other instructions.

2.2 Structural Constraints

Static constraints constitute the straightforward part of bytecode verification. On the other hand, structural constraints, presented in JS4.8.2, constitute the interesting part, whose goal is to verify that certain type safety properties are satisfied when the code is executed. Structural constraints apply to instruction sequences. So, they assume that static constraints are satisfied. Remarks about structural constraints follow.

2.2.1 Terminology

The terms "static" and "structural", applied to the constraints given in JS4.8.1 and JS4.8.2, respectively, sound a little confusing and misleading. Both kinds of constraints are meant to be checked *statically*, without executing the code—in this case, prior to executing the code. So, using the adjective "static" only for the first kind of constraints does not seem ideal.

With a stretch, one could interpret structural constraints as requirements that must be satisfied at run time, even though they are checked statically, at verification time. However, static constraints must be satisfied at run time as well. This point about structural constraints referring to run time versus verification time is revisited in Section 2.2.2.

2.2.2 Undecidability

Many of the structural constraints express facts that must hold when certain instructions are executed. For example, when `getfield` is executed, the top of the operand stack must contain a reference to an object whose class is (a subclass of) the one specified by the instruction's operand.

However, establishing this kind of properties, taken literally, is undecidable. As it is well-known in program analysis, only conservative approximations can be algorithmically computed. Certainly, these structural constraints are meant to be checked only approximately. This supports the view that structural constraints, as stated, apply to run time and not to verification time, as mentioned in Section 2.2.1.

But structural constraints are given in the context of the format of valid class files, i.e., in JS4. So, it would seem more appropriate to provide constraints that can be computed, and that can be used as a public definition of valid class files. This is discussed in more detail in Section 3.2.2.

2.2.3 Redundancy

The first constraint of the list states that each instruction must be executed with an appropriate number and type of values in the operand stack and local variables. The notion of "appropriate" can be determined by the specification of the instruction given in JS6. For example, `getfield` requires the top of the operand stack to contain a reference to an instance of the class specified by the instruction's operand.

It turns out that the same requirement on `getfield` is stated explicitly by a separate structural constraint in the list. However, it is clearly redundant, because it can be derived from the first one.

Another constraint in the list states that no instruction must pop more values from the operand stack than it contains (i.e., no stack underflow must occur). Again,

this is a simple consequence of the first constraint: each instruction requires the presence of a certain number of values in the operand stack, and only those values are popped.

As a matter of fact, it could be argued that even the first constraint of the list is redundant. The specification of instructions given in JS6 includes the types of the values that each instruction expects to find in the operand stack and local variables. As explained in JS6.1, it is the task of bytecode verification to ensure that these expectations are met. So, the first constraint of the list, as stated, does not really say anything new.

2.2.4 Lack of Explanation

The motivation for several of the structural constraints is type safety. However, some constraints in the list do not appear directly related to type safety, and no explicit motivation is given for them in JS4.8.2.

An example is the constraint stating that no uninitialized object can be present in the operand stack or local variables when a backward branch is taken. It also states that no uninitialized object can be present in a local variable in code protected by an exception handler.

Other examples are the constraints related to subroutines: subroutines cannot be called recursively; the instruction following a `jsr` may be returned to only by a single `ret`; each return address can be returned to at most once; etc.

Yet another example is the constraint requiring that, if an instruction can be executed along different execution paths, the operand stack must have the same size prior to execution of the instruction.

Actually, *some* explanation for these constraints can be derived from JS4.9. This is discussed in Section 2.3.

2.2.5 Contradiction

There are two structural constraints that contradict each other. The first says that fields of an uninitialized object cannot be accessed; the object must be initialized first. But the second says that the code of a constructor can store values into some fields of the object that is being initialized—which, strictly speaking, is still uninitialized.

2.2.6 Possibly Out of Place

A structural constraint states that the `invokespecial` instruction must reference an instance initialization method, a method in the current class, or a method in a superclass of the current class. Checking if the method belongs to a superclass of the current class requires resolving the method and the class. Along the same line of reasoning for the constraint on `new` discussed in Section 2.1, it could be argued that this property must be checked at run time, not at verification time.

A structural constraint requires each method return instruction (such as `ireturn` and `freturn`) to match the method's return type. This is a simple check that does not involve type analysis[2]. So, this could well be a static constraint among those in

[2]Of course, type analysis is required to check that the operand stack always has a value of the right type when the instruction is executed. However, this is implied by the first structural constraint in the list, namely that each instruction must be executed with an appropriate number

JS4.8.1.

Another structural constraint is that execution never falls off the bottom of the instruction sequence. This is also rather simple to check and does not involve type analysis. Basically, the last instruction of the sequence must be one that cannot transfer control to the non-existent following one, directly or indirectly[3]. Some static constraints in JS4.8.1 ensure that the operand of each control transfer instruction (e.g., branches) points to an instruction in the code, and not outside it or in the middle of an instruction. For these reasons, the requirement that execution cannot fall off the end of code would fit well as a static constraint among those in JS4.8.1.

2.2.7 Heterogeneity

A final remark is that the structural constraints listed in JS4.8.2 are somewhat "heterogeneous". Most of them state properties that must hold when certain instructions are executed. Others state, instead, requirements relating all possible executions: for example, that the instruction following a `jsr` can be returned to by a single `ret`.

Heterogeneity is not necessarily bad. However, it certainly contrasts with the homogeneity of static constraints, which are all about instruction opcodes and operands[4].

2.3 Verification Algorithm

JS4.9 sketches an algorithm for bytecode verification. The first part of JS4.9.2 essentially explains how to check static constraints and is quite straightforward. A mildly interesting point is that the explanation includes checking that execution cannot fall off the end of code. This supports the view of this requirement as a static constraint, as argued in Section 2.2.6.

The second part of JS4.9.2 explains how to check a decidable approximation of structural constraints, by means of a data flow analysis [NNH98]. This description is followed, in JS4.9.3 through JS4.9.6, by some clarifications concerning the treatment of certain instructions. Remarks about these descriptions follow.

2.3.1 Merging of Operand Stack Types

Type information for the operand stack consists of a sequence of types, modeling the size and contents of the operand stack. The algorithmic description of the data flow analysis in JS4.9.2 prescribes that type information for the operand stack can be merged only when sizes are the same.

Though not the only possibility [Cog01b], this is certainly a sensible strategy that simplifies the analysis. In any case, it provides some explanation for one of the structural constraints mentioned in Section 2.2.4, namely that the operand stack must have the same size along all execution paths. So, that structural constraint is in some sense a "forward reference" to the data flow analysis algorithm. This somewhat challenges the view that structural constraints are just undecidable requirements that must hold at run time, as opposed to describing a computable approximation.

---

and type of values.

[3] The "indirect" case applies to `jsr`: control can be transfered to the following instruction when returning from the called subroutine.

[4] With the exception of the requirement on `new` discussed in Section 2.1.

This means that undecidable requirements and decidable approximations are mixed in JS4.8.2.

2.3.2 Reference Types

The algorithmic description of the data flow analysis talks about "reference types" that are assigned to operand stack positions and local variables, propagated through control flow paths, merged, etc. However, it fails to make it explicit whether these reference types are just names or names plus class loaders[5].

Given that class loaders are runtime objects, the only way for the algorithm to know the identities of the loaders would be to actually load the classes. This is in contrast with the general principle, stated in several points of JS, that classes may be loaded lazily, if and when they are required for execution.

Therefore, the most reasonable interpretation of the "reference types" used in the algorithm is that they are only names. However, it turns out that this lack of clarity is the origin of some type safety bugs [CG01, TH99].

2.3.3 Interface Types

The first edition of JS prescribes that the result of merging two reference types is their first common superclass or superinterface. This works fine for classes, but the first common superinterface of two interfaces may not be unique, because of multiple inheritance.

Therefore, in the second edition of JS the statement has been changed to just say that the result of merging two reference types is their first common superclass. Since `java.lang.Object` is considered a superclass of every interface, the result of merging two interfaces is always `java.lang.Object`.

But this requires a special treatment of `java.lang.Object` when it is the target of an `invokeinterface` instruction. Since `java.lang.Object` may derive from merging two interface types, bytecode verification should allow `invokeinterface` to operate on it. Otherwise, bytecode produced by Java compilers would be rejected.

Normally, `invokeinterface` is only allowed to operate on a class that implements the interface referenced by `invokeinterface` or on a subinterface of it—and `java.lang.Object` is none of them. In order to maintain type safety, a run-time check is necessary when an `invokeinterface` instruction is executed.

Two observations apply to this treatment of interface types. The first is that it is not particularly clean. The second is that JS does not make clear the implications of this approach, e.g., that run-time checks are necessary for `invokeinterface`.

2.3.4 Object Initialization

Bytecode verification must ensure that objects are initialized before they are used. In order to do that, the data flow analysis algorithm must use a special type for the reference to a newly created, uninitialized object. This special type is changed to a regular class type after the constructor is invoked. Since several copies of the object reference can be stored in the operand stack or local variables before invoking the constructor, all the occurrences of the special type must be changed.

[5] As stated in JS5.3, a class or interface in the JVM is identified by its (fully qualified) name plus its defining loader.

After an object of a class is created and before the object is initialized, another object of the same class could be created. The algorithm must distinguish between types for references to the first object and types for references to the second object. The solution prescribed in JS4.9.4 is to use the index of the `new` instruction (i.e., its position in the code) as part of the special type for uninitialized objects. In this way, since different objects are created by different instructions, they have different types assigned to them.

A potential source of trouble is a `new` instruction being part of a loop. If it is possible to go through the loop without initializing the object, how can the algorithm distinguish between objects created during two different iterations of the loop? To avoid this problem, JS4.9.4 prescribes that the algorithm must make sure that no type for an uninitialized object exists in the operand stack or local variables, whenever a backward branch is taken. For analogous reasons, it prescribes that no type for an uninitialized object must exists in a local variable in instructions protected by an exception handler[6].

This provides some explanation for the structural constraints mentioned in Section 2.2.4, about absence of uninitialized objects when backward branches are taken and in code protected by exception handlers. The discussion in Section 2.3.1 applies to them: they are forward references to the algorithm and they somewhat mix undecidable constraints with decidable approximations.

But the main point is that these restrictions on (types for) uninitialized objects are completely unnecessary. This is described in detail and formally proved in [Cog01c].

Intuitively, the reason is the following. Consider, for example, how the data flow analysis computes type assignments for the instructions of a loop. Starting from the first instruction of the method, a type for an uninitialized object can be only introduced by a `new` instruction. After it, the type can be copied around. If a backward branch is taken to an instruction preceding the object creation instruction, the types assigned to the branch instruction are merged with those at the target of the branch. Since the latter do not include the same type for the uninitialized object, such a type disappears as a result of merging. So, when the `new` instruction is reached again, there is no copy of the type for the uninitialized object introduced during the first iteration, and hence no confusion can arise.

2.3.5 Subroutines

Subroutines constitute the trickiest aspect of bytecode verification. The reason is that, in order to perform accurate type inference, the flow of control determined by subroutines must be taken into account. However, subroutines may not be textually delimited and may be exited implicitly by branching or throwing exceptions (i.e., not by a `ret`).

Here, "accurate type inference" means type inference that is both correct (i.e., does not allow type safety to be broken) and not excessively approximate. On one hand, simple-minded treatments of subroutines would ensure correctness but would reject programs produced by compilers. On the other hand, more elaborate treatments would accept all compiled programs, but would raise the risk that type

[6] No restrictions are given on the operand stack because the stack is emptied when an exception is thrown—the thrown exception is then pushed onto the stack.

safety could be broken, if the properties and implications of such treatments are not fully understood.

A detailed discussion of these issues can be found in [Cog01a], which also includes a thorough analysis of the treatment of subroutines described in JS4.9.6 and JS4.8.2, along with its motivations. Replicating this analysis here would make this paper inappropriately long: the reader is referred to [Cog01a].

In a nutshell, the main points are the following. In JS4.9.6 an approach to verify bytecode with subroutines is sketched. The structural constraints in JS4.8.2 that refer to subroutines can be considered to provide further information about the approach.

But the overall description omits important details, such as how to merge lists of `jsr` targets (which are associated to instructions, as prescribed in JS4.9.6) from converging control flow paths.

Some aspects of the approach, such as keeping track of modified variables by means of bit vectors, are explicitly motivated in JS4.9.6. Other aspects of the approach, such as prohibiting recursive subroutine calls, are not explicitly motivated. It turns out that the reason behind some of these unexplained restrictions is to guarantee type safety. However, others turn out to be unnecessary to this purpose.

In addition, the approach ends up rejecting bytecode that can be produced by compilers. For example, in Java 2 SDK 1.2 for Solaris bytecode verification rejects code produced by the compiler; see [Cog01a] for details.

The incompleteness of the description and the failure to clarify how some of the indicated restrictions serve to guarantee type safety, open the potential to incorrect interpretation leading to implementations where type safety could be broken. In addition, the complexity of the approach translates into complexity of the implementation, which requires more effort and is more susceptible to bugs and attacks. Last but not least, the rejection of bytecode produced by compilers is undesirable, as discussed in Section 3.2.2.

3 Improvements

In my opinion, there are two main ways in which the official specification of bytecode verification should be improved. First, the scope of bytecode verification should be clarified. Second, a precise characterization of it should be given.

3.1 Scope

The goal of bytecode verification is to statically establish that certain type safety properties always hold at run time. In this way, the interpreter or just-in-time compiler can omit checks of such properties, resulting in better performance.

The exact type safety properties that bytecode verification must establish can be determined from the specification of bytecode instructions in JS6. That specification includes statements using "must", such as "the top of the stack *must* contain a value of type `int`". As explained in JS6.1, the meaning of "must" is that the execution engine expects the expressed requirements to hold, and it is the task of class verification to make sure they indeed hold[7].

[7] There are a few exceptions to this statement, i.e., specifications of instructions in JS6 that use "must" for checks intended to be performed at run time. An example is the `aastore` instruction.

Some of these requirements are ensured by the resolution process. An example is that the method symbolically referenced by a method invocation instruction must exist, have the indicated argument and return types, and be accessible to the class where the method invocation instruction is.

Other requirements are ensured by checking the static constraints on method code given in JS4.8.1. An example is that the index of a local variable, used as an operand of an instruction, must be within the range of local variables for the method.

The remaining requirements are the type analysis portion of bytecode verification. An example is that the top of the operand stack must contain a value of type `int` when certain instructions are executed.

There are a few complications about the requirements ensured by the type analysis. These complications and their solutions are now discussed.

3.1.1 Disambiguation of Reference Types

As discussed in Section 2.3.2, JS fails to clarify whether bytecode verification should use names only or names plus loaders. The most reasonable interpretation, which has the advantage of allowing lazier class loading, is that bytecode verification uses names only.

This is correct as long as there is an intended disambiguation for class names, accompanied by mechanisms to enforce that the disambiguation in a method is consistent with the one in another method with which objects are exchanged. Consider the following example. A method m_1 has an argument of type `C` (a class name), whose intended disambiguation in m_1 is a class c (identified by the name, `C`, plus a loader). Now, suppose that another method m_2 calls m_1, passing an object of type `C` to it. If `C` is disambiguated to a class c' in m_2, it must be the case that $c = c'$. Otherwise, type safety could be broken.

In the first edition of JS, these issues were not mentioned. Type safety bugs related to these issues were found in earlier implementations of the JVM [Sar97, DFW96].

Those bugs were corrected by the introduction of loading constraints [LB98], described in JS5. Loading constraints ensure that classes exchanging objects, through methods and fields, agree on the actual classes of these objects, not only on their names. Loading constraints are external to bytecode verification. They are part of the class loading mechanisms, which complement bytecode verification to enforce type safety, together with resolution and residual run-time checks.

Formal evidence that bytecode verification can use names only and leave to loading constraints the task of avoiding ambiguities between classes with the same name, is given in [QGC00].

3.1.2 Merging of Reference Types

JS4.9.2 prescribes that the result of merging two class names `C` and `D` is (the name of) their first common superclass. This requires resolving `C` and `D` to actual classes, and

Its specification says that the type of the object to store must be assignment-compatible with the component type of the array. However, the same specification states that a run-time exception is thrown if that is not the case. So, despite the use of "must", it is clear that it is not the task of bytecode verification to ensure that property. The same applies to the other few instructions where "must" is used for a run-time check.

then traversing their ancestry to find their first common superclass. An immediate drawback of this approach is premature class loading. A more serious drawback is that type safety can be broken [CG01].

It turns out that it is possible to avoid this premature loading by assigning finite sets of names, instead of just names, to local variables and operand stack positions [Gol98, Qia99]. Merging of {C} and {D} yields {C, D}. This also prevents type safety from being broken [CG01].

The use of sets of names also provides a cleaner treatment of interfaces. Interface names are treated exactly like class names[8]. Since merging is set union, multiple inheritance of interfaces does not constitute a problem, and there is no need to use `java.lang.Object` as the result of merging. Therefore, no special treatment of `java.lang.Object` is necessary and hence no run-time checks must be performed when `invokeinterface` is executed.

3.1.3 Subtype Relation

Occasionally, during bytecode verification, a reference type `C` is the target of a method invocation instruction whose operand specifies `D` as the class or interface of membership for the method to be invoked. While the existence of the method is checked during resolution, bytecode verification must ensure that `C` is a subtype of `D`. JS4.9.1 prescribes that the two names are resolved in order to check that the desired subtype relation holds. But this results in premature loading.

A better approach is to generate a subtype constraint of the form $\texttt{C} < \texttt{D}$ [Gol98, QGC00, CG01]. Subtype constraints are checked if and when classes are loaded. This is analogous to the treatment of the loading constraints introduced in [LB98]. In fact, these two kinds of constraints can be integrated in the overall class loading mechanisms and formal arguments can be provided that type safety is guaranteed [QGC00].

3.1.4 Protected Fields and Methods

A subtle and often neglected point of bytecode verification is related to protected fields and methods. The specifications of the instructions to access fields and methods include the following requirement: if the referenced field or method is protected, and is declared either in the current class (i.e., the one whose code performs the field or method access) or in one of its superclasses, then the class of the object whose field or method is being accessed must be either the current class or a subclass of it.

This requirement derives from an analogous requirement stated in the specification of the Java programming language [GJSB00]. It has to do with the principle that protected fields or methods of an object can only be accessed from outside their package by code that is responsible to implement that object.

Anyhow, this requirement should be somehow checked by bytecode verification. Otherwise, additional run-time checks would be needed. This is not explicitly mentioned in JS4.8 or JS4.9, but it can be derived from JS6.

[8] In fact, whether a name denotes a class or an interface can be only determined by resolving the name. For instance, if a method has an argument of type `R`, that could denote a class as well as an interface.

A straightforward solution is to resolve the field or method, see if it is protected and declared in the current class or in a superclass, and in that case check that the class type assigned to the operand stack position in question is the current class or a subtype of it.

To avoid premature loading, the last check could be replaced by the generation of a subtype constraints. But the subtype constraint must be satisfied only if the field or method is protected and declared in the current class or a superclass. Otherwise, it does not need to hold, and it is actually wrong to require it to hold.

The solution is to generate a "conditional" subtype constraint, whose condition contains information about the field or method [Cog01b, Cog01c]. Conditional subtype constraints can be integrated with the unconditional ones and checked lazily, if and when classes are loaded and fields or methods are resolved.

3.2 Characterization

The summary of the discussion above is that the type safety properties that bytecode verification must guarantee can be derived from the specification of instructions given in JS6. In addition, bytecode verification can be made a purely functional component of the JVM. It is activated by passing a byte sequence as input, purported to be method code, accompanied by some context information such as the method's signature. The returned answer is either failure or success. In the latter case, some subtype constraints, conditional or unconditional, can also be generated. Bytecode verification never needs to load any class.

If the rectangular space of Figure 1 represents all possible code (i.e., all byte sequences), the larger, solid-line oval delimits the code that is *acceptable* by bytecode verification. This consists of all byte sequences satisfying the following two requirements: first, they must represent sequences of well-formed instructions; second, the instruction sequence must satisfy the type safety properties derived from JS6.

3.2.1 Sequences of Well-Formed Instructions

The first requirement essentially amounts to satisfaction of the static constraints given in JS4.8.1. Static constraints give a precise and computable characterization of which byte sequences represent sequences of valid instructions.

While they can be derived from the specification of instructions in JS6, it may be useful to collect them all in one place of JS—currently, JS4.8.1.

In my opinion, the only necessary or desired improvements are removing the constraint about `new` discussed in Section 2.1 and adding the two constraints about return instructions and execution not falling off the end of code, both discussed in Section 2.2.6. Also, based on the arguments in Section 2.2.1, the term "static constraints" could be replaced by something like "well-formedness constraints".

3.2.2 Type Safety Properties

The second requirement for acceptable code, i.e., the type safety properties that must be guaranteed at run time, can be stated precisely. They essentially depend only on the semantics of bytecode instructions. Unfortunately, it is undecidable whether an instruction sequence satisfies this characterization or not. So, bytecode verification must be a decidable approximation of this characterization, indicated by the mid-size, dashed-line oval in Figure 1: the oval delimits *accepted* code.

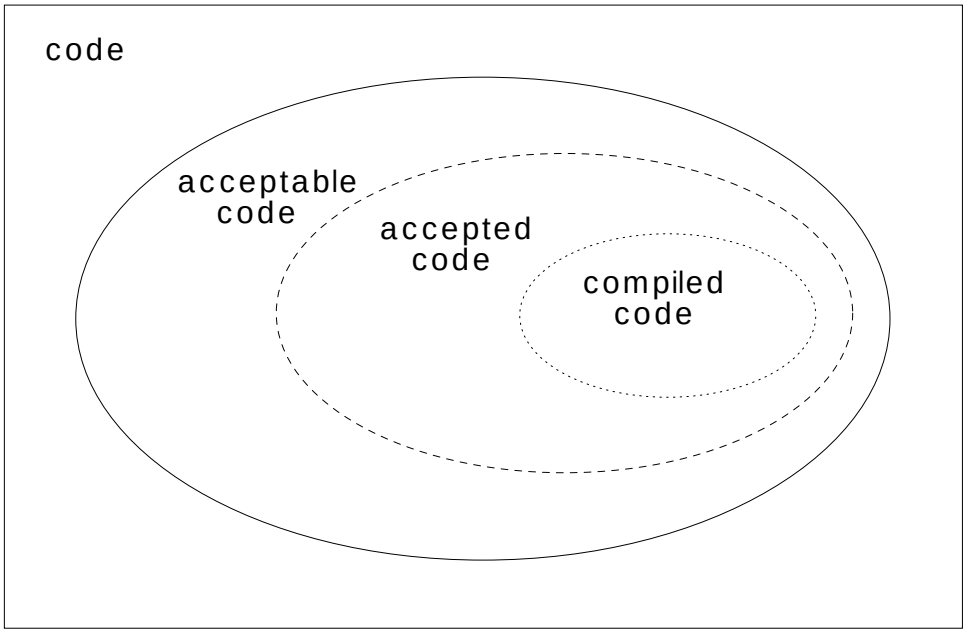


Figure 1: Characterizing bytecode verification

A desideratum is that any code produced by correct compilers is accepted by bytecode verification. It would be disappointing if, after successfully compiling a Java program, the JVM rejected the resulting class files. *Compiled* code (i.e., produced by compilers), is delimited by the smaller, dotted-line oval in Figure 1. So, the containment relationship among the ovals must be as indicated in the figure.

From a point of view, any choice of the accepted-code oval is fine, as long as it is contained in the acceptable-code oval and contains the compiled-code oval. Failure of either containment would cause type unsafety or rejection of compiled code.

While acceptable code can be characterized precisely based on the specification of instructions, compiled code depends on how compilers are implemented. Even if all current compilers used common compilation strategies, future compilers might use new strategies to generate better code. Therefore, the compiled-code oval is a moving target, and hence not very suitable to "universal" characterization.

The best approach is to give a precise characterization of accepted code. All implementors of the JVM should write bytecode verification algorithms that exactly recognize the specified subset. The specification of accepted code would also become a contract between developers of compilers and of the JVM: as long as a compiler produces code that falls inside the subset, that code is accepted by bytecode verification.

The definition of the accepted-code subset must embody an optimal trade-off between two criteria. The first criterion is that bytecode verification should be as simple and efficient as possible. The second criterion is that it should accept as much code as possible. Privileging the first could limit future compilers or reject code from current compilers; privileging the second could make implementations more susceptible to errors and hence to attacks exploiting the errors.

Based on the above considerations, the following are the necessary or desired improvements for JS, in my opinion. First of all, the notion of structural constraints should be purified of those forward references to the computable approximation about operand stack size (Section 2.3.1), uninitialized objects (Section 2.3.4), and subroutines (Section 2.3.5). Redundant constraints should also be eliminated. The contradiction regarding whether a constructor can store values into fields of an uninitialized object should be resolved, perhaps by disallowing that, because it simplifies verification [Cog01b, Cog01c]. Also, based on the arguments in Section 2.2.1, the term "structural constraints" could be replaced by something like "typing constraints".

In this way, these constraints would express all the type safety properties that acceptable code must satisfy. Note that these properties are "homogeneous" in some sense—see discussion in Section 2.2.7. While these can be derived from JS6, it may be useful to collect them all in one place of JS—currently, JS4.8.2.

More important, the algorithmic description in JS4.9 should be made much more complete and precise. It should be pointed out that the algorithm is an approximation of the undecidable constraints. The improved treatments of object initialization (Section 2.3.4), merging of reference types (Section 3.1.2), subtype relation (Section 3.1.3), and protected members (Section 3.1.4) should be incorporated. A solution to the problem, mentioned in Section 2.3.5, of compiled bytecode being rejected because of the treatment of subroutines, is given in [Cog01a]: this solution should be also incorporated. A formal description that fulfills all of these criteria is given in [Cog01b].

Finally, it should be made clear how bytecode verification cooperates with the other type safety mechanisms of the JVM. Issues include the use of class names in bytecode verification and the disambiguation of them to classes, as well as the generation of subtype constraints as opposed to loading classes.

4 Conclusion

The topic of Java bytecode verification has attracted the interest of several researchers. As a result, there has been a large number of publications on the subject [CG01, CGQ98, CL99, FC00, FC01, FM99a, FM99b, FM99c, Fre98, Gol98, HT98, Jon98, KN00, Nip01, O'C99, Pus99, PV98, Qia99, Qia00, Req00, RR98, SA99, SSB01, Yel99]. These works have greatly contributed to the clarification of key issues in bytecode verification, including pointing out some inadequacies in JS and proposing improvements.

To my knowledge, this paper is currently the only work to provide a comprehensive analysis of the official specification of bytecode verification and a comprehensive plan for improvement. Some details of the analysis and improvements (e.g., subroutines) are covered in other papers that are explicitly referred from this paper.

The need and desire to improve JS, in particular the description of class verification, "ideally to the point of constituting a formal specification", is explicitly stated in the Appendix of JS. This paper contributes to that goal.

References

[CG01] Alessandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency—Practice and Experience*, 2001. To appear.

[CGQ98] 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*, October 1998.

[CL99] Ludovic Casset and Jean Louis Lanet. A formal specification of the Java bytecode semantics using the B method. In *Proc. 1st ECOOP Workshop on Formal Techniques for Java Programs*, June 1999.

[Cog01a] Alessandro Coglio. Java bytecode subroutines demystified. Technical report, Kestrel Institute, 2001. Forthcoming at `http://www.kestrel.edu/java`.

[Cog01b] Alessandro Coglio. Java bytecode verification: A complete formalization. Technical report, Kestrel Institute, 2001. Forthcoming at `http://www.kestrel.edu/java`.

[Cog01c] Alessandro Coglio. Java bytecode verification: Another complete formalization. Technical report, Kestrel Institute, 2001. Forthcoming at `http://www.kestrel.edu/java`.

[DFW96] Drew Dean, Edward Felten, and Dan Wallach. Java security: From HotJava to Netscape and beyond. In *Proc. IEEE Symposium of Security and Privacy*, pages 190–200, May 1996.

[FC00] Philip Fong and Robert Cameron. Proof linking: Modular verification of mobile programs in the presence of lazy, dynamic linking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):379–409, October 2000.

[FC01] Philip Fong and Robert Cameron. Proof linking: Distributed verification of Java classfiles in the presence of multiple classloaders. In *Proc. 1st Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 53–66. USENIX, 2001.

[FM99a] Stephen Freund and John Mitchell. A formal framework for the Java bytecode language and verifier. In *Proc. 14th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, volume 34, number 10 of *ACM SIGPLAN Notices*, pages 147–166, October 1999.

[FM99b] Stephen Freund and John Mitchell. A type system for Java bytecode subroutines and exceptions. Technical Note STAN-CS-TN-99-91, Computer Science Department, Stanford University, August 1999.

[FM99c] Stephen Freund and John Mitchell. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1196–1250, November 1999.

[Fre98] Stephen Freund. The costs and benefits of Java bytecode subroutines. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley, second edition, 2000.

[Gol98] Allen Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security (CCS'98)*, pages 49–58, November 1998.

[Gon99] Li Gong. *Inside Java™ 2 Platform Security*. Addison-Wesley, 1999.

[HT98] Masami Hagiya and Akihiko Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. In *Proc. 5th Static Analysis Symposium (SAS'98)*, volume 1503 of *Lecture Notes in Computer Science*, pages 17–32. Springer, September 1998.

[Jon98] Mark Jones. The functions of Java bytecode. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.

[KN00] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. In *Proc. 2nd ECOOP Workshop on Formal Techniques for Java Programs*, June 2000.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java™ virtual machine. In *Proc. 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, volume 33, number 10 of *ACM SIGPLAN Notices*, pages 36–44, October 1998.

[LY99] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[Nip01] Tobias Nipkow. Verified bytecode verifiers. In *Proc. 4th Conference on Foundations of Software Science and Computation Structures (FOSSACS'01)*, volume 2030 of *Lecture Notes in Computer Science*, pages 347–363. Springer, April 2001.

[NNH98] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1998.

[O'C99] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 70–78, January 1999.

[Pus99] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Proc. 5th Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 89–103, March 1999.

[PV98] Joachim Posegga and Harald Vogt. Java bytecode verification using model checking. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.

[QGC00] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *Proc. 15th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, volume 35, number 10 of *ACM SIGPLAN Notices*, pages 325–336, October 2000. Long version available at `http://www.kestrel.edu/java`.

[Qia99] 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™*, volume 1523 of *Lecture Notes in Computer Science*, pages 271–312. Springer, 1999.

[Qia00] Zhenyu Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4):638–672, July 2000.

[Req00] Antoine Requet. A B model for ensuring soundness of a large subset of the Java Card virtual machine. In *Proc. 5th ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'00)*, pages 29–45, April 2000.

[RR98] Eva Rose and Kristoffer Rose. Lightweight bytecode verification. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.

[SA99] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(1):90–137, January 1999.

[Sar97] Vijay Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997. Available at `http://www.research.att.com/vj/bug.html`.

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. *Java™ and the Java™ Virtual Machine—Definition, Verification, Validation*. Springer, 2001.

[TH99] Akihiko Tozawa and Masami Hagiya. Careful analysis of type spoofing. In *Proc. Java-Informations-Tage 1999 (JIT'99)*, pages 290–296. Springer, September 1999.

[Yel99] Phillip Yelland. A compositional account of the Java Virtual Machine. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 57–69, January 1999.

Reinforcing Fragile Base Classes

Kees Huizing and Ruurd Kuiper**

Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven,
The Netherlands,
keesh@win.tue.nl, wsinruur@win.tue.nl

Abstract. The Fragile Base Class problem is approached from the angle of a proof system for class invariants. It is shown that the source of the FBC itself can be understood as a problem of dynamic binding of predicates in the correctness proof of invariants. A solution is presented based on an extension of the concept of behavioural subtyping and the novel notion of cooperative contracts. Thus, flexible boundaries of reuse can be specified for each class.

1 Introduction and Position

When a developer believes in the assumption that changing the implementation of a method within the boundaries of the specification (contract) should leave the behaviour of other methods unchanged, he may be caught off guard by the Fragile Base Class problem (FBC).

Specifically: if a reuser subclasses a class C to a class D by inheritance, and later the provider of C revises C, keeping its contract, the reuser may find that D no longer satisfies its contract.

For this, inheritance is often blamed and therefore rejected as a vehicle for code reuse. This we regard as an unacceptable limitation on code reuse. Similarly, we regard disallowing the provider to change the base class as too restrictive.

In the literature, the solution to the FBC is mainly sought in syntactically restricting the allowed class dependencies during development. We propose a different approach that leads to a more fine-grained, or if you like, semantic, check of the dependency between classes and methods.

When we study the proof obligations in a formal proof system, we can pin down the problem to dynamic binding. Either the provider of the base class has to prove the invariant of a (future!) subclass, or the subclass developer needs access to the implementation of the base class to prove the invariant.

We solve this problem in two steps. First, we remove the dependency of the base class on the derived class by extending the notion of behavioural

** The authors are partially supported by ITEA DESS.

subtyping. This means that the developer of the subclass has to prove additional clauses for methods that are inherited.

The second step of the solution is a notion of cooperative contract. The developer of the base class can use a kind of parametrized postconditions that allow the reuser to derive more properties about a method than could be derived from an ordinary contract, without giving the reuser access to the implementation. Without cooperative contracts some code would not be amenable to reuse.

By choosing a level of cooperation in the contract, the provider can define the boundaries of reuse.

Workshop Position: Both providers and reusers of classes should be allowed to make adaptations to classes; the FBC should be understood and remedied in the context of a proof system:

1. user developments from the base classes should be in accordance with a new notion of behavioural subtyping;
2. provider's changes to a base class should respect a new notion of cooperative contract.

2 The Fragile Base Class

The following example is modified from an example in [8]. Consider a class `Set` that represents a set of integers with methods `add(int)` and `addSquared(int)`. The first method simply adds an integer to the set; the second one computes the square of an integer and then calls `add()` to add the result to the set. In Java this class could be defined as follows.

```
class Set {
    ⟨ I: true⟩
    ...

    public add(int n) {
        ...
        ⟨post: s = s~ ∪ {n} ⟩
    }

    public addSquared(int n) {
      add(n*n);
       ⟨post: s = s~ ∪ {n^2} ⟩
    }
}
```

We write assertions surrounded with angle brackets to avoid confusion with the Java curly brackets.

Here, s represents the contents of the set; a variable in a postcondition with a $\sim$-symbol attached represents the old value, i.e., the value at the start of the method. To keep the example simple, we used a trivial integrity check.

Now a subclass `CountingSet` of `Set` is made that adds an instance variable keeping track of the number of elements in the set. For this purpose, method `add()` is overridden and we get the following class.

```
class CountingSet extends Set {
    ⟨ I: |s| = size ⟩
    int size;
    ...

    public add(int n) {
        size++;
        ...
        ⟨ post: s = s~ ∪ {n} ∧ size = size~ + |{n} \ s~|⟩
    }
```

Here, I is the invariant of the class, which specifies the consistent states of all objects of this class. A user of an object of this type may expect the invariant to hold. As a consequence, all public methods should establish the invariant at method return.

Note that the postcondition of `add()` has been strengthened with information about `size`.

`CountingSet` has no need to override `addSquared()`, since this latter method calls `add()` and thanks to dynamic binding, the correct version of `add()` will be executed, depending on the type of the object underhand.

This is an example of the design pattern Template Method, where execution of a primitive operation (`add()` in this example) is delegated to a subclass [1].

Now suppose the class `Set` is revised. For some reason, maybe of efficiency, the method `addSquared` does not call `add()` anymore, but puts the element directly into the set. This seems harmless, since the contract of `addSquared` is maintained. For `Set`-objects everything works. For `CountingSet`-objects however, things go wrong. Since `add()` isn't called anymore, the variable `size` will not be updated when an element is added by means of `addSquared()`. As a consequence, the invariant **I** is not maintained.

This problem is known in the literature as the Fragile Base Class Problem ([5]). This version is called the semantic problem. There is also a *syntactic version*, which concerns the problems when the subclass is not recompiled after revision of the base class. Since Java solves the syntactic problem, the semantic one is even more important. In this paper, we will mean the semantic problem when we speak about the Fragile

Base Class problem. There are several related problems, such as infinite recursion as a consequence of incorrect overriding. As we only consider partial correctness here, we will not treat this problem. For an extensive list of issues related to the Fragile Base Class problem, see [8].

3 Pinning down the problem

When we use a formal verification system, it becomes clear where the intuition makes an unjustified assumption that causes the FBC problem. For this purpose, we use a proof system along the lines of the one presented in [2]; other systems such as [7] could do as well. The problem is centered around the concept of dynamic binding.

3.1 Dynamic binding

In Java, the method call `b.m()` means: choose the first definition of `m` that you find when going up in the type hierarchy, starting in the class to which object `b` belongs, and then execute this method in the context of object `b`. With "object `b`" we mean the object that results from evaluating the expression `b`. Note that this value depends on the state, and so does the class of which method `m()` is chosen. This state dependency of method choice is called *dynamic binding*.

We extend this notation to predicates. The expression $b.I$ means: interpret predicate I in the context of object b. As with method invocation, the choice of I depends on the class of b.

To simplify the discussion, without losing anything of the essence, we formulate a proof rule for a parameterless method in the absence of recursion.

$$\frac{\begin{array}{l}\text{for all classes } C \text{ containing } m\text{:}\\ \langle \texttt{this}.\mathsf{pre} \wedge \texttt{this}.I \wedge \ldots\rangle\ \mathsf{body}_{m_C}\ \langle \texttt{this}.\mathit{post} \wedge \texttt{this}.I \wedge \ldots\rangle\end{array}}{\langle b.\mathsf{pre}\rangle\ b.m()\ \langle b.\mathsf{post} \wedge b.I\rangle}$$

The premise requires proofs for the bodies of all the methods that could be executed as a result of the call $b.m()$. We keep on the safe side by taking every method called m. Since the correctness of all these bodies should be proven anyhow, albeit not for this specific call, we are not interested in removing this redundancy from the rule.

On the dots, more invariants may appear, depending on the proof system and the circumstances. This does not interfere with the exposition, however; details can be found in [2].

The object `this` in the premise refers to the object on which the method body is executed. We write `this.pre` etc. to stress that the predicate `pre`

has to be evaluated in the context of this object and likewise for post and I.

Now dynamic binding puts us a problem. What are the predicates pre, post, and I? Since we don't know the dynamic type of b, we don't know which version of $m()$ will be executed and hence, which precondition, postcondition, and invariant will apply.

3.2 The solution

In practice, the pre- and postconditions will not arbitrarily change in a subtype (inheritance) hierarchy. A method in a subclass should at least fulfill the contract of the method it overrides. This idea is called *behavioural subtyping* after [4] and it amounts to the requirement that, when $D <: C$ (D is a subtype of C), the following implications should hold:

- $\mathsf{pre}_C \Rightarrow \mathsf{pre}_D$
- $\mathsf{post}_D \Rightarrow \mathsf{post}_C$
- $I_D \Rightarrow I_C$

This is a principle of good OO design and it enables us to replace in many cases dynamic reference to predicates with static ones. When b is an object and P is a predicate (pre/postcondition or invariant), we use the notation $b{:}P$ for the *staticly bound* P, i.e., if C is the static type of expression b, then $b{:}P = (C)b.P$, where (C) is the type cast operator. We assume that the static type of an expression is always known. Surely it can be derived from the program context. Under assumption of behavioural subtyping, we can formulate a friendlier proof rule for method call:

$$\frac{\begin{array}{l}\text{for all classes } C \text{ containing } m\text{:}\\ \langle \texttt{this}{:}\mathsf{pre} \wedge \texttt{this}{:}I \wedge \ldots \rangle\ \mathsf{body}_{m_C}\ \langle \texttt{this}{:}\mathsf{post} \wedge \texttt{this}.I \wedge \ldots \rangle\end{array}}{\langle b{:}\mathsf{pre} \wedge \ldots \rangle\ b.m()\ \langle b{:}\mathsf{post} \wedge b{:}I \wedge \ldots \rangle}$$

In this rule, all occurrences of dynamic binding of predicates are removed, except one: this.I in the premise. In a minute we will see how to get rid of this dynamic binding also.

The dynamic references in the *conclusion* can be replaced by static ones because of behavioural subtyping. This is in fact an application of the rule of consequence: $\langle P \rangle\ S\ \langle Q \rangle$ implies $\langle P' \rangle\ S\ \langle Q' \rangle$ if $P' \Rightarrow P$ and $Q \Rightarrow Q'$.

Substituting this:pre and this:post in the *premise* is allowed because of the rules for method choice in Java and most other OO languages. If the dynamic type D of this differs from the static type C, there can be no method m defined in D, since otherwise method m from D was chosen instead of C.

For the invariant in the premise, however, this reasoning does not apply, as each type may have its own invariant, independent of whether m is overridden or not. The stronger obligation to prove `this`.I remains here. It is here that the intuition unjustifiedly assumes that proving the invariant of class C is enough.

In terms of the fragile base class example: The method `addSquared` has been defined in class `Set` and hence the proof of its correctness is performed in the context of `Set` (the static type of `this` is `Set`). Since `addSquared` is not redefined in `CountingSet`, exection of this method may well be in the context of an object of the class `CountingSet` and then the *dynamic* type of `this` is `CountingSet`.

In the most pregnant examples of the Fragile Base Class, the development of base class C is both in time and place remote from the development of derived class D. E.g., company X produces a class library of which C is a part. Then company Y uses this library and reuses C by inheritance. X cannot be held responsible for the invariants of subclasses yet to be made. On the other hand, passing the proof obligation to the developer of D would require X to give its users insight in its source code. This is unpractical and from a commercial point of view unwanted. Furthermore, it denies the concept of abstraction by contract.

So the only reasonable proof obligation in the premise is `this`:I.

This obligation is too weak, however, and would render the proof rule unsound. To solve this, we propose a strengthening of the notion of behavioural subtyping: When D has a stronger invariant than C, it should be proven from the contract (specification) provided by C.

Definition 1 (Reinforced Behavioural Subtyping). *Type D is a reinforced behavioural subtype of C if:*

1. $I_D \Rightarrow I_C$
2. *for every method m_D overriding m_C:*
   $\mathsf{pre}_{m_C} \Rightarrow \mathsf{pre}_{m_D}$
   $\mathsf{post}_{m_D} \Rightarrow \mathsf{post}_{m_C}$
3. *for every non-private method m not overridden in D and any object d of type D:*
   $d{:}(I \restriction \mathsf{inv}_m \wedge I^{\sim} \wedge \mathsf{post}_m \wedge I_C) \Rightarrow d{:}I$.

Here, the set inv_m consists of the variables that are not changed during execution of m. It is derived from the specification of m. (This set is specified explicitly or implicitly by specifying which variables are allowed to change. Here, we do not elaborate on how to specify inv precisely.)

The predicate $I^{\sim}$ is defined as the predicate I with all variables adorned with the symbol $\sim$. It simply says that I was true at the start of the execution of m.

Obligation 3 is new. It requires the developer of D to prove the possibly stronger invariant of the derived class after execution of m. For this he

may use the invariant of the base class I_C, the postcondition of m, the stronger invariant restricted to variables that do not change under m, and the knowledge that this stronger invariant held at the start of m.

Since this proof obligation is about a method that is not overridden in D, it is possible that the developer of D has no access to the contract of m, e.g., if m is a private method of C of a supertype thereof. This would make obligation 3 impossible to fulfill. We take the approach, however, that private methods need not establish the invariant, as in [3].

This proof obligation leaves not much room for the developer of the derived class to strengthen the invariant. In the Fragile Base Class example, e.g., the invariant of `CountingSet` cannot be proven. For situations like this we propose the concept of *cooperative contracts*. The postcondition of a method may contain references to postconditions of other methods, in particular methods to be overridden in subclasses. In proof obligation 3 of reinforced behavioural subtyping, the interpretation of these references depends on the type of d. This way, such a reference may result in a stronger predicate if the corresponding method is overridden in the subclass. To see this, suppose post_m refers to a post_n, the postcondition of another method n. When this method is overridden in D with a stronger postcondition, the conjunct post_m will be effectively stronger too, since it is evaluated in the context of an object of class D. If done right, this makes it possible to prove the stronger invariant of class D.

So under the assumption of Reinforced Behavioural Subtyping we have the following **proof rule for non-recursive method call**:

$$\frac{\begin{array}{l}\text{for all classes } C \text{ containing } m\text{:}\\ \langle \texttt{this}{:}\mathsf{pre} \wedge \texttt{this}{:}I \wedge \ldots \rangle\ \mathsf{body}_{m_C}\ \langle \texttt{this}{:}\mathsf{post} \wedge \texttt{this}{:}I \wedge \ldots \rangle\end{array}}{\langle b{:}\mathsf{pre} \wedge \ldots \rangle\ b.m()\ \langle b{:}\mathsf{post} \wedge b{:}I \wedge \ldots \rangle}$$

Now we can go back to the example of the Fragile Base Class and change the postcondition of `addSquared()` to:

$$\mathsf{post}_{addSquared} : \mathsf{post}_{add}(n^2)$$

Proof obligation 3 now becomes (we assume there are no relevant invariance properties in the specification of `addSquared`):

$$d{:}(|s^{\sim}| = size^{\sim} \wedge \mathsf{post}_{add}(n^2)) \Rightarrow d{:}(|s| = size)$$

which is equivalent to

$$d{:}(|s^{\sim}| = size^{\sim} \wedge s = s^{\sim} \cup \{n^2\} \wedge size = size^{\sim} + |\{n^2\} \setminus s^{\sim}|) \Rightarrow d{:}(|s| = size)$$

which follows easily from the facts

$$|x \cup y| = |x| + |y| - |x \cap y|$$

and

$$|x \setminus y| = |x| - |x \cap y|$$

Of course, we could have strengthened the postcondition of `add` in the class `CountingSet` with the invariant. This would have trivialized the proof above, however. We believe that the current proof is more interesting because it shows the role for the predicate $I^{\sim}$.

This shows that the correctness proof of the subclass is based on the specification only of the base class, in contrast to the original situation where formal correctness needs the code of the base class. As a consequence, this allows for revising the base class, within the boundaries of the contract. E.g., in the method `addSquared` above, the implementation of the integrity check could be altered without fear of the Fragile Base Class problem. The provider of the base class can decide how cooperative the contract will be.

4 Conclusion

We have shown how to use formal methods to better understand the Fragile Base Class problem and how to solve the problem. Other approaches to this problem can be found in the literature. [3] has an extensive framework to specify properties of (Java) programs. It elaboreates on properties of the call graph, wehereas we use a more assertional approach, concentrating on the validity of invariants.

In [6], the notion of cooperation contracts is introduced as part of a solution of the Fragile Base Class problem. Their approach is purely syntactical and their notion is not the same as our cooperative contracts, which play a central role in proving the validity of assertions.

Mikhajlov and Sekerinski in [8] formulate several requirements that guarantee true refinement of superclasses. These requirements are more restrictive and do not allow the example of a subclass invariant that combines instance variables of the subclass with those of the superclass as in the `CountingSet` example.

We believe that we have clarified the Fragile Base Class problem, and, with it, similar problems connected with inheritance and dynamic binding in object-oriented languages. The novel approach of cooperative contracts allows for a fine grained semantic solution to the problem of safe code reuse in the subtle frameworks that can be found in many object oriented program designs.

References

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
2. K. Huizing, R. Kuiper, and SOOP. Verification of Object Oriented Programs Using Class Invariants. In *Fundamental Approaches to Software Engineering (FASE 2000)* (Maibaum, Ed.), Berlin, 2000, Lecture

Notes in Computer Science, Vol. 1783, Springer-Verlag, Berlin, 2000, pp. 208–221.
3. Clyde Ruby and Gary T. Leavens. Safely Creating Correct Subclasses without Seeing Superclass Code. In *OOPSLA 2000 – Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Minneapolis, Minnesota. SIGPLAN Vol. 35(10), pp. 208–228, 2000.
4. B. Liskov and J. Wing, *A behavioral notion of subtyping*, ACM TOPLAS, 16:6, pp. 1811–1841, 1994.
5. C. Szyperski, *Component software: Beyond object-oriented programming*, Addison-Wesley, 1998.
6. M. Mezini. Maintaining the consistency and behavior of class libraries during their evolution. In *Conference Proceedings of OOPSLA '97*, ACM SIGPLAN Notices, Vol. 32(10), pp. 1–21, Oct. 1997.
7. A. Poetzsch-Heffter and P. Müller, *Logical foundations for typed object-oriented languages*, in D. Gries and W.P. de Roever, Eds., Programming Concepts and Methods (PROCOMET), 1998.
8. L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *ECOOP'98 – Object-Oriented Programming 12th European Conference* (E. Jul, Ed.), Brussels, July 1998, pp. 355–382, Lecture Notes in Computer Science Vol. 1445, Springer-Verlag, 1998.

Java Separate Type Checking is not Safe
(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. Java supports *separate type-checking* in the sense that compilation can be invoked on a single source fragment, and this may enforce type-checking of other either source or binary fragments existing in the environment. However, the Java specification does not define precise rules on how this process should be performed, therefore the outcome of compilation may strongly depend on the particular compiler implementation. Furthermore, rules adopted by standard Java compilers, as SDK and Jikes, can produce binary fragments whose execution throws linking related errors. We introduce a simple framework which allows to formally express the process of separate compilation and the related formal notion of type safety. Moreover, we define, for a small subset of Java, a type system for separate compilation which we conjecture to be safe.

1 Introduction

Traditional type systems for programming languages define the well-formedness of self-contained programs, and are said to be *safe* if the (result of the compilation of) a well-typed program is guaranteed to well-behave at run time (see [6, 4, 5] for the Java case).

However, in languages supporting *separate compilation* and *dynamic linking*, like Java, this simple framework is no longer adequate. Indeed, it is possible to type-check a single source fragment in a context where other fragments are present either in source or in binary form. Hence, there are two main new ingredients to be considered in typing rules: checks can be performed not only on source, but also on binary fragments, and, for type-checking a fragment, it can be necessary to type-check other (source or binary) fragments, following some strategy.

Moreover, the output of the compilation phase is not a self-contained executable program, but a collection of binary fragments which can be linked and executed in many different ways. Hence, the type safety notion must be expressed in a more flexible form.

* Partially supported by Murst - TOSCA Teoria della Concorrenza, Linguaggi di Ordine Superiore e Strutture di Tipi.

In this paper, we introduce a simple framework for separate compilation, modeled as a function which, given a set of fragment names and a *compilation context* consisting of both source and binary fragments, produces a collection of binary fragments, and we define a related notion of type safety.

Our aim is to face the following problems related to Java separate compilation.

- There is *no* specification of separate compilation in [2], hence the outcome of compilations may strongly depend on the particular compiler implementation.
- Rules adopted by existing compilers can be quite complex and cannot be easily explained informally.
- As known by Java programmers, rules adopted by standard Java compilers, as SDK and Jikes, can produce binary fragments whose execution throws linking related errors. This seems in contradiction with the fact that type safety results have been proved for the Java language [6, 4, 5]; the explanation, as we will illustrate in more detail in the following, is that these type systems, and the related type safety results, are only related to a special case, which is the compilation of a self-contained set of source fragments.

Our framework is a formal basis for defining type systems for languages supporting separate compilation, notably Java, and formally reasoning about them by defining and proving good properties. For reasons of space, here we focus on type safety, however there exist other kinds of good properties one could expect from separate compilation (see end of Sect.3 and the Conclusion).

In order to illustrate our approach, we define, for a small Java subset, a type system for separate compilation which we conjecture to be safe (a formal proof would require the definition of a simple execution model, not considered here for lack of space).

The work presented in this paper is a first step towards the formal definition and comparison of different type systems for Java separate compilation, corresponding, e.g., either to standard Java compilers, , or to extended compilers which perform additional checks. The overall motivation of this research is the following.

As illustrated in detail in the following, standard compilers perform very few checks on binary fragments, relying on the fact that these checks can be in practice delegated to the JVM[1], which finds linking related errors and throws corresponding exceptions (see examples in Sect.2), thus guaranteeing that execution does not crash. However, we argue that this is not a good enough motivation. Indeed, the fact that the JVM has a run-time verifier (hence intercepts error situations) cannot be used as a justification for not trying to anticipate at compile-time checks which actually *can* be performed earlier; otherwise, following the same principle, one could also throw away checks on source fragments since in any case the fact that the execution does not crash is guaranteed by the bytecode verifier, hence these checks are in a sense redundant. In our opinion,

[1] Java Virtual Machine.

even though the run-time verification cannot, of course, be eliminated in Java[2], it is worthwhile to investigate the possibility of anticipate at compile-time as many checks as possible, as it is in the long tradition of type systems. The obvious advantage is earlier error detection; then, in principle, the possibility that execution in a context of "certified" bytecode fragments obtained by a "smart" compiler could be performed without some run-time checks (as it is already the case for a context of binary fragments resulting from the compilation of all source fragments).

The paper is organized as follows. In Sect.2 we present simple examples to illustrate type-checking rules adopted by the SDK and Jikes compilers and to show that these rules are not safe. In Sect.3 we introduce our framework and formally express type safety. In Sect.4 we show, for a small subset of Java, a type system for separate compilation which we conjecture to be safe. Finally, Sect.5 summarizes the contribution of the paper and outlines further work.

2 Some motivating examples

In this section we illustrate by means of some examples the type-checking rules adopted by the two Java compilers SDK 1.3 and Jikes 1.11 (which apparently seem to coincide[3]), and we show that these rules are not safe.

In the following, we will call *compilation context* all the source[4] and binary fragments which are available to the compiler (the notion will be formalized in the next section). If both the source and the corresponding binary fragment are present for a class, then standard compilers inspect the binary and ignore the source, while the source is inspected if the binary is obsolete, that is, source has been changed after last compilation.

The first example illustrates non-safe behaviour due to the fact that, when checking a binary fragment, standard compilers do not enforce checking of all used fragments.

```
class A{ static void main(String[] args){new B().m();} }
class B{ int m(){return new C().m();} }
class C{ int m(){return 1;} }
```

If, in a compilation context cc_0 consisting of the three source fragments, we invoke the compiler on `A.java`, then compilation of `B.java` and `C.java` is enforced, so that, after compilation, we obtain a new context cc_1 where the binary fragments of the three classes are available. However, if we re-compile `A.java` in the context cc_2 obtained by removing from cc_1 the binary fragment of `C`, then re-compilation of `C.java` is not enforced[5], therefore we obtain again the context cc_2 (hence, no static error has been detected); however, if we try to execute class `A` in this context, then error `NoClassDefFoundError` is thrown.

[2] To deal with fragments which are not known to be the result of some compilation.
[3] Except that Jikes supports compilation options that enforce more checks.
[4] We assume for simplicity a unique file for each class.
[5] In Jikes re-compilation of `C.java` can be enforced with the option `+F` or `+U`.

Indeed, in standard compilers, when a fragment named `N` is checked, this always enforces (transitively) checking the parent of `N`, regardless `N` is in source or binary form[6], whereas used fragments are (transitively) checked only when `N` is in source form. This rule is not safe since it can lead to linking related errors, as shown above. In the type system in Sect.4, instead, parent and used fragments are always (transitively) checked.

Next examples illustrates cases in which the non-safe behaviour is not related to dependencies among checking fragments, but rather to the fact that some checks which could be in principle performed on binary fragments are not actually performed.

In the context cc_1, as previously defined, assume to modify `C.java` in the following way:

```
class C{ C m(){return new C();} }
```

Let cc'_2 denote the context obtained from cc_1 by modifying the source fragment of `C` as shown above. If we re-compile both `A.java` and `C.java`, then we obtain a new context cc'_3 (hence, no static error has been detected). However, in this new context, the execution of class `A` throws `NoSuchMethodError`. The problem is that, when checking `B.class`, compilers do not check that class `C` should have a method `int m()`, as would be checked if only the source of `B` were available.

A similar situation arises in the following example:

```
class A{ static void main(String[] args){new B().m()} }
class B{ D m(){return new C();} }
class C extends D {}
class D {}
```

Assume, analogously to the example above, to first compile all fragments, then modify `C.java` as follows:

```
class C {}
```

If we re-compile `A.java` and `C.java` in this context, then we get no static error, but the execution of class `A` throws `VerifyError`. The problem, again, is that, when checking `B.class`, compilers do not check that class `C` should be a subtype of `D`, as would be checked if only the source of `B` were available.

Finally, consider the following source fragments:

```
class A{ static void main(String[] args){new B().m()} }
class B{ int m(){return new C().m();} }
class C extends D {}
class D { int m(){return 1;} }
```

[6] Hence in an analogous example where `A` extends `B` which extends `C` re-compilation of `C.java` would be enforced even by checking `B.class`, thus causing no run-time error.

and the situation in which we start from the context containing the source fragments above, we compile all of them, and then we remove `B.java`[7] and modify `C.java` as follows:

```
class D {}
class C extends D{ int m() { return 1;} }
```

Again, re-compiling `A.java`, `C.java` and `D.java` we get no static error and obtain a context in which the execution of class `A` throws `NoSuchMethodError`. Here the problem is that the call `new C().m()` in `B.class` is annotated with the class `D` where method `m` was previously declared and the JVM verifies that `m` is actually declared either in `D` or in some superclass of `D`. Note that, as in the preceding example, in presence of `B.java` the problem can be fixed by re-compiling it; in this case, however, no static error is detected, but a new binary fragment for `B` where the call is annotated with `C` is produced.

In summary, these three examples show that standard compilers do not perfrom on binary fragments some checks which could be possibly performed at compile-time. These are either checks which are performed on source fragments, or checks related to additional informations stored in the bytecode which make it less "abstract" w.r.t. to source. In the type system we define in the following, on the contrary, these checks on binaries are performed, hence in the three examples a static error would be raised.

As final remark, the examples above also show that rules for Java separate compilation are not trivial to understand and express and that, therefore, the behavior of the existing compilers cannot be always easily predicted; other examples, not related to violating type safety, where the compilers exhibit unexpected behavior can be found in [1].

3 Framework

We introduce now a simple framework allowing to model separate compilation and to express the property of type safety in a formal way.

*Notations.*We denote by $[A \rightharpoonup_{fin} B]$ the set of the *finite partial functions* from A into B, that is, functions from A into B which are defined on a finite subset of A. For each $f \in [A \rightharpoonup_{fin} B]$, we set $Def(f) = \{a \in A | f(a) \in B\}$. □

Let us denote by $\mathbb{C}$ the set of fragment names, ranged over by c, and by $\mathbb{S}$ and $\mathbb{B}$ the set of source and binary fragments, respectively. We assume that $\mathbb{S} \cap \mathbb{B} = \emptyset$. In the Java case, fragment names will be class/interface names, source fragments will be `.java` files containing (for simplicity) exactly one class/interface declaration, and binary fragments will be `.class` files. However, the model we present is general and can be applied to fragments of different nature.

[7] In presence of `B.java` the counter-example works as well, but the error can be detected by forcing its re-compilation.

A *compilation context* cc is a pair $\langle cc_b, cc_s \rangle \in CC = [\mathbb{C} \rightharpoonup_{fin} \mathbb{B}] \times [\mathbb{C} \rightharpoonup_{fin} \mathbb{S}]$. In general $Def(cc_b) \cap Def(cc_s) \neq \emptyset$, since for some fragment both the source and the binary can be available (intuitively, this means that the binary is obsolete).

The results of (successful) compilations are finite partial functions from class names into binary fragments. Hence, we can model the compilation process by a (partial) function:

$$\mathcal{C} : \wp(\mathbb{C}) \times CC \rightharpoonup [\mathbb{C} \rightharpoonup_{fin} \mathbb{B}]$$

where $\mathcal{C}(C, \langle cc_b, cc_s \rangle) = cc'_b$ intuitively means that the compilation, invoked on fragments with names in C, in the compilation context consisting of binary fragments cc_b and source fragments cc_s, produces binary fragments cc'_b.

We introduce now the formal property of type safety for separate compilation. For our purposes, we can abstract from all detailes of the linking and execution model and just assume a very general judgment of the form $cc_b \vdash c \rightsquigarrow \mathrm{OK}$ which is valid if and only if execution of c in the context of binary fragments cc_b does not throw any linking related error. In the Java case, for instance, this judgment corresponds to start execution from class[8] c in a context where all binaries in cc_b are available to the JVM, hence some of them could be dynamically linked during execution.

Definition 1. *A compilation function $\mathcal{C}$ is* type safe *iff for any compilation context $\langle cc_b, cc_s \rangle$ and set of fragment names C, if $\mathcal{C}(C, \langle cc_b, cc_s \rangle) = cc'_b$, then, for any $c \in Def(cc'_b)$, $cc'_b \vdash c \rightsquigarrow \mathrm{OK}$.*

Note that type safety requires that execution does not raise linking related errors only when started from classes that were the product of the compilation. An error raised by an execution started from a class c present in the original binary context cc_b can be either an error whic was already present (that is, $cc_b \vdash c \rightsquigarrow \mathrm{OK}$ does not hold), hence not due to compilation, or is due to the fact that some binary used by c has been modified. In this case we say that the compilation function does not satisfy *contextual binary compatibility* [1].

4 A safe type system for separate compilation

In this section, we define a type system (that we conjecture to be safe) which models separate compilation for a small Java subset .

The language we consider is shown in Fig. 1; metavariables `C`, `m`,`x` and `N` range over sets of class, method and parameter names, and integer literals, respectively. Both source and binary fragments are specified.

A source fragment `S` is a class declaration consisting of the class name, the name of the superclass and a set of method declarations. A method declaration consists of a method header and a method body (an expression). A method header consists of a (return) type, a method name and a sequence of parameter

[8] We also ignore for simplicity the fact that c should have a `main` method.

types and names. There are four kinds of expressions: instance creation, parameter name, integer literal and method invocation. A type is either a class name or `int`.

A binary fragment `B` consists of the name of the superclass, a set of annotated method headers and a set of type constraints `KS`. An annotated method header is a method header prefixed by an annotation indicating the class which contains the method declaration. A type constraint `K` is either a subtype constraint $\mathtt{C}_1 \leq \mathtt{C}_2$, or an implementation constraint $\mathtt{C} \lhd \mathtt{AMHS}$, stating that class `C` must provide annotated methods `AMHS`.

Note that here, for simplicity, binary fragments contain no code, but only some type information which can, however, easily retrieved from a regular Java `.class` file.

$$
\begin{array}{rll}
\mathtt{S} & ::= & \mathtt{class\ C\ extends\ C'\ \{\ MDS\ \}} \\
\mathtt{MDS} & ::= & \mathtt{MD}_1 \ldots\ \mathtt{MD}_n \qquad (n \geq 0) \\
\mathtt{MD} & ::= & \mathtt{MH\ \{\ return\ E;\ \}} \\
\mathtt{MH} & ::= & \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n) \qquad (n \geq 0) \\
\mathtt{E} & ::= & \mathtt{new\ C} \mid \mathtt{x} \mid \mathtt{N} \\
 & & \mathtt{E}_0.\mathtt{m}(\mathtt{E}_1, \ldots, \mathtt{E}_n) \qquad (n \geq 0) \\
\mathtt{T} & ::= & \mathtt{C} \mid \mathtt{int} \\
 & & \\
\mathtt{B} & ::= & \langle \mathtt{C}, \mathtt{AMHS}, \mathtt{KS} \rangle \\
\mathtt{KS} & ::= & \mathtt{K}_1 \ldots\ \mathtt{K}_n \qquad (n \geq 0) \\
\mathtt{K} & ::= & \mathtt{C}_1 \leq \mathtt{C}_2 \mid \mathtt{C} \lhd \mathtt{AMHS} \\
\mathtt{AMHS} & ::= & \mathtt{C}_1\ \mathtt{T}_1\ \mathtt{m}(\bar{\mathtt{T}}_1) \ldots \mathtt{C}_n\ \mathtt{T}_n\ \mathtt{m}(\bar{\mathtt{T}}_n) \qquad (n \geq 0) \\
\bar{\mathtt{T}} & ::= & \mathtt{T}_1 \ldots \mathtt{T}_n \qquad (n \geq 0)
\end{array}
$$

Fig. 1. Syntax and types

The top-level rules of the type system are defined in Fig.2.
The main judgment $cc \vdash \mathtt{CS} \rightsquigarrow cc_b$ is valid whenever the compilation invoked on the class names in `CS` in compilation context cc successfully produces the binary context cc_b.

The compilation can be split in two distinct phases; first, all classes in `CS` (and, implicitly, all classes which classes in `CS` depends on) are type-checked (hypotheses), then binary fragments are produced for all the type-checked classes which were not yet in binary form (conclusion).

The side condition $\mathtt{CS} \subseteq \mathit{Def}(cc_s)$ ensures that all classes in `CS` have a source fragment in cc; if not so, compilation fails, otherwise classes in `CS` are sequentially type-checked (hypotheses).

Judgment $cc; \Gamma \vdash \mathtt{C} \rightsquigarrow \Gamma'$ is valid whenever class `C` is well-typed w.r.t. compilation context cc and class environment Γ; Γ' is the new class environment produced during the type-checking of `C`. A class environment is a finite map associating with each class name `C` a pair $\langle \mathtt{C}', \mathtt{AMHS} \rangle$, where $\mathtt{C}'$ denotes the superclass of `C`, while `AMHS` is the set of all annotated method headers (either inherited

$$\frac{\langle cc_b, cc_s\rangle; \Gamma_0 \vdash \mathtt{C}_1 \rightsquigarrow \Gamma_1 \quad \ldots \quad \langle cc_b, cc_s\rangle; \Gamma_{n-1} \vdash \mathtt{C}_n \rightsquigarrow \Gamma_n}{\langle cc_b, cc_s\rangle \vdash \mathtt{CS} \rightsquigarrow cc'_b}$$

$$\mathtt{CS} = \{\mathtt{C}_1, \ldots, \mathtt{C}_n\} \subseteq \mathit{Def}(cc_s)$$
$$\mathit{Def}(\Gamma_0) = \{\mathtt{Object}\}, \Gamma_0(\mathtt{Object}) = \langle \bot, \emptyset \rangle$$
$$\mathit{Def}(cc'_b) = \mathit{Def}(\Gamma_n) \setminus \mathit{Def}(cc_b)$$
$$\forall\, \mathtt{C} \in \mathit{Def}(cc'_b)\ \ cc'_b(\mathtt{C}) = \mathit{bin}(cc_s, \Gamma_n, \mathtt{C})$$

$$\frac{}{cc; \Gamma \vdash \mathtt{int} \rightsquigarrow \Gamma} \qquad \frac{}{cc; \Gamma \vdash \mathtt{C} \rightsquigarrow \Gamma} \quad \mathtt{C} \in \mathit{Def}(\Gamma)$$

$$\frac{\langle cc_b, cc_s\rangle; \Gamma \vdash \mathtt{C}_1 \rightsquigarrow \Gamma_1 \quad \langle cc_b, cc_s\rangle; \Gamma_1[\mathtt{C} \mapsto \langle \mathtt{C}_1, \mathtt{AMHS}\rangle] \vdash \mathtt{KS} \rightsquigarrow \Gamma_2}{\langle cc_b, cc_s\rangle; \Gamma \vdash \mathtt{C} \rightsquigarrow \Gamma_2}$$

$$\mathtt{C} \notin \mathit{Def}(\Gamma)$$
$$cc_b(\mathtt{C}) = \langle \mathtt{C}_1, \mathtt{AMHS}', \mathtt{KS}\rangle$$
$$\Gamma_1(\mathtt{C}_1) = \langle \_, \mathtt{AMHS}_1\rangle$$
$$\mathtt{AMHS}_1[\mathtt{AMHS}'] = \mathtt{AMHS}$$

$$\frac{\langle cc_b, cc_s\rangle; \Gamma \vdash \mathtt{C}_1 \rightsquigarrow \Gamma_1 \quad \langle cc_b, cc_s\rangle; \Gamma_1[\mathtt{C} \mapsto \langle \mathtt{C}_1, \mathtt{AMHS}\rangle] \vdash \mathtt{MDS} \rightsquigarrow \Gamma_2}{\langle cc_b, cc_s\rangle; \Gamma \vdash \mathtt{C} \rightsquigarrow \Gamma_2}$$

$$\mathtt{C} \notin \mathit{Def}(\Gamma) \cup \mathit{Def}(cc_b)$$
$$cc_s(\mathtt{C}) = \mathtt{class\ C\ extends\ C}_1\ \{\ \mathtt{MDS}\ \}$$
$$\mathtt{AMHS}' = \mathit{Amhs}(\mathtt{C}, \mathtt{MDS})$$
$$\Gamma_1(\mathtt{C}_1) = \langle \_, \mathtt{AMHS}_1\rangle$$
$$\mathtt{AMHS}_1[\mathtt{AMHS}'] = \mathtt{AMHS}$$

Fig. 2. Top-level rules

or declared) of C. Class environments model the needed type information about classes collected by the compiler while inspecting source and binary fragments in the compilation context.

The initial class environment Γ_0 (see the corresponding side condition) contains only the predefined empty class (with no superclass) Object[9]. Class environment Γ_1, produced while type-checking $\mathtt{C}_1$, contains (besides Γ_0) type information about all classes needed for type-checking $\mathtt{C}_1$: all superclasses of $\mathtt{C}_1$ and all classes used (both directly and indirectly) by $\mathtt{C}_1$.

The new class environment Γ_1 is used for checking next class $\mathtt{C}_2$ and so on, until producing an environment Γ_n containing which have been type-checked; from this set we can easily retrieve the set of all classes which need to be compiled (see the side condition defining cc'_b).

The remaining rules specify type-checking of primitive types and classes. Type-checking of primitive types and or classes already collected in the class environment is trivial.

The other two rules concern classes which have not been inspected yet (the former deals with binary fragments, whereas the latter with source fragments). They are almost symmetric, except that when both binary and source fragment are present, priority is given to the former. First, the direct parent class $\mathtt{C}_1$ is type-checked; then, from the annotated method headers of $\mathtt{C}_1$ and those declared in C, the annotated method headers of C are derived (and rules on overriding are checked). Finally, either the set of type constraints (in the binary case) or the set of method declarations (in the source case) of C is type-checked.

[9] For simplicity, we ignore all the predefined methods of Object.

For lack of space, all other rules and auxiliary functions are defined in the Appendix.

Finally we show how the second example discussed in Sect.2 can be modeled in the framework defined above.

The compilation context $cc_0 = \langle cc_b^0, cc_s^0 \rangle$ is defined by $cc_b^0 = \emptyset$, $cc_s^0 = \{\mathtt{A} \mapsto \mathtt{S_A}, \mathtt{B} \mapsto \mathtt{S_B}, \mathtt{C} \mapsto \mathtt{S_C}\}$, where $\mathtt{S_A}$, $\mathtt{S_B}$, and $\mathtt{S_C}$ are the source code of `A`, `B`, and `C` as defined in the example[10].

The compilation context $cc_1 = \langle cc_b^1, cc_s^0 \rangle$ (corresponding to the context after invoking the compiler on `A`) is obtained by updating the previous binary context cc_b^0 with the binary context cc_b derived from the judgment $cc_0 \vdash \{\mathtt{A}\} \leadsto cc_b$. Since in this case cc_b^0 is empty, we have $cc_b^1 = cc_b = \{\mathtt{A} \mapsto \mathtt{B_A}, \mathtt{B} \mapsto \mathtt{B_B}, \mathtt{C} \mapsto \mathtt{B_C}\}$, where

$$\mathtt{B_A} = \langle \mathtt{Object}, \{\mathtt{A\ int\ main()}\}, \{\mathtt{B} \leq \mathtt{B}, \mathtt{B} \lhd \{\mathtt{B\ int\ m1()}\}\} \rangle$$
$$\mathtt{B_B} = \langle \mathtt{Object}, \{\mathtt{B\ int\ m1()}\}, \{\mathtt{C} \leq \mathtt{C}, \mathtt{C} \lhd \{\mathtt{C\ int\ m1()}\}\} \rangle$$
$$\mathtt{B_C} = \langle \mathtt{Object}, \{\mathtt{C\ int\ m1()}\}, \emptyset \rangle$$

The subtype constraints $\mathtt{B} \leq \mathtt{B}$ and $\mathtt{C} \leq \mathtt{C}$ simply require the existence of class `B` and `C`, respectively, otherwise no constructor could be correctly invoked on them.

The context cc_2' is obtained from cc_1 by changing the source code of `C` (according to the example), therefore $cc_2' = \langle cc_b^1, cc_s^2 \rangle$, where $cc_s^2 = cc_s^0[\mathtt{S'_C}/\mathtt{C}]$ is obtained from cc_s^0 by updating `C` with the new source $\mathtt{S'_C}$.

Finally, `A` cannot be successfully compiled in context cc_2, since there is no cc_b s.t. the judgment $cc_2 \vdash \{\mathtt{A}\} \leadsto cc_b$ is valid.

5 Conclusion

We have shown that typing rules for Java separate compilation can be quite complex and cannot easily explained informally. Moreover, they can be be unsafe, as happens for SDK and Jikes compilers since they perform very few checks on binary fragments delegating them to the JVM. We argue that a more robust compiler implementation should perform as much checks as possible at compile time, delegating to the JVM only those checks that can only be performed at run time.

We have introduced a simple framework which allows to formally model separate compilation and the related properties. Within this framework, we have defined, for a small subset of Java, a type system for separate compilation which we conjecture to be type safe.

In this paper, for lack of space, we have focused on the safety property; however, there are other interesting properties one can express for separate compilation, like *contextual binary compatibility* (mentioned at the end of Sect.3) and *monotonicity*, that is, the fact that when a subset of the source fragments composing a program is changed, re-compiling only this set gives the same result as

[10] Where, however, `static` has been removed and `void` replaced with `int`.

re-compiling the whole program (this property is mentioned as desirable in [3] and formalized in [1]).

The work presented in this paper is a first step towards the formal definition and comparison of different type systems for Java separate compilation, corresponding, e.g., either to standard Java compilers, or to extended compilers which perform additional checks. A lot of work still has to be done. On the theoretical side, we plan to define a complete execution and linking model for the toy language defined in this paper, including a toy bytecode, thus allowing to formally prove type safety. We also want to study the formal relations between the type safety property analyzed in this paper and other properties like monotonicity and contextual binary compatibility [1]. On the practical side, we plan to extend the safe type system defined here to more relevant Java subsets and to develop extended compilers which satisfy good properties like type safety.

Acknowledgments: We warmly thank Sophia Drossopoulou for her precious contribution to stimulate and enhance this work.

References

1. D. Ancona, G. Lagorio, and E. Zucca. Monotone separate compilation in Java. Technical Report, DISI. Submitted for publication, April 2001.
2. G. Bracha, J. Gosling, B. Joy, and G. Steele. *The Java™ Language Specification, Second Edition*. Addison-Wesley, 2000.
3. L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, January 1997.
4. 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*, number 1523 in Lecture Notes in Computer Science, pages 41–82. Springer Verlag, Berlin, 1999.
5. D. Syme. Proving Java type sound. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 83–118. Springer Verlag, 1999.
6. D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 119–156. Springer Verlag, 1999.

A Appendix

Class environments

$$\begin{aligned} \Gamma &::= \mathtt{C}_1{:}\langle \mathtt{C}_1^{\perp}, \mathtt{AMHS}_1\rangle, \ldots, \mathtt{C}_n{:}\langle \mathtt{C}_n^{\perp}, \mathtt{AMHS}_n\rangle \ (n \geq 0) \\ \mathtt{C}^{\perp} &::= \perp \mid \mathtt{C} \end{aligned}$$

Binary class generation

$$bin(cc_s, \Gamma, \mathtt{C}) = \langle \mathtt{C}_1, \mathtt{AMHS}, \mathtt{KS} \rangle \qquad \text{if } cc_s(\mathtt{C}) = \mathtt{class\ C\ extends\ C}_1\ \{\ \mathtt{MDS}\ \}\ \ \mathtt{AMHS} = Amhs(\mathtt{MDS})\ \ \Gamma \vdash \mathtt{MDS} \leadsto \mathtt{KS}$$

Annotated methods update A set of method headers AMHS is well-formed if it does not contain overloaded methods.

$$\mathtt{AMHS'}[\mathtt{AMHS}] = \begin{cases} \mathtt{AMHS} \to \mathtt{AMHS'} & \text{if } \mathtt{AMHS} \to \mathtt{AMHS'} \text{ is well-formed} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{where } \mathtt{AMHS} \to \mathtt{AMHS'} = \mathtt{AMHS} \cup \{\mathtt{C\ T\ m}(\bar{\mathtt{T}}) \mid \nexists\, \mathtt{C}_1\ s.t.\ \mathtt{C}_1\ \mathtt{T\ m}(\bar{\mathtt{T}}) \in \mathtt{AMHS}\}$$

Annotation and extraction of method headers

$$Amhs(\mathtt{C}, \mathtt{MDS}) = annotate(\mathtt{C}, Mhs(\mathtt{MDS}))$$

$$annotate(\mathtt{C}, \mathtt{MH}_1\ \ldots\ \mathtt{MH}_n) = \mathtt{C\ MH}_1\ \ldots\ \mathtt{C\ MH}_n$$

$$Mhs(\mathtt{MH}_1\{\ \mathtt{return\ E}_1;\ \}\ldots\ \mathtt{MH}_n\{\ \mathtt{return\ E}_n;\ \}) = \mathtt{MH}_1 \ldots \mathtt{MH}_n$$

Method resolution

$$RetType(\Gamma, \mathtt{C}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n) = \mathtt{T'} \text{ if } \begin{cases} \Gamma(\mathtt{C}) = \mathtt{AMHS} \\ \mathtt{C}_1\ \mathtt{T'}\ \mathtt{m}(\mathtt{T}'_1\ \mathtt{x}_1, \ldots, \mathtt{T}'_n\ \mathtt{x}'_n) \in \mathtt{AMHS} \\ \Gamma \vdash \mathtt{T}_i \leq \mathtt{T}'_i \text{ for } i = 1..n \end{cases}$$

$$\frac{cc; \Gamma_0 \vdash \mathtt{K}_1 \leadsto \Gamma_1 \ldots cc; \Gamma_{n-1} \vdash \mathtt{K}_n \leadsto \Gamma_n}{cc; \Gamma_0 \vdash \mathtt{K}_1\ \ldots\ \mathtt{K}_n \leadsto \Gamma_n}$$

$$\frac{cc; \Gamma \vdash \mathtt{C}_1 \leadsto \Gamma_1 \quad cc; \Gamma \vdash \mathtt{C}_2 \leadsto \Gamma_2 \quad \Gamma_2 \vdash \mathtt{C}_1 \leq \mathtt{C}_2}{cc; \Gamma \vdash \mathtt{C}_1 \leq \mathtt{C}_2 \leadsto \Gamma_2}$$

$$\frac{cc; \Gamma \vdash \mathtt{C} \leadsto \Gamma_1 \quad \Gamma_1 \vdash \mathtt{AMHS}_1 \lhd \mathtt{AMHS}}{cc; \Gamma \vdash \mathtt{C} \lhd \mathtt{AMHS} \leadsto \Gamma_1} \qquad \Gamma_1(\mathtt{C}) = \mathtt{AMHS}_1$$

Fig. 3. Type-checking sets of constraints

$$\frac{\Gamma \vdash \mathtt{MD}_1 \rightsquigarrow \mathtt{KS}_1 \ldots \Gamma \vdash \mathtt{MD}_n \rightsquigarrow \mathtt{KS}_n}{\Gamma \vdash \mathtt{MD}_1 \ \ldots \ \mathtt{MD}_n \rightsquigarrow \mathtt{KS}_1 \ \ldots \ \mathtt{KS}_n}$$

$$\frac{\Gamma; \{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash \mathtt{E} : \mathtt{T} \rightsquigarrow \mathtt{KS}}{\Gamma \vdash \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return\ E};\ \} \rightsquigarrow \mathtt{KS}\ \mathtt{T}_0 \leq \mathtt{T}_0 \ldots \mathtt{T}_n \leq \mathtt{T}_n\ \mathtt{T} \leq \mathtt{T}_0}$$

$$\frac{}{\Gamma; \Pi \vdash \mathtt{new\ C} : \mathtt{C} \rightsquigarrow \mathtt{C} \leq \mathtt{C}} \qquad \frac{}{\Gamma; \Pi \vdash \mathtt{N} : \mathtt{int} \rightsquigarrow \Lambda} \qquad \frac{}{\Gamma; \Pi \vdash \mathtt{x} : \mathtt{T} \rightsquigarrow \Lambda} \quad \Pi(\mathtt{x}) = \mathtt{T}$$

$$\frac{\begin{array}{l}\Gamma; \Pi \vdash \mathtt{E}_0 : \mathtt{C} \rightsquigarrow \mathtt{KS}_0 \\ \Gamma; \Pi \vdash \mathtt{E}_1 : \mathtt{T}_1 \rightsquigarrow \mathtt{KS}_1 \\ \ldots \\ \Gamma_{n-1}; \Pi \vdash \mathtt{E}_n : \mathtt{T}_n \rightsquigarrow \mathtt{KS}_n\end{array}}{\Gamma; \Pi \vdash \mathtt{E}_0.m(\mathtt{E}_1, \ldots, \mathtt{E}_n) : \mathtt{T} \rightsquigarrow \mathtt{KS}_0 \ldots \mathtt{KS}_n\ \mathtt{C} \lhd \{\mathtt{C}_1\ \mathtt{T}\ \mathtt{m}(\mathtt{T}_1 \ldots \mathtt{T}_n)\}} \quad \Gamma(\mathtt{C}) = \mathtt{AMHS}_1\ \mathtt{C}_1\ \mathtt{T}\ \mathtt{m}(\mathtt{T}_1 \ldots \mathtt{T}_n)\ \mathtt{AMHS}_2$$

Fig. 4. Code generation

$$\frac{cc; \Gamma_0 \vdash \mathtt{MD}_1 \rightsquigarrow \Gamma_1 \ldots cc; \Gamma_{n-1} \vdash \mathtt{MD}_n \rightsquigarrow \Gamma_n}{cc; \Gamma_0 \vdash \mathtt{MD}_1 \ \ldots \ \mathtt{MD}_n \rightsquigarrow \Gamma_n}$$

$$\frac{\begin{array}{l}cc; \Gamma \vdash \mathtt{T}_0 \rightsquigarrow \Gamma_0 \ldots cc; \Gamma \vdash \mathtt{T}_n \rightsquigarrow \Gamma_n \\ cc; \Gamma_n; \{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash \mathtt{E} : \mathtt{T} \rightsquigarrow \Gamma' \\ \Gamma' \vdash \mathtt{T} \leq \mathtt{T}_0\end{array}}{cc; \Gamma \vdash \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return\ E};\ \} \rightsquigarrow \Gamma'}$$

$$\frac{cc; \Gamma \vdash \mathtt{C} \rightsquigarrow \Gamma'}{cc; \Gamma; \Pi \vdash \mathtt{new\ C} : \mathtt{C} \rightsquigarrow \Gamma'} \qquad \frac{}{cc; \Gamma; \Pi \vdash \mathtt{N} : \mathtt{int} \rightsquigarrow \Gamma}$$

$$\frac{}{cc; \Gamma; \Pi \vdash \mathtt{x} : \mathtt{T} \rightsquigarrow \Gamma} \quad \Pi(\mathtt{x}) = \mathtt{T}$$

$$\frac{\begin{array}{l}cc; \Gamma; \Pi \vdash \mathtt{E}_0 : \mathtt{C} \rightsquigarrow \Gamma_0 \\ cc; \Gamma_0; \Pi \vdash \mathtt{E}_1 : \mathtt{T}_1 \rightsquigarrow \Gamma_1 \\ \ldots \\ cc; \Gamma_{n-1}; \Pi \vdash \mathtt{E}_n : \mathtt{T}_n \rightsquigarrow \Gamma_n\end{array}}{cc; \Gamma; \Pi \vdash \mathtt{E}_0.m(\mathtt{E}_1, \ldots, \mathtt{E}_n) : \mathtt{T} \rightsquigarrow \Gamma_n} \quad \mathit{RetType}(\Gamma, \mathtt{C}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n) = \mathtt{T}$$

Fig. 5. Type-checking of source class bodies

$$\frac{\Gamma \vdash \mathtt{C}'_1 \leq \mathtt{C}_1 \ldots \Gamma \vdash \mathtt{C}'_n \leq \mathtt{C}_n}{\Gamma \vdash \{\mathtt{C}_1\ \mathtt{T}_1\ \mathtt{m}_1(\bar{\mathtt{T}}_1), \ldots, \mathtt{C}_k\ \mathtt{T}_k\ \mathtt{m}_k(\bar{\mathtt{T}}_k)\} \lhd \{\mathtt{C}'_1\ \mathtt{T}_1\ \mathtt{m}_1(\bar{\mathtt{T}}_1), \ldots, \mathtt{C}'_n\ \mathtt{T}_n\ \mathtt{m}_n(\bar{\mathtt{T}}_n)\}} \quad n \leq k$$

$$\frac{}{\Gamma \vdash \mathtt{int} \leq \mathtt{int}} \qquad \frac{}{\Gamma \vdash \mathtt{C} \leq \mathtt{C}} \quad \mathtt{C} \in \mathit{Def}(\Gamma)$$

$$\frac{}{\Gamma \vdash \mathtt{C} \leq \mathtt{C}'} \quad \Gamma(\mathtt{C}) = \langle \mathtt{C}', \_ \rangle \qquad \frac{\Gamma \vdash \mathtt{C} \leq \mathtt{C}' \quad \Gamma \vdash \mathtt{C}' \leq \mathtt{C}''}{\Gamma \vdash \mathtt{C} \leq \mathtt{C}'}$$

Fig. 6. Implementation and widening

From FGJ to Java according to LM translator

Mirko Viroli
DEIS – Università di Bologna
via Rasi e Spinelli 176
47023 Cesena (FC), Italy
mviroli@deis.unibo.it

ABSTRACT

In this paper we present a formalization of LM translator [11], a proposal for adding parametric polymorphism to Java by means of a logical extension of Generic Java [6]. LM translator overcomes the type-integration lacks of Generic Java thanks to several distinctive features: run-time information on parametric types is carried into type descriptors created at load-time, reflective features of Java are exploited for dealing with legacy classes, hashtables are used to tackle performance issues, and special method descriptors tables support dynamic dispatching of method calls. Because of all these tricks LM translator turns out to be a complex system, whose description and understanding are often quite complicated, whereas translation approaches were typically used also because of their simplicity.

So, the declared goal of the formalization is to help reasoning about LM, reaching a good trade-off between compactness of its description and completeness in treating its features. Technically, this is done by modeling LM translator as a compilation of FGJ [3], a core calculus for Generic Java, into full Java. This choice for the source and target languages allows to directly focus on the key issues related to parametric polymorphism, thanks to the minimality of FGJ, and not to constrain in any way the power and features of LM translator, as the target language is full Java. The formalization obtained is satisfactorily compact; the only significant issues left out are the management of legacy classes and the internal details of the library classes supporting the translation.

In this paper we also argue that this kind of formalization is the most suitable support for the development of an actual prototype of the translator. This is motivated by outlining our current work on this direction, in which the formalization here introduced plays a crucial role. The key property of our methodology is the separation between the peculiar aspects of the translation and the details not strictly related to parametric polymorphism. This is meant to speed up the tuning of the translation towards an optimized implementation for the Java programming language.

1. INTRODUCTION AND MOTIVATION

In the context of the proposals for adding generics to the Java programming language as a response to Sun's call [1], Generic Java (GJ) [6] is going to be the basis for the first actual release of Java providing this extension [8]. Its basic idea is to fully rely on the so-called *erasure technique*, that is translating a parametric class into the Java class one would have written without having parametric polymorphism and exploiting the homogeneous generic idiom [5]. All the information on the type parameters of a given parametric class or method is completely lost in the translated code, as parametric classes and methods are translated into their monomorphic version, e.g. `List<String>` is translated to `List`, and each type variable is translated to its bound (`Object` by default). This technique allows GJ to enjoy full upward compatibility properties and also avoids almost any translation overhead [6].

However, GJ has also a well recognized limitation. Due to type erasure, those operations involving run-time inspection of the type of an object, basically type-casts and instance tests (Java operator `instanceof`), cannot be supported for parametric types. Being unable to do this may be a serious constraint. On the one hand, even though one adds parametric polymorphism to the language, the need for using heterogeneous collections of elements remains, therefore, the problem of accessing objects previously up-casted still exists. Type-casts and instance tests are the only way to safely recover the proper type of such objects. On the other hand, these type-dependent operations are the basis for supporting successful Java mechanisms related to persistence such as Java Serialization, Java Remote Method Invocation and JavaBeans. Adding parametric polymorphism to the language but then disabling its integration with these important mechanisms clearly leads to a somewhat incomplete extension. As discussed in [8], however, there is some lack of experience with constructs supporting generic types at run-time, and it is unclear whether the current proposals supporting this mechanism addresses performance and compatibility in the proper way. So, the choice was to rely on GJ anyway, at least until future studies may lead to solutions effectively supporting parametric types at run-time.

Currently, the proposal that seems the most reasonable starting point for this research is LM translator [11]. It extends the behaviour of GJ translator so as to make the code produced carrying the necessary information about the instantiation of the type parameters, into Java objects called *type descriptors* and *method descriptors*. These descriptors are created avoiding unnecessary space overhead and run-time overhead, and are exploited to implement those operations requiring run-time information on the parametric type of an object.

An implementation based on LM translator is likely to be a good compromise between the performance overhead introduced and the expressiveness power gained by the language. However, the description and the understanding of the effects of LM translator on the code are not as simple as for GJ. In fact, LM translator puts together a lot of heterogeneous ingredients. First of all, the type and method descriptors supporting generics at run-time are entirely created at load-time, and Java Reflection is used to integrate this management with legacy classes. Then, special data structures ex-

ploiting hashtables are automatically built by the translator in order to reduce the performance overhead of accessing descriptors. Finally, dynamic dispatching of method calls is supported through the simulation of the management of Virtual Method Tables, exploiting structures of method descriptors called Virtual Parametric Methods Tables [10].

In this situation, a formalization would be a necessary tool for harnessing this complexity. It may allow to focus on the relevant issues abstracting away from unimportant ones, and to describe in a comprehensive way the important ideas so as to avoid risks of incomplete specifications. Among all these aspects, the formalization is usually the key tool for leading to robust implementations.

In principle, we may follow the approach of [3], where the semantics of GJ is given in terms of a translation from a source core language, which is a subset of the language accepted by GJ, to a target core language, which is a subset of Java. In particular, these core languages are called, respectively, Featherweight Generic Java (FGJ) and Featherweight Java (FJ), and they include only those few language features directly involved in the translation of generics. This approach has a number of flavors: (i) it keeps the specification as compact as possible, (ii) it completely defines the features of interest discarding secondary ones, (iii) it allows to capture the core concepts of the translation, and (iv) it allows for proving important correctness results, such as type-preservation. However, all these properties hold together because the ideas behind the corresponding system are few and relatively simple.

For LM translator this would not be possible. The basic problem in fact is to isolate as target language a subset of Java having all the features needed to describe the effects of LM translator on the code. Such a language would not be as small as one would like it to be. Suppose we want to proceed by using as target language an extension of FJ. We should add to FJ (i) reflection, in order to deal with legacy Java classes and to support the management of descriptors, (ii) side-effects and class loading, to model the fundamental idea that descriptors are not created each time by need, but once and for all at the application boot-strap, (iii) static fields of client classes, containing the descriptors they use, and (iv) static methods of parametric classes, storing important facilities for creating their descriptors. Then, it must be also considered that an important role is played by some library class of Java, such as arrays, vectors and hashtables. With such a complex target language it would be basically impossible to prove properties in a paper of reasonable space.

Clearly, one may think of dropping some of the features of LM trying to focus on just few issues, so as to rely on a smaller target language. For instance, we may discard the idea of load-time creation of descriptors, and dropping class-loading and side-effects from the target language. However, doing this will made our description of LM lacking one of the main reasons that motivated its introduction. In general, dropping relevant features simplifies the proof of properties but leads to incomplete specifications.

So, in this paper our goal is to define a compact formalization allowing for a complete description of the features of LM translation. This is done by means of a translation from the source language FGJ (with some minimal extension), which is a subset of the language accepted by LM, to a large language, which is Java itself. This choice will turn out to be the best one to our end, as the complexity of LM translator is mostly harnessed in a couple of pages of rules.

Then, we describe our development of an actual implementation of LM translator, highlighting that the formalization here introduced plays a crucial role in supporting it. Basically, it allows to directly write a core prototype of the system containing all the relevant features of LM translator. This prototype can be gradually turned into the final implementation by just adding the programming constructs left out by FGJ, without the need of dealing with further issues strictly related to parametric polymorphism. Our goal is not to produce a fully-featured implementation for Java, but to have a tool allowing fast prototyping and tuning of LM.

The remainder of the paper is as follows. Section 2 overviews LM translator, and Section 3 depicts FGJ focusing on the notation and on the functions introduced in [3] which will be helpful to our formalization. Section 4 provides the actual compilation of FGJ into Java, and Section 5 a comprehensive example, which helps understanding the key concepts of the formalization. Section 6 discusses how we meant to fill the gap between this formalization and the actual implementation of the translator.

2. LM TRANSLATOR

In the code produced by LM translator [11], each class that needs to perform some operation on parametric types, such as type-casts, instance tests, and object allocations, will handle type descriptors for these types, which are objects of the library class `$TD`. In particular, these type descriptors are stored into static fields created by the translator, and are initialized at load-time. When allocating an object from a parametric type (`new` expression), the corresponding descriptor is passed as first argument in the constructor, and will then be used to make that object keeping information on its type. Then, these descriptors are exploited to implement type-casts and instance tests, using the methods `cast` and `isInstance` of the class of descriptors `$TD`. See the signature of class `$TD` in Figure 1.

Similar management is supported for parametric methods, with analogous method descriptors of class `$MD` which are passed at invocation-time. However, here the problem is complicated due to the need of dealing with dynamic dispatching. In [10] we proposed a solution exploiting a data structure called Virtual Parametric Method Table (VPMT). Each descriptor carries its own VPMT, which analogously to a Virtual Methods Table (VMT) [4], allows the proper method descriptor to be bound to the body at invocation-time. The VPMT is an array with one entry for each virtual method (i.e., a public or protected method of Java); each entry contains a vector of method descriptors. The position of a given instantiation of a parametric method is invariant through VPMTs of different subclasses, so by just using this position the actual receiver of the invocation can access the right descriptor [10].

Another key aspect of LM translator is that descriptors are kept by descriptors managers living at run-time, respectively in the class `$TDM` (Type Descriptors Manager) and `$MDM` (Method Descriptors Manager). Their basic goal is to prevent descriptors from being created twice. This is obtained by registering descriptors in their manager each time they are created. Finally, LM translator also relies on a special technique for treating those parametric types that exploit type variables of the scope within their parameters [11]. As the actual instantiation of these types is unknown until the type variables get actually instantiated, we let each type descriptor and method descriptor carrying those type/method descriptors using its parameters, called friend types/methods. Then, when we register a

```
public class $TD {

  public Class c;
  public $TD[] params;
  public $TD[] friendTs;
  public int[] friendMs;
  public $TD father;
  public Vector[] VPMT;

  public $TD(Class c, $TD[] p) { ... }

  public boolean checkNew() { ... }

  boolean isInstance(Object o){
    return (o instanceof $Parametric)
      ?(($Parametric)o).getTD().isSubType(this)
      :c.isInstance(o);
  }
  Object cast(Object o){
    if (isInstance(o)) return o;
    throw new ClassCastException(...);
  }
}
```

```
public class $MD {

  public int mID;
  public int lPos;
  public $TD tR;
  public $TD[] params;
  public $TD[] friendTs;
  public int[] friendMs;

  public $MD($TD t, int mID, $TD[] p) { ... }
  public boolean checkNew() { ... }
}

public class $TDM{
  public static $TD register(Class c) { ... }
  public static $TD register(Class c,$TD[] p) { ... }
  public static void initVPMT($TD t){ ... }
}

public class $MDM {
  public static $MD register($TD tR,int mID,$TD[] p) { ... }
  public static void propagate($TD tR,int mID,$TD[] p){ ... }
}
```

Figure 1: Some detail on the structure of the library classes

descriptor we are able to automatically register its friend descriptors. This is the main task accomplished by the methods `createTD` and `createMD`, built by the translator in each parametric class. See some detail on the implementation of the library classes supporting the translation in Figure 1.

In general, we will assume that the source language for LM translator is the same as that of GJ, even though at this point of our research we haven't faced yet the implementation of issues such as inner parametric classes, parametric exceptions, and so on, which are all managed by GJ. However, we are confident they can be implemented in a satisfactory way in our framework.

Some of the core ideas of LM translator have been borrowed from NextGen, a code-expansion technique for translating generics in Java proposed in [2]. There, type-passing style is exploited for implementing parametric methods, as objects called *snippet* environments are passed at invocation-time in an analogous way of our method descriptors. These environments are created at load-time as well, leading to very small run-time overhead. However, the code-expansion technique leads to high memory and code footprint, so the approaches exploiting this technique will not be used for the actual implementation of the Java programming language, as motivated in [8].

The general idea of LM translator seems not to be strictly related to Java, but of a somewhat general appealing. In fact, the proposal for extending Microsoft's .NET Common Language Runtime with generics shown in [9] uses a similar technique. The main difference is that in this proposal the necessary information on the parametric types is not built eagerly, at load-time, but only once on a by-need basis. This style can be used in LM as well. Basically it implies that type descriptors are not created in the initialization code of the client classes, but the first time they are accessed. We are currently working on this variation of LM, which we believe may lead to a better implementation.

3. FEATHERWEIGHT GENERIC JAVA

The source language for our formalization is Featherweight Generic Java (FGJ) [3], a core-calculus for GJ focusing on parametric classes, parametric methods and fields, and in which expressions can be only type-casts, object allocations, field accessing and method invocations. Language constructs remaining out from FGJ, and from the formalization we are going to introduce as well, are interfaces, inner classes and arrays.

To the end of introducing the syntax of FGJ, we let the metavariables `C` and `D` range over class names; `S`, `T`, and `U` range over types; `X` and `Y` range over type variables; `N` and `P` range over non-variable types; `CL` ranges over class declarations, `K` over constructors, `M` over method declarations, `f` over field names, `m` over method names, `x` over variables, and `e` over expressions. We denote by $\overline{\mathtt{a}}$ a list of elements $\mathtt{a}_1, \mathtt{a}_2, ..$, and the empty list by ϵ. Then, we abuse the notation of function application, as a function $f(\mathtt{a}) = \mathtt{b}$ can be used in $f(\overline{\mathtt{a}}) = \overline{\mathtt{b}}$ to denote the application on all the elements of the list $\overline{\mathtt{a}}$, returning the list of results $\overline{\mathtt{b}}$. Substituting the sub-term `a` by `b` into the term `c` is denoted by $[\mathtt{b}/\mathtt{a}]\mathtt{c}$, while $[\overline{\mathtt{a}} \mapsto \overline{\mathtt{b}}]f$ is the function mapping $\overline{\mathtt{a}}$ to $\overline{\mathtt{b}}$ and any other element `c` to $f(\mathtt{c})$. The grammar of FGJ is the following:

```
CL  ::=  class C<X ◁ N> ◁ N{T f; K M}
K   ::=  C(T f){super(f); this.f=f;}
M   ::=  <X ◁ N> T m (T x){ ↑ e;}
e   ::=  x | e.f | e.m<T>(e)
         | new N(e) | (T)e
T   ::=  X | N
N   ::=  C<T>
```

The environments Δ and Γ are used to model the type system of FGJ. $\Delta = \overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}$ maps type variables into their bounds, which are non-variable types. $\Gamma = \overline{\mathtt{y}} : \overline{\mathtt{T}}$ maps variables into types, that is formal parameters to their declared type. Then, we have the following judgements (see [3] for their actual semantics):

- The subtyping judgement $\Delta \vdash \mathtt{T} <: \mathtt{U}$, stating that the type T is a subtype of U under the environment Δ.
- Expression typing judgement $\Delta; \Gamma \vdash \mathtt{e} \in \mathtt{T}$, that under a Δ environment and a Γ environment, gives the type T to the expression e.
- Class typing judgement CL OK, stating whether a class definition is well-formed and well-typed.

We suppose the existence of a class table CT, taking class names and returning class definitions. Among other usual properties, we suppose that the root type Object is in the dominion of CT, and that all the class definitions are OK.

Our version of FGJ is actually slightly different from the one presented in [3]. In order to stress the fact that LM translator allows for implementing type-operations without any limitation we changed the syntax of type-casts, allowing to cast an object not only to a non-variable type N as in [3], but to a parametric type T in general. The corresponding change on the type system and on the relative proofs should be minimal. In particular, now we don't need any rule for checking valid down-casts as in [3], as all type-casts are compilable by LM translation. To the end of defining our translation we need the following look-up functions defined in [3]:

- *mtypemax*$(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{D}} \mapsto \mathtt{D}$, returning the erased argument types $\overline{\mathtt{D}}$ and return type D of the method m in C. In particular, this is done by finding the method in the highest superclass in which it is defined.
- *fieldsmax*$(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$, returning couples - erased type, field name - of the class C, finding field types in the highest superclass in which they are defined.

Looking for members in the highest class and comparing the result with standard lookup functions permits to determine whether some type variable has been instantiated due to an extend clause. In such a case in fact, the so-called *stupid casts* should be automatically inserted by the translator [3].

4. THE FORMALIZATION

This section presents the core of the paper, that is the compilation of FGJ into Java according to the translation schema of LM introduced in [11, 10]. As already mentioned our version of FGJ provides full type-cast ability. [11] highlights that translating type-casts is much the same than translating instance tests. So, the translation of type-casts is a mean for the translation of type-dependent operations in general.

This compilation abstracts away from the actual translation of non-parametric classes and methods. For instance, in an actual implementation the classes that do not have type parameters will not keep a local type descriptor, and methods without type parameters will not need to receive the method descriptor as first argument, and so on. In general the final translator should be able to translate pure Java sources to themselves. Then, here we also abstract from the details on the implementation of library classes $TD, $MD, $TDM and $MDM. An interested reader can refer to [11, 10].

The whole translation resembles the one shown in [3], both on notation and on semantics. The two basic differences are that LM translator provides special translation for operations involving parametric types and/or instantiating type parameters, and that the proper code to create descriptors and to keep track of them should be added to each class.

4.1 Auxiliary functions

The basic idea behind LM translator is to handle Java objects called type descriptors and methods descriptors, containing the necessary information to translate casts, object allocations and method invocations, included in the corresponding class or method, respectively. While a type descriptor is completely identified by the corresponding non variable type N, for method descriptors we use *method signatures* L, where $\mathtt{L} ::= \mathtt{T.m<}\overline{\mathtt{T}}\mathtt{>}$.

The information on what type descriptors and method descriptors have to be created is statically gathered at translation-time, and are modeled by means of the functions *getT* and *getM*. In particular, *getT* is used to get the parametric types used in casts, allocations and as receivers of method calls. This can be done either (i) from an expression e in the environments Δ and Γ (*getT*$_{\Delta,\Gamma}(\mathtt{e})$), (ii) from the body of a method M defined in class C (*getT*$(\mathtt{C}, \mathtt{M})$) or (iii) from all the expressions contained in class C (*getT*$(\mathtt{C})$). Analogously, *getM* gets the signature of the parametric methods invoked (i) from an expression e in the environments Δ and Γ (*getM*$_{\Delta,\Gamma}(\mathtt{e})$), (ii) from the body of a method M defined in class C (*getM*$(\mathtt{C}, \mathtt{M})$) and (iii) from all the expressions contained in a class C (*getM*$(\mathtt{C})$). Their semantics is shown in Figure 2. The operator $\bullet$ is meant to join two lists or an element and a list, discarding duplicates and preserving order. For instance we have $\mathtt{a,b,c} \bullet \mathtt{a,d,c,f} = \mathtt{a,b,c,d,f}$. Basically, the function $getT$ gathers the types used in casts, allocations and as receivers of method calls, while the function $getM$ gathers the signatures of the methods invoked.

We divide parametric types and method signatures into those having fully-instantiated type parameters, called *bound types* and *bound methods*, and those that instead contain some type variable, called *free types* and *free methods*, respectively. In the former case their descriptors are completely known, so they can be registered at the load-time of the classes which use them. In the latter case, instead, they are registered only when the descriptor of the enclosing class/method is registered, as only at this time the instantiation of the type parameters is known. We introduce the predicates *boundT*$_\Delta(\mathtt{N})$, *freeT*$_\Delta(\mathtt{N})$, *boundM*$_\Delta(\mathtt{L})$ and *freeM*$_\Delta(\mathtt{L})$ to check if under the environment Δ, specifying the type variables of the scope, the type N and the signature L are, respectively, bound or free. The notation for these predicates will be abused, so that when applying them to a list the result is the sublist of the elements satisfying the predicate. We have:

$$
\begin{array}{lcl}
\textit{freeT}_\Delta(\mathtt{N}) & \Leftrightarrow & \exists \mathtt{X} \in \textit{dom}(\Delta), \exists \mathtt{O} : \ [\mathtt{O/X}]\mathtt{N} \neq \mathtt{N} \\
\textit{freeM}_\Delta(\mathtt{L}) & \Leftrightarrow & \exists \mathtt{X} \in \textit{dom}(\Delta), \exists \mathtt{O} : \ [\mathtt{O/X}]\mathtt{L} \neq \mathtt{L} \\
\textit{boundT}_\Delta(\mathtt{N}) & \Leftrightarrow & \text{not} \quad \textit{freeT}_\Delta(\mathtt{L}) \\
\textit{boundM}_\Delta(\mathtt{L}) & \Leftrightarrow & \text{not} \quad \textit{freeM}_\Delta(\mathtt{L})
\end{array}
$$

We provide facilities for accessing the type descriptor of a given type and the method descriptor of a given method signature. We introduce two environments for types and signatures, denoted by symbols Θ and Π, respectively. $\Theta = \overline{\mathtt{T}} \mapsto \overline{\mathtt{e}}_J$ associates types to Java expressions representing the corresponding descriptor, $\Pi_\Theta = \overline{\mathtt{L}} \mapsto \overline{\mathtt{e}}_J$ binds method signatures to Java expressions representing the corresponding descriptors, the latter possibly exploiting the Θ environment to resolve some type descriptor. We access the de-

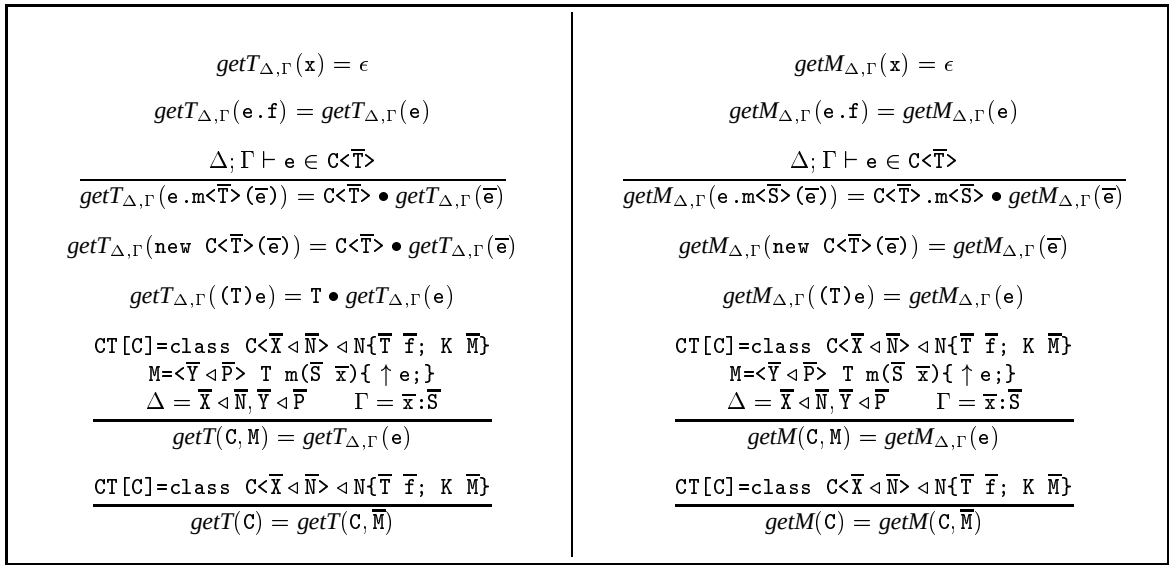


Figure 2: Gathering functions for type and method descriptors

scriptor for a type T by the notation $|\mathtt{T}|_\Theta$ and that of a method signature L by $|\mathtt{L}|_{\Pi,\Theta}$. Then, we introduce a standard Θ environment denoted by Θ_s, implementing the registration of bound types, defined as:

$$\Theta_s = \mathtt{C<\overline{T}>} \mapsto \mathtt{C.createTD(new\ \$TD[]\{}|\overline{\mathtt{T}}|_{\Theta_s}\mathtt{\})}$$

that is, exploiting the static method `createTD` of the class, which accepts the array of type descriptors for the parameters. When a Θ environment has to deal with free types as well, we should add the specification on how to resolve type variables, and this can be done exploiting the environment $[\overline{\mathtt{X}} \mapsto \overline{\mathtt{e}}_J]\Theta$, according to our notation for function substitution. Then, we also introduce a standard Π environment Π_s as follows:

$$\frac{\mathtt{CT[C]=class\ C<\overline{X}\triangleleft\overline{N}>\triangleleft N\{\overline{T}\ \overline{f};\ K\ \overline{M}\}} \quad \mathtt{M_i=<\overline{Y}\triangleleft\overline{P}>\ T\ m(\overline{S}\ \overline{x})\{\uparrow e;\}} \quad \mathit{pos}(\mathtt{C},\mathtt{M}_i)=j}{\Pi_{s\,\Theta} = \mathtt{C<\overline{T}>.m<\overline{U}>} \mapsto \mathtt{C.createMD(}|\mathtt{C<\overline{T}>}|_\Theta, j, \mathtt{new\ \$TD[]\{}|\overline{\mathtt{U}}|_\Theta\mathtt{\})}}$$

The static method `createMD` is created by the translator, and accepts the type descriptor of the receiver, the position of `m` in `C`, and an array containing the descriptors for the parameters. The semantics of function $\mathit{pos}(\mathtt{C},\mathtt{M})$ is described in Figure 3. It returns the position of `M` in the VPMT of `C`. Basically, a method adds a new element to the VPMT only if it does not override a method in the super-class, otherwise its position is *inherited* from that method. The content of the methods `createTD` and `createMD` will be shown in the next sections.

4.2 Erasing types

The erasure of types is mostly the same as that of FGJ. We have the function returning the bound of a type as:

$$\mathit{bound}_\Delta(\mathtt{X}) := \Delta(\mathtt{X}) \qquad \mathit{bound}_\Delta(\mathtt{N}) := \mathtt{N}$$

and then the erasure function from FGJ types to Java classes:

$$|\mathtt{T}|_\Delta := \mathtt{C} \quad \text{if} \quad \mathit{bound}_\Delta(\mathtt{T}) = \mathtt{C<\overline{T}>},\quad \mathtt{T} \neq \mathtt{Object<>}$$

$$|\mathtt{Object<>}|_\Delta := \mathtt{LMObj}$$

In this formalization `Object` is translated into a special library class `LMObj`, as shown in top of Figure 3.

4.3 Translating Classes

For the rules defining translation of classes refer to Figure 3. FGJ classes are erased to Java classes: (i) by providing the proper translation of fields (erasing their types), methods ($|\mathtt{M}|_\mathtt{C}$) and constructor ($|\mathtt{K}|$), (ii) by adding the static methods `createTD` and `createMD` handling the creation of free types and methods (obtained by $|\mathtt{C}|_{CTD}$ and $|\mathtt{C}|_{CMD}$, respectively), and (iii) by adding static fields meant to contain bound descriptors, according to the functions *buildSTD*() and *buildSMD*(). The translation of the root class `Object` is similar, but we don't have methods, `createMD` and static fields.

4.4 Static Type and Method Descriptors

Bound type descriptors and method descriptors are completely known at compile-time, so they can be created once and for all at the class load-time, i.e., in the initialization code of newly-created static fields, exploiting standard environments Θ_s and Π_s. In particular, when registering a parametric type the actual descriptor is yielded, while registering a method returns the position of the method descriptor in the VPMT (for details on this, see [10]). Then, because of the special management of VPMTs, instead of creating the descriptors for the free method signature `L`, we create its version $\mathit{topM}_\Delta(\mathtt{L})$ in the highest super-class defining it. In fact, registering a method descriptors will cause a down-propagation of registrations on all the sub-types, so starting from the top version guarantees all the VPMTs to be properly completed.[1]

4.5 Translating the Constructor

The translation of a constructor `K` provides for the extra-argument, containing the type descriptor of the current instance. The field

[1] These are details of the library classes supporting the translation, which are mostly unimportant here.

Translation for classes:

$$|\texttt{class Object<> extends Object<>\{\}}| = \left\lceil \begin{array}{l} \texttt{class LMObj extends Object \{} \\ \quad \texttt{\$TD td; \$TD getTD()\{return td;\}} \\ \quad \texttt{LMObj(\$TD td)\{this.td=td;\}} \\ \quad |\texttt{C}|_{CTD} \\ \texttt{\}} \end{array} \right\rfloor$$

$$\frac{\Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \qquad \mathit{buildSTD}(\texttt{C}) = \overline{\texttt{f}}_{ST}\ \overline{\texttt{e}}_{ST} \qquad \mathit{buildSMD}(\texttt{C}) = \overline{\texttt{f}}_{SM}\ \overline{\texttt{e}}_{SM} \qquad \texttt{C} \neq \texttt{Object}}{|\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}}| = \left\lceil \begin{array}{l} \texttt{class C extends } |\texttt{N}|_{\Delta}\ \texttt{\{} \\ \quad \texttt{static } \overline{\texttt{TD}}\ \overline{\texttt{f}}_{ST}\texttt{=}\overline{\texttt{e}}_{ST}\texttt{;} \\ \quad \texttt{static int } \overline{\texttt{f}}_{SM}\texttt{=}\overline{\texttt{e}}_{SM}\texttt{;} \\ \quad |\overline{\texttt{T}}|_{\Delta}\ \overline{\texttt{f}}\texttt{;}\quad |\texttt{C}|_{CTD}\quad |\texttt{C}|_{CMD}\quad |\texttt{K}|\quad |\overline{\texttt{M}}|_{\texttt{C}} \\ \texttt{\}} \end{array} \right\rfloor}$$

Static descriptors initilization:

$$\frac{\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{N\{...\}} \qquad \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \qquad \overline{\texttt{N}}_B = \mathit{boundT}_{\Delta}(\mathit{getT}(\texttt{C}))}{\mathit{buildSTD}(\texttt{C}) = \overline{\texttt{f}}_{ST} \quad |\overline{\texttt{N}}_B|_{\Theta_s}}$$

$$\frac{\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{N \{...\}} \qquad \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \qquad \overline{\texttt{L}}_B = \mathit{topM}_{\Delta}(\mathit{boundM}_{\Delta}(\mathit{getM}(\texttt{C})))}{\mathit{buildSMD}(\texttt{C}) = \overline{\texttt{f}}_{SM} \quad |\overline{\texttt{L}}_B|_{\Pi_s,\Theta_s}}$$

Translating the constructor:

$$\frac{\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;K }\overline{\texttt{M}}\texttt{\}} \qquad \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}}}{|\texttt{C(}\overline{\texttt{T}}'\ \overline{\texttt{f}}'\texttt{,}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)\{...\}}| = \left\lceil \begin{array}{l} \texttt{\$TD td; \$TD getTD()\{return td;\}} \\ \texttt{C(\$TD td,}|\overline{\texttt{T}}'|_{\Delta}\ \overline{\texttt{f}}'\texttt{, }|\overline{\texttt{T}}|_{\Delta}\ \overline{\texttt{f}}\texttt{)\{} \\ \quad \texttt{super(td.father,}\overline{\texttt{f}}'\texttt{);} \\ \quad \texttt{this.td=td;this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;} \\ \texttt{\}} \end{array} \right\rfloor}$$

Implementing `createTD`:

$$\frac{\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \overline{\texttt{N}}_F = \mathit{freeT}_{\Delta}(\mathit{getT}(\texttt{C})) \qquad \Theta_p = [\texttt{X}_i \mapsto \texttt{p[i]}]\Theta_s \qquad \overline{\texttt{L}}_F = \mathit{topM}_{\Delta}(\mathit{freeM}_{\Delta}(\mathit{getM}(\texttt{C}))) \qquad \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}}}{|\texttt{C}|_{CTD} = \left\lceil \begin{array}{l} \texttt{static \$TD createTD(\$TD[] p)\{} \\ \quad \texttt{\$TD t=\$TDM.register(C.class,p);} \\ \quad \texttt{if (t.checkNew())\{} \\ \quad\quad \texttt{t.father=}|\texttt{N}|_{\Theta_p}\texttt{;} \\ \quad\quad \texttt{t.friendTs=new \$TD[]\{}|\overline{\texttt{N}}_F|_{\Theta_p}\texttt{\};} \\ \quad\quad \texttt{t.friendMs=new int[]\{}|\overline{\texttt{L}}_F|_{\Pi_s,\Theta_p}\texttt{\};} \\ \quad\quad \texttt{t.initVPMT(); \}} \\ \quad \texttt{return t;} \\ \texttt{\}} \end{array} \right\rfloor}$$

Auxiliary functions:

$$\mathit{meths}(\texttt{Object}) = \epsilon$$

$$\frac{\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{D<}\overline{\texttt{U}}\texttt{>\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \texttt{M}_i\texttt{=<}\overline{\texttt{Y}}_i \triangleleft \overline{\texttt{P}}_i\texttt{> T}_i\ \texttt{m}_i\texttt{(}\overline{\texttt{S}}_i\ \overline{\texttt{x}}_i\texttt{)\{}\uparrow \texttt{e}_i\texttt{;\}}}{\mathit{meths}(\texttt{C}) = \mathit{meths}(\texttt{D}) \bullet \overline{\texttt{m}}}$$

$$\frac{\mathit{meths}(\texttt{C}) = \overline{\texttt{m}} \qquad \texttt{M} = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> T m}_i\texttt{(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{}\uparrow \texttt{e;\}}}{\mathit{pos}(\texttt{C},\texttt{M}) = i}$$

$$\frac{\texttt{CT[C] = class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \texttt{M}_i\texttt{=<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> T m(}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)\{return e;\}} \qquad \Delta = \overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}}{\mathit{pMeth}(\texttt{C},\texttt{M}_i) = \mathit{topM}_{\Delta}(\mathit{freeM}_{\Delta}(\mathit{getM}(\texttt{C},\texttt{m}))) \ ; \ \mathit{freeT}_{\Delta}(\mathit{getT}(\texttt{C},\texttt{m})) \ ; \ [\texttt{Y}_j \mapsto \texttt{p[}j\texttt{]}][\texttt{X}_k \mapsto \texttt{td.p[}k\texttt{]}]\Theta_s}$$

$$\frac{\texttt{CT[C] = class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{N \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad f : i \mapsto \epsilon; \epsilon; \varnothing}{\mathit{mEnv}(\texttt{C}) = [\mathit{pos}(\texttt{C},\texttt{M}_i) \mapsto \mathit{pMeth}(\texttt{C},\texttt{M}_i)]f}$$

Implementation of `createMD`:

$$\frac{\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{N \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \mathit{mEnv}(\texttt{C},\texttt{M}_i) = i \mapsto \overline{\texttt{L}}_i; \overline{\texttt{N}}_i; \Theta_{\texttt{i}}}{|\texttt{C}|_{CMD} = \left\lceil \begin{array}{l} \texttt{static int createMD(\$TD t,int pos,\$TD[] p)\{} \\ \quad \texttt{\$MD m=\$MDM.register(t,pos,p);} \\ \quad \texttt{if (m.checkNew())\{} \\ \quad\quad \texttt{t.VPMT[pos].addElement(m);} \\ \quad\quad \texttt{m.lPos=t.VPMT[pos].size()-1;} \\ \quad\quad \texttt{if (pos==}\vec{i}\texttt{)\{} \\ \quad\quad\quad \texttt{t.friendTs=new \$TD[]\{}|\overline{\texttt{N}}_i|_{\Theta_i}\texttt{\};} \\ \quad\quad\quad \texttt{t.friendMs=new int[]\{}|\overline{\texttt{L}}_i|_{\Pi_s,\Theta_i}\texttt{\};} \\ \quad\quad \texttt{\}} \\ \quad\quad \texttt{\$MDM.propagate(m); \}} \\ \quad \texttt{return m.lPos;} \\ \texttt{\}} \end{array} \right\rfloor}$$

Environments for a method:

$$\frac{\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \qquad \mathit{pMeth}(\texttt{C},\texttt{M}_j) = \overline{\texttt{L}}_M; \overline{\texttt{N}}_M; \Theta_{\texttt{M}} \qquad \mathit{pos}(\texttt{C},\texttt{M}_j) = l \qquad \mathit{buildSTD}(\texttt{C}) = \overline{\texttt{f}}_{ST} \quad |\overline{\texttt{N}}_B|_{\Theta_s}; \qquad \mathit{freeT}_{\Delta}(\mathit{getT}(\texttt{C})) = \overline{\texttt{N}}_F \qquad \texttt{e}_m = \texttt{((\$MD)(td.VPMT[}l\texttt{].elementAt(md)))}}{\Theta^{\texttt{C},\texttt{M}_j} = [\texttt{C<}\overline{\texttt{X}}\texttt{>} \mapsto \texttt{td}]\texttt{N}_{Fi} \mapsto \texttt{td.friendTs[}i\texttt{]}, \overline{\texttt{N}}_{Mk} \mapsto \texttt{e}_m\texttt{.friendTs[}k\texttt{]}, \overline{\texttt{N}}_B \mapsto \overline{\texttt{f}}_{ST}}$$

$$\frac{\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}} \qquad \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \qquad \mathit{pMeth}(\texttt{C},\texttt{M}_j) = \overline{\texttt{L}}_M; \overline{\texttt{N}}_M; \Theta_{\texttt{M}} \qquad \mathit{pos}(\texttt{C},\texttt{M}_j) = l \qquad \texttt{M}_j\texttt{=<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> T m(}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)\{return e;\}} \qquad \mathit{buildSMD}(\texttt{C}) = \overline{\texttt{f}}_{SM} \quad |\overline{\texttt{L}}_B|_{\Pi_s,\Theta_s}; \qquad \mathit{topM}_{\Delta}(\mathit{freeM}_{\Delta}(\mathit{getM}(\texttt{C}))) = \overline{\texttt{L}}_F \qquad \texttt{e}_m = \texttt{((\$MD)(td.VPMT[}l\texttt{].elementAt(md)))}}{\Pi^{\texttt{C},\texttt{M}_j} = [\texttt{C<}\overline{\texttt{X}}\texttt{>.m<}\overline{\texttt{Y}}\texttt{>} \mapsto \texttt{md}]\texttt{L}_{Fi} \mapsto \texttt{td.friendMs[}i\texttt{]}, \overline{\texttt{L}}_{Mk} \mapsto \texttt{e}_m\texttt{.friendMs[}k\texttt{]}, \overline{\texttt{L}}_B \mapsto \overline{\texttt{f}}_{SM}}$$

Translation for methods:

$$\frac{\texttt{CT[C]=class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> }\triangleleft \texttt{ N \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;K }\overline{\texttt{M}}\texttt{\}} \qquad \Delta = \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \qquad \Gamma = \overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}\texttt{, this:C<}\overline{\texttt{X}}\texttt{>} \qquad \mathit{mtypemax}(\texttt{m},\texttt{C}) = \overline{\texttt{D}} \mapsto \texttt{D} \qquad \texttt{e}_i = \begin{cases} \texttt{x}'_i & \text{if} \quad \texttt{D}_i = |\texttt{T}_i|_{\Delta} \\ (|\texttt{T}_i|_{\Delta})\texttt{x}'_i & \text{otherwise} \end{cases}}{|\texttt{M=T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{return e}_0\texttt{;\}}|_{\texttt{C}} = \left\lceil \begin{array}{l} \texttt{D m(int md,}\overline{\texttt{D}}\ \overline{\texttt{x}}'\texttt{)\{} \\ \quad \texttt{return } [\overline{\texttt{e}}/\overline{\texttt{x}}]\,|\texttt{e}_0|_{\Delta,\Gamma,\Theta^{\texttt{C},\texttt{M}},\Pi^{\texttt{C},\texttt{M}}} \\ \texttt{\}} \end{array} \right\rfloor}$$

Figure 3: Main translation functions

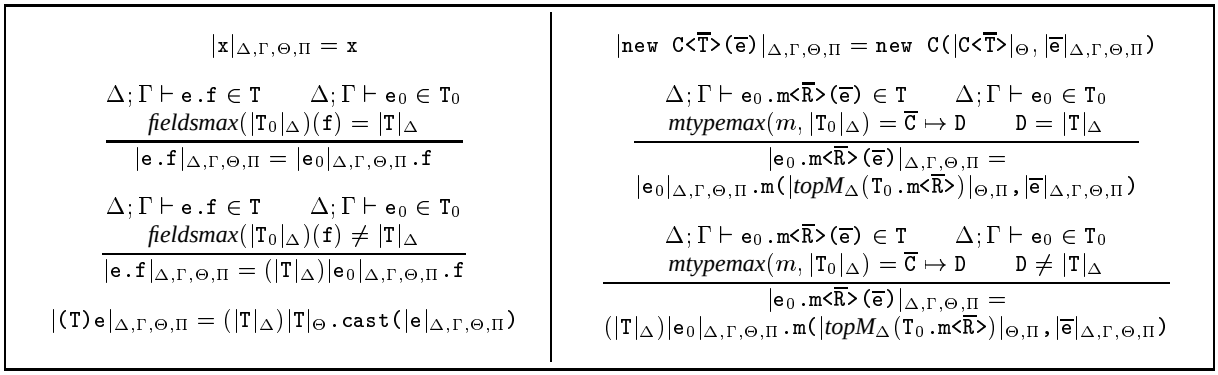


Figure 4: Translating expressions

`$TD.father` is used to pass to the super-class its descriptor. Then, the descriptor is stored into an instance field `td`, inserted by the translation in each class, and yielded by a method `getTD` [2].

4.6 Creating type and method descriptors

The method `createTD` accepts the descriptors of the type parameters, and registers the descriptor of the type into `$TDM`. Then, if this is the first time this was registered (controlled with `checkNew`), in `createTD` we also fill the content of the fields (i) `father`, with the descriptor of the direct super-type `N`, (ii) `friendTs`, with the descriptor for the free types of the class, and (iii) `friendMs` with the descriptor for the free methods of the class. This is supported by the Θ_p environment mapping the type variables to the arguments of the method `createTD`.

For method descriptors, first of all we build an auxiliary function *pMeth*(`C`, `M`) taking a class `C` and a method `M`, and returning a triplet of elements $\overline{\texttt{L}}$; $\overline{\texttt{N}}$; Θ containing respectively: (i) the free method signatures in `M` exploiting some of its type variables $\overline{\texttt{Y}}$, (ii) the free types in `M` exploiting some of its type variables $\overline{\texttt{Y}}$, and (iii) a Θ environment binding `C`'s type variables $\overline{\texttt{X}}$ and `M`'s type variables $\overline{\texttt{Y}}$ to the corresponding descriptors. In particular variables $\overline{\texttt{Y}}$ are associated to the arguments `p` of `createMD`, while variables $\overline{\texttt{X}}$ are associated to the field `p` of the descriptor `t`, representing the receiver for the invocation. Then, the function *mEnv*(`C`) maps positions in the VPMT to such triplets, leaving blanks (ϵ; ϵ; $\oslash$) the triplets for those methods which are not redefined in `C`, but are just inherited from the super-class. In fact, the function f, which maps each position to ϵ, is filled only with the triplets for the methods actually defined in `C`.

The method `createMD` accepts the receiver descriptor `t`, the unique identifier `pos` of the method, and the descriptors of the parameters `p`. It registers the method descriptors, and in the case this was the first time, it proceeds by filling the rest of the object `m`. In particular, it adds `m` in the VPMT of `t`, it registers its position in `m.lPos` and then, depending on i, it stores the fields `friendTs` and `friendMs`, by exploiting the result of the auxiliary function *pMeth*(`C`, $\texttt{M}_i$). The presence of $\overline{i}$ in the `if` statement is meant to model a sequence of `if` statements on all the values assumed by i.

4.7 Translating methods

First of all, we build the $\Theta^{\texttt{C,M}}$ and $\Pi^{\texttt{C,M}}$ environments that will be used to translate the body of a method `M` in `C`. They associate (i) bound descriptors to the static fields of the class, (ii) descriptors using only type variables of the class to friend types/methods of the instance field `td`, and (iii) descriptors using type variables of the method to friend types/methods of the method descriptor $\texttt{e}_m$ of the current VPMT. Then, the type of `this` is associated to the the local type descriptor `td`, and the signature of the current descriptor is associated to the formal argument `md`.

The translation for a method follows the pattern of FGJ. The only difference here, is that we have to add an extra argument that will contain the position of the method descriptor in the VPMT. The environments $\Theta^{\texttt{C,M}}$ and $\Pi^{\texttt{C,M}}$ created in this way will then be used to properly translate the expressions contained in the method `M` of class `C`.

4.8 Translating expressions

The rules defining translation of expressions are shown in Figure 4. The translation for a variable `x` and for a field accessing `e.f` is the same as in FGJ. A cast `(T)e` is translated so as to invoke the method `cast` on the descriptor for `T`, passing the translation of `e`. The allocation expression `new N(`$\overline{\texttt{e}}$`)` is translated by passing as first argument the descriptor for `N`. Analogously, in method invocation we pass as first argument the position of the method descriptor in its VPMT.

5. AN EXAMPLE

In order to allow a better understanding of our formalization, here we provide a comprehensive example of translation. Our source code is shown in Figure 5. It contains a FGJ class `Pair` with two type parameters, two corresponding fields, and five methods doing several things. The corresponding translation according to our formalization is provided in Figure 6; The class `Pair` has the bound type `Pair<Object,Object>` and no bound methods, so the translated class will just have the static field `fST_0`. The translation of the constructor directly follows from the definition. The class `Pair<R,S>` has the free type `Pair<S,R>` (used by the method `reverse`) and the free method signature `Pair<R,S>.reverse<>()` (used in the method `chgSecond`), from which follows the method `createTD` in the translated code.

Then, each method has its own free types and free method signatures. In particular, `Pair<R,S>.chgFirst<T>` has the friend type `Pair<T,S>`, while `Pair<R,S>.chgSecond<T>` has the friend type `Pair<T,R>` (receiver of the latter invocation of `reverse`), and the friend methods `Pair<S,R>.chgFirst<T>` and `Pair<T,R>.reverse()`. Correspondingly, in the translated class we have the method `createMD`

[2] Such a method supports the inspection of the descriptor of an object.

```
class Pair<R extends Object,S extends Object> extends Object{
    R r;
    S s;
    Pair(R r,S s){ super();this.r=r;this.s=s;}
    <> Pair<S,R> reverse(){ return new Pair<S,R>(this.s,this.r);}
    <> Pair<Object,Object> getOO(){return new Pair<Object,Object>(new Object(),new Object());}
    <T> Pair<T,S> chgFirst(T t){ return new Pair<T,S>(t,this.s);}
    <T> Pair<R,T> chgSecond(T t){ return this.reverse<>().chgFirst<T>(t).reverse<>();}
    <> R castToFirst(Object o){ return (R)o;}
}
```

Figure 5: Example of source code

```
class Pair extends LMObj{

    static TD fST_0=Pair.createTD(new $TD[]{LMObj.createTD[]{},LMObj.createTD[]{}});

    Object r; Object s;

    $TD td; $TD getTD(){ return td;}
    Pair($TD td,Object r,Object s){ super(td.father);this.r=r;this.s=s;}

    static $TD createTD($TD[] p){
      $TD t=$TDM.register(Pair.class,p);
      if (t.checkNew()){
        t.father=LMObj.createTD(new $TD[]{});
        t.friendTs=new $TD[]{  Pair.createTD(new $TD[]{p[1],p[0]})  };
        t.friendMs=new int[]{  Pair.createMD(new $TD[]{p[0],p[1]},0,new $TD[]{})};
        t.initVPMT();
      }
      return t;
    }

    static int createMD($TD t,int pos,$TD[] p){
      $MD m=$MDM.register(t,pos,p);
      if (m.checkNew()){
        t.VPMT[pos].addElement(m);
        m.lPos=t.VPMT[pos].size()-1;
        if (pos==0) {   t.friendTs=new $TD[]{};
                        t.friendMs=new int[]{};  }
        if (pos==1) {   t.friendTs=new $TD[]{};
                        t.friendMs=new int[]{};  }
        if (pos==2) {   t.friendTs=new $TD[]{ Pair.createTD(new $TD[]{p[0],td.friendTs[1]})};
                        t.friendMs=new int[]{};  }
        if (pos==3) {   t.friendTs=new $TD[]{ Pair.createTD(new $TD[]{p[0],td.param[0]})};
                        t.friendMs=new int[]{ Pair.createMD(td.friendTs[0],2,new $TD[]{p[0]}),
                                              Pair.createMD(t.friendTs[0],0,new $TD[]{})     }; }
        if (pos==4) {   t.friendTs=new $TD[]{};
                        t.friendMs=new int[]{};  }
        $MDM.propagate(m);
      }
      return m;
    }

    Pair reverse(int md){ return new Pair(td.friendTs[0],this.s,this.r);}

    Pair getOO(int md){return new Pair(fST_0,new Object(),new Object());}

    Pair chgFirst(int md,Object t){
        return new Pair((($MD)(td.VPMT[2].elementAt(md))).friendTs[0],t,this.s);}

    Pair chgSecond(int md,Object t){
        return this.reverse(td.friendMs[0])
                   .chgFirst((($MD)(td.VPMT[2].elementAt(md))).friendMs[0],t)
                   .reverse((($MD)(td.VPMT[2].elementAt(md))).friendMs[1]);}

    Object castToFirst(Object o){ return td.p[0].cast(o);}
}
```

Figure 6: Translation of the source code

as shown in Figure 6.

Finally, the translation of methods simply proceeds as follows. In the signature, argument types and return type are erased, and one argument of type `int` is added. Then, the translation of the returning expression extends the one done for FGJ, but we pass the type descriptor in the `new` expressions, the position of method descriptor in method calls, and we exploit the method `$TD.cast` for implementing casts. Bound type descriptors and method descriptors are accessed through the static fields of the class, free types and methods of the class through the friends of the local type descriptor `td`, and free types and methods of the methods through friends of the current method descriptor. The latter is obtained accessing the VPMT of `td` and exploiting the formal parameter `md`, by the expression `(($MD)(td.VPMT[l].elementAt(md)))` where `l` is the (static) position of the method in the VPMT, say 0 for `reverse`, 1 for `getOO`, 2 for `chgFirst`, and so on.

6. FROM FORMALIZATION TO IMPLEMENTATION

We think it is interesting to discuss how we are going to develop an implementation of LM out from the formalization here introduced. First of all, we rely on a free tool that builds parsers and/or tree generators from source specification files containing BNF-like grammars, which is Sun's JavaCC product [7]. The main features of this tool is that it produces Java code, and permits to insert pieces of Java code in the source specification, allowing to have a fine-grained control on the parsing process and on the shape of the generated tree. Then, the JavaCC distribution also includes the source for creating a parser for Java 1.2.

Our goal here is not to produce an actual implementation suitable for a large-scale release. Instead, the objective is to obtain a prototype of LM translator, which can be used as a tool supporting the measurements of the performance of the translated code. In fact, since relevant performance issues concerns memory footprint and load-time overhead, we need to translate medium-large benchmarks, so we can't just rely on small applications translated by-hand.

Also, we would like to obtain an implementation allowing for the fast prototyping of new versions of the translation. In fact, we believe that the process of measuring performance will give feedbacks motivating an appropriate tuning of the translation. Basically, we intend to address this issue by clearly encapsulating the part of the translator dealing with the key concepts related to parametric polymorphism, i.e., the part implementing the formalization provided in this paper.

As our target is a prototype, we don't need it to be fully-featured. The language accepted is broadly similar to the one accepted by GJ. The two basic differences are (i) that we won't allow to omit type parameters specification from method invocation[3], and (ii) we won't deal with raw types (see [6] for details on these issues). Producing a fast translation process is not a primary goal. Then, we don't need our translator to intercept all the kinds of error due to an incorrect program. We accept that errors can be caught by either the translator or by the successive actual compilation of the resulting Java file. As a result, we won't issue any constraint on how the errors intercepted by the translator are notified to the user.

[3]Type parameter inference is not possible, in general, in the implementations supporting parametric types at run-time

The need for carefully studying a methodology for building the translator comes in from the fact that the complete syntax of Java is huge, so the translator will be a very complex system. Two arguments suffice in emphasizing this: the tree generator for Java, as created by JavaCC, is about 9000 lines of Java code, and the number of different nodes of the tree, which is basically the number of cases we have to consider during the translation, is bigger than 50.

It worth noting that the methodology we will discuss here is not strongly tested, and it should basically considered a proposal for addressing the specification here discussed. When the implementation will be finished we expect to have gathered feedbacks and experience for improving the methodology and describing its details. The important thing here, is our claim that the way we formalized the translation in this paper is the most suitable approach for supporting the actual implementation of the prototype. Basically, it plays a crucial role in encapsulating the core of the translation, allowing changes to be limited to this part, and allowing to build the actual translator on top of it in an incremental way.

1. The first step of the process is to obtain a trivial translator for the source language, translating a source code written in Java with parametric polymorphism in itself. Thanks to the JavaCC tool and the JavaCC specification file of Java 1.2 provided with the distribution, this can be simply done as follows:

   - Creating a suitable tree generator from the JavaCC specification file of Java 1.2, by *decorating* it with the appropriate insertion of Java code.
   - Adding the syntax needed to support parametric polymorphism.
   - Defining a simple visitor for the tree, re-producing the source code given as input.

   The basic goal of this step is to immediately produce the JavaCC specification file of the full source language, which helps in understanding the complexity of the domain and in enumerating all the language constructs we will have to deal with. Later, this will help in deciding what is the better order for incrementally building the translator on the top of its core.

2. The second step is to produce the formalization of the translation. In particular it is important that doing this we focus on all the important issues of the translation, and abstract away from details that are conceptually unnecessary. The one given in the paper fulfills both the requirements, but left out, mostly for space reasons, important aspects such as parametric interfaces and inner classes.

3. The third step is to implement a one-to-one translator taking a source file containing a closed set of FGJ classes and producing a valid Java file. This turns out to be a mere programming exercise, as the source language is very simple and its translation behaviour has been precisely described in the formalization. What we obtain from this step, is the core of the translator.

   At the current time, our development process reached this point, that is, we already have a translator for FGJ to Java, which for instance, can be used to translate our example of Section 5. So, the next steps actually describe what we meant to do for obtaining the full prototype.

4. The following step, is to adapt this translation so as to produce a somewhat fully-featured translator for FGJ into Java, comprehending the management of multiple source files, and dealing with packages and package-qualified class names. The only difference between the result of this step and the final prototype is on the "size" of the source language.

5. The translator produced so far has to be carried from accepting FGJ files to full Java files providing parametric polymorphism, by adding one language constructs at a time. Doing this we exploit the specification file obtained in Step 1, basically adding one BNF rule each time. The most appropriate way of doing this is, orderly:

   - adding the management of class members left-out by FGJ, such as inner classes and constructors, as well as the management of interfaces and static members;
   - adding primitive types and arrays;
   - adding the remaining constructs about expressions;
   - adding the remaining constructs about statements;
   - adding the management of other issues, such as field shadowing and method overriding.

   Where possible, each addition should be tested and debugged alone.

6. Finally, remaining orthogonal issues can be addressed, such as dealing with the inspection of external classes that are used by the code we are translating, but whose source code is not available. These can be either legacy Java classes or LM-generated classes.

7. CONCLUSIONS

The contribution of this paper is twofold. On the one hand, it provides for a compact yet comprehensive formalization of LM translator, a proposal for extending Java with parametric polymorphism by means of a translation. As the LM translator is a complex system, whose informal description tends to be long and typically requires to provide many low level details, the formalization here introduced is an unavoidable tool for its complete and precise understanding. On the other hand, we described how such a formalization can help in quickly building a core prototype which later can be incrementally extended so as to lead to the final implementation.

Our current and future work is devoted on applying our methodology for implementing LM translator, and to go further with its formalization. In fact, the one given in this paper cannot be applied for directly proving properties such as type preservation. By reducing the scope of the target language we should be able to follow the same approach taken in [3]. In particular, it should be possible to define a translation covering a sufficiently large subset of LM behaviour by means of a compilation of FGJ into an extension of FJ with side-effects (field assignment) and a minimal support of reflection.

8. REFERENCES

[1] G. Bracha. *Adding Generic Types to the JavaTM Programming Language*. Java Specification Request, JSR-000014, http://java.sun.com, 1998.

[2] C. Cartwright and G. Steele. Compatible genericity with run-time types for the Java programming language. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 201–215. ACM, October 1998.

[3] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java, a minimal core calculus for Java and GJ. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 132–146. ACM, October 1999.

[4] M.Ellis and B.Stroustroup. *The Annotated C++*. Addison-Wesley, 1990.

[5] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Symposium on Principles of Programming Languages*, pages 146–159. ACM, 1997.

[6] M. Odersky, P. Wadler, G. Bracha, and D. Stoutamire. Making the future safe for the past: Adding Genericity to the Java programming language. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 183–200. ACM, October 1998.

[7] Sun Microsystem. JavaCC 2.0. Distributed by Metamata, http://www.webgain.com/products/metamata/java_doc.html.

[8] Sun Microsystems. *JSR-14 Public Draft*. http://java.sun.com, 2001.

[9] D. Syme and A. Kennedy. Design and implementation of generics for the .NET Common Language Runtime. In *proceedings of Programmin Languages Design and Implementation (PLDI2001)*. ACM, June 2001.

[10] M. Viroli. Parametric polymorphism in Java: an efficient implementation for parametric methods. In *Symposium on Applied Computing (SAC)*, pages 610–619. ACM, March 2001.

[11] M. Viroli and A. Natali. Parametric Polymorphism in Java: an approach to translation based on reflective features. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 146–165. ACM, Oct 2000.

A mode system for read-only references in Java

Mats Skoglund and Tobias Wrigstad
{matte,tobias}@dsv.su.se

Department of Computer and Systems Sciences (DSV),
Stockholm University/Royal Institute of Technology

1 Introduction

The use of references is typical in object-oriented programming. They are used for *e.g.* constructing compound objects, *i.e.* objects that hold references to other objects, and to perform in-place updates. Object sharing with references introduces problems since every object holding a reference to another can use it to invoke methods that modify the referenced object's state. This makes it harder to control the origin of changes to a specific object and thus makes it harder to reason about programs [8].

Since all references always give full access to their referenced objects' protocol, it is generally not possible to share an object by reference without risking that its mutator methods are used to modify it. In C++, this problem can be partly addressed by the use of const objects, const pointers and const methods although with some limitations. This is not possible in Java since it lacks a similar construct. Instead, approaches such as the proxy-pattern or more Java-specific constructs such as interfaces can be used to some extent but with some limitations, see *e.g.* [7] and [8].

We propose a new Java construct for read-only references that protects the transitive state of its referenced object. It is an improvement over C++'s const in that it is transitive and furthermore it does not suffer from some of the limitations of *e.g.* protection through interfaces or the limitations of the read-only constructs reviewed in related work.

2 Motivation

The local state of an object is made up of the values of its member variables and can be modified by *e.g.* assigning to a member variable in the object. With compound objects it is sometimes desirable to reason about the *transitive state* as opposed to the local state. The transitive state is made up not only of the member variables of the object, but also of the member variables' member variables and their member variables' member variables and so on. The transitive state of a compound object can be modified by either changing the local state of the object or by changing the local state of an object contained in the compound object's transitive state. We will use the term state to denote the transitive state throughout this paper.

A problem with compound objects arises when references to the compound object's sub-objects are shared with others. If all changes to a compound object's transitive state must be made via the object's protocol *e.g.* in order to keep the compound object consistent, sharing sub-objects with others can lead to errors. Changes made to a sub-object directly via a reference and not via the compound object's protocol bypasses any controls in the compound object's methods. Thus, such changes might invalidate internal invariants of the compound object leading to errors.

2.1 An example of a problem situation with sharing via references

The event mechanism in Java can be used for propagating state change notifications from one source object to multiple listener objects. When using events according to the JavaBeans API specification [5], one or more objects are normally included in the event object to represent the

state change to be notified. This makes the event object a compound object with references to sub-objects. An object contained in an event object is possibly shared with other compound objects (such as the source object).

It is strongly recommended in the JavaBean API specification that public access to an event object's members is denied and that accessor methods are used for exposing an event object's transitive state. However, if an object is exported from the event object through an accessor method by reference, the transitive state of the event object may still be changed underfoot if *e.g.* a mutator is invoked on the exported object (for a more general discussion, see [7] and [9]).

In Java, objects are always exported in such an "uncontrolled way" since we cannot impose "access rights" on the exported references – existing mutator methods may always be invoked. An exporter has no control over the use of the exported references and a receiver has no knowledge of whether it violates the exporter's intentions if it modifies an object via an incoming reference.

2.2 Related work

Approaches addressing the problem of exporting objects via references that can be used only to read the state of an object but not to modify it have been proposed in *e.g.* [7,11,6,9,8]. Most of these approaches use different kinds of read-only constructs to enforce the property that a reference cannot be used to modify the transitive state of its referenced object. The proposed constructs use variable annotations to indicate their read-only properties in the program. The read-only variable annotations are often accompanied by read-only annotations on methods to indicate which methods are safe to invoke on read-only variables.

Boyland et.al. [1] points out that the proposed read-only constructs are similar in their definitions but differ in semantics since the proposed constructs were defined for different purposes. Common properties of read-only references are that they cannot be used to modify the object that they refer and that only read-only methods can be invoked on them.

A common property of read-only methods is that they should not modify its enclosing object. Some of the read-only mechanisms require this last property to be transitive on compound objects *i.e.* a read-only method should not modify its enclosing objects transitive state. The motivation for transitive protection is that if the mechanism only protects the local state of an object it is sometimes possible to obtain a reference to a sub-object of a compound object that is not protected and use that reference to modify the compound object's transitive state and thus perhaps violate some invariant [6]. This is also the case if the mechanism would only protect a finite number of levels of the transitive state. Then it would be possible to obtain protected references from member variables' member variables in a number of steps and perhaps finally obtain an unprotected reference that can be used for modification.

Restrictions on ordinary programming We believe that the proposed constructs are too limiting on ordinary programming in many respects. For example, in order to protect the transitive state of a compound object, different read-only method constructs named *functional methods* and *clean messages* are defined in [8] and [9] respectively. These constructions have, in our point of view, some unnecessary limitations on ordinary programming. For example, neither a functional method nor a clean message can be used to modify objects, not even objects that are not part of the state of the protected object. This does not permit neither functional methods nor clean messages to *e.g.* perform in-place updates, even if *e.g.* the types of the parameters exclude the possibility that any argument is an alias to the enclosing object's transitive state. In our point of view, this is an unnecessary restriction since the read-onlyness should only concern the object that encloses the method.

Furthermore, in [8] a functional method may not return writeable references to instances created within the method which makes their construct unsuitable for *e.g.* the factory method pattern as described in [3].

In our point of, a read-only method should only protect its enclosing object's transitive state when invoked on a read reference but not necessarily when invoked on a write reference. For

example, a getter method should be able to return a writable reference to a member variable if invoked on a write reference since the object holding the write reference is already allowed to modify sub-objects of the referenced object. This means that there is no need for *e.g.* defining two different getter methods depending of the type of the reference, one that returns a read-only reference to an object and another that returns a writeable reference to the same object.

In [7], every programmer-supplied type has an implicit *readonly* supertype. The type of every variable declared as readonly will be changed to the corresponding readonly supertype consisting only of those methods that have been annotated by the programmer not to be state changing. As pointed out in [8] the use of implicit supertypes limits the use of protected member variables and methods, and furthermore, it creates a dependency on inheritance and a type system.

Validation of annotations The read-only constructs presented in [9] requires that the annotation of methods is done by the programmer and assumes that it is done *correctly*, *i.e.* methods that are read-only should be annotated as such by the programmer. This means that a programmer can, deliberately or undeliberately, annotate a method as read-only even if it modifies its enclosing object's state, making it possible to modify an object via read references with the problems described above.

In [7], as in [9], the programmer decides which methods should be read-only by annotations on the methods. However, no technique is presented that ensures that the annotations are correct with respect to the code. This means that methods may exist that are clearly read-only but cannot be used as such since they lack the readonly qualifier. Also, it is not clear whether methods that are annotated as read-only are validated not to modify their enclosing objects' state. If not, a method that modifies its enclosing object's state can possibly be invoked via read-only references.

C++'s const C++'s const can be used to some extent to achieve protection of objects. However, const methods only protect the local state of the enclosing object. To make the protection offered by const transitive, we need to include all objects in the transitive state in the local state. This can be done by storing an object in a variable as opposed to storing a reference to it. Thus, transitive protection with const precludes object sharing since all sub-objects need to be included in the enclosing object. While it is possible to store a reference to an object included in another object in a field, the referenced object will not achieve the transitive protection from changes from the object storing the reference. Furthermore, objects may not be passed to another as arguments to a method and stored in member fields without copying, precluding any intended sharing. The protection offered by const can easily be circumvented since const can be cast away.

Java's interfaces Interfaces in Java can be used to export parts of an object's protocol hiding any mutators and thus protecting an object from changes via a reference declared to be of the interface type. There are, however, a number of problems with references. The programmer must decide which methods that are mutating, which might be far from easy due to *e.g.* dynamic binding. Moreover, there is nothing to verify that the method implementing the method declaration in the interface does not modify its enclosing object's state. Also, there is nothing to prevent a subclass from overriding a supposedly non-mutating method with a mutator. Interfaces may also be circumvented by casting given that the actual class (or any superclass to that class) of the object is known by the programmer. Interfaces cannot be used for hiding private or protected methods from **this**.

3 A read reference approach to safe object exporting

We propose a mode system that allows exportation of objects without the risk of modification via read references similar to [7,11,6,8], but without some of their restrictions on ordinary programming.

The mode system works by *mode annotations* on variables and methods which control the flow of references in the system in some respects. Each variable, formal parameter, method etc. is associated with a mode qualifier (a mode for short). The modes controls valid assignments, valid method invocations etc.

The mode annotations on variables control what kind of references may be held by the variable. We distinguish between two kinds of references, *read references* and *write references*. Simply put, a write reference is a regular Java reference and a read reference is a reference that is never used for modification of its referenced object (including retrieval of write references that may in turn be used for modification).

The system should be statically checkable[1]. This means that a program can be statically validated to behave correctly with respect to the modes of the references held by the variables and the modes of the methods. Without the **caseModeOf** construct (see below, page 5), there is no need for an actual run-time representation of modes.

A formalisation of the static mode-checking system is shown in Appendix A. We show the rules for well-modedness in a modification of ClassicJava [2]. We do not show any operational semantics since these should be pretty straight-forward.

3.1 The core annotations

The core annotations are read/write annotations which are used to annotate member variables, local variables, formal parameters, method returns and methods. Basically a method declared as read (a read method for short) may not modify its enclosing object's state. Methods declared as write more or less behave like ordinary Java methods. These loose definitions will be refined below.

A variable or formal parameter declared as read (read variables for short) may not be used to invoke write methods and may thus not be used for modification of its referenced object[2]. Note that read variables may be assigned to, in Java terms they are not `final`. Variables declared as write may be used to invoke both read and write methods. The mode of the method return controls the mode of the reference returned by the method. A write variable always holds write references and a read variable always holds read references.

3.2 Approximate annotations

In addition to the core annotations, we have any/context annotations. These annotations do not apply to methods and differ from the core annotations in that these are not really modes but approximations of modes in compile-time (*i.e.* the content of an any or context variable may be either a read reference or a write reference in run-time). A context annotation of a variable means that the mode of the variable is the same as the mode of the *context*. For the moment, let the mode of the context be the same as the mode of the method. Thus, a context variable will be treated as read in a read method and as write in a write method. The last property requires that only write variables be assigned to context variables. Otherwise a read reference stored in a context variable could be wrongfully treated as write reference in a write context.

The any annotation is similar to the context annotation but does not depend on the mode of the context in the same way. Any reference may be assigned to an any variable regardless of its mode. For member variables in a write context, all references in any variables have the same mode as when they were stored in the variable. For member variables in a read context, all any variables will hold read references. For non-member variables, the mode of a reference held by an any variable is always the same as when it was stored in the variable regardless of the mode of the context.

Statically, an any variable must be treated as a read variable since it may contain a read reference whose 'readness' would be broken were it possible to invoke write method on it. Dynamically,

[1] The **caseModeOf** (see below, page 5) construct enables static checkability by performing run-time checks still ensuring that the program is well-behaved with respect to modes.

[2] For simplicity, we disregard from public variables.

an any variable may be converted to a write variable (*i.e.* treated as if it held a write reference) if the mode of the reference stored in the variable is really write, similar to a regular downcast.

3.3 Declarations

Member variables must be declared as either read, context or any. They may not be declared as write since write variables always hold write references which could be used to change the referenced object in read methods[3]. The **this** variable is context.

Local variables and method returns must be declared as either read, write, context or any. Variables declared as context must always refer to members of the enclosing object, *i.e.* may only be assigned from members or other context variables.

Formal parameters must be declared as either read, write or any. They may not be declared as context since the mode of the argument passed to the methods may not have the same mode as the method's context.

Context We say that all statements in a method body executes in the same *context*. The mode of the context is the run-time mode of the reference used to invoke the method. For a write method, this is always write since only write references may be used to invoke write methods. For a read method, the context is either read or write since references of both modes may be used for invoking read methods. Thus, in a read method the mode of a context variable may be either read or write. In compile-time, we must assume that the mode of a context variable in a read method is read by conservativness. However, we supply a dynamic construct that can be used to determine the actual mode of the context in run-time, increasing the flexibility of read methods. This construct is called **caseModeOf** (see below).

Note that this means that the definition of a read method above has been slightly altered. The new definition of a read method states that the a read method does not modify its enclosing object's state when invoked via a read reference. When invoked via a write reference, a read method may behave as a write method since the mode of the references held by any variables and context variables may now be write. Thus, the annotations on methods state the possible modes of the context analogous with the approximate annotations on variables and parameters. For example, the **caseModeOf** construct may change a method's behaviour according to the mode of the context.

Adding optional dynamics The **caseModeOf** construct has two purposes. It can dynamically check the mode of an any variable to allow it to be used as a write variable if it holds a write reference, and it can dynamically check the mode of the context to allow context variables to be used as write variables if the mode of the context is write.

The layout of the **caseModeOf** construct is similar to Java's switch-statement. It consists of a check on a variable and two blocks, the read block and the write block. The checked variable will be assumed to be write in the write block and read in the read block. It will not proceed to any other block. In run-time, the mode of the reference held by the variable will be checked and the corresponding block executed.

The **caseModeOf** construct is optional in the sense that it can be removed along with the any qualifier. This yields a system which is statically checkable without any need for dynamic checks or run-time representation of modes but with the drawback that it becomes less flexible.

Methods A method must be declared as either read or write. For simplicity, we require that all method overriding preserves the mode of all formal parameters and also the mode of the method.

A read method treats all member variables as `final` and statically assumes that all context variables contain read references, *i.e.* must be treated as read variables. This ensures that

[3] Other approaches are of course possible, such as to prevent names of write members to appear in bodies of write methods.

the enclosing object's state cannot be changed by the read method since any accessible member variable is read. Inside the write block of a **caseModeOf** statement on a member variable in a read method, the member may be modified since this block will only be executed in a write context, *i.e.* when the method was invoked via a write reference and it thus may modify its enclosing object's state.

A write method treats all context member variables as write and may thus change or export write references to the state of its enclosing object. Variables declared as read or any must still be treated as read. An any variable may, however, be modified in the write block of a **caseModeOf** statement testing the mode of its contained reference.

Parameters to methods and aliasing Note that a read method may modify the state of its enclosing object even in a read context if a write reference to the state is passed in as an argument. We allow this since the owner of that reference may use it for modification outside the method meaning that any protection from this situation inside the method does not significantly reduce the possibility of changes via the existing write reference. Moreover, it is arguable that the presence of a write reference outside the object should indicate that the changes to the state of the object should be allowed.

To avoid this, a possible solution is to require that all formal parameters to read methods are annotated with read (too restrictive in our sense). Another approach is to require formal parameters to be annotated with any and dynamically check all arguments for aliasing converting any incoming aliases to read references. These techniques could be somewhat optimised by *e.g.* checking if the possible types of possible arguments overlap with the possible state of this.

3.4 A brief example

```
1  public class SecNewThermometerEvent {
2    private read Object source;
3    private context Thermometer thermometer;
4    public void setSource(read Object s):write {
5      this.source = s;
6    }
7    public void setThermometer(write Thermometer th):write {
8      this.thermometer = th;
9    }
10   public read Object getSource():read {
11     return this.source;
12   }
13   public context Thermometer getThermometer():read {
14     return this.thermometer;
15   }
16 }
17 ...
18 write SecNewThermometerEvent event = new SecNewThermometerEvent();
19 event.setSource(read this);
20 event.setThemometer(thermometer);
21 radiators.notify(read event);
22 ...
```

Figure1. Example of event annotated with modes

The code above below, admittedly somewhat contrived for pedagogical reasons, uses the mode system in the construction and use of an event class. The event is used to notify radiator objects

in a room object that a thermometer object has been inserted in the room. The example is written in a Java language extended with the mode qualifiers on variables and methods.

A room object that holds a write reference to the newly inserted thermometer creates a new `SecNewThermometerEvent` object and assigns it to the write variable `event` (18). Then the room invokes `setSource` (19) that assigns the `source` variable in `event` (5). It then invokes the `setThermometer` method passing the thermometer as a parameter (20) and a reference to the thermometer will be stored in the `thermometer` variable in `event` (8). Finally, the radiators in the room are notified about the newly inserted thermometer by invocation of `notify` (21), passing `event` as read so it will be protected.

It is possible to assign `th` to `thermometer` (8) even though the parameter `th` is a write variable and the member variable `thermometer` is declared context. This is since a variable declared as context can be treated as a write variable in a write method and since the `setThermometer` method is declared as write (7) the assignment is allowed. Since the event object is sent to its receivers only as read (21) and since a context variable is always treated as read in a read context, the receivers will only be able to obtain read references to the thermometer object.
The member variable `source` holds a reference to the creator of the event object so that the receivers should be able to *e.g.* determine the origin of the event. This can be done by for example comparing the identity of the object held by `source` with another object's identity. Since `source` is declared as read (2) it is safe to export it from the event via the `getSource()` method rest assured that no receiver will be able to modify it via the exported reference. The formal parameter to `setSource`, `s`, is declared as read (4) since it will be stored in the read variable `source` (5) that in this example should never be allowed to be used to modify the creator.

The `thermometer` variable holds a reference to the thermometer inserted into the room. The `thermometer` is declared context (3), which means that since the event is sent to its receivers only via read references, the receivers may not obtain write references to the thermometer from the event. Neither may the receivers invoke the setters on the event since they are both write methods (4)(7).

4 Conclusions

We have presented a system for controlling references in Java. We distinguish between read references and write references. Our formal mode system is statically checkable and ensures that references exported as read will never be used to modify its referenced object. For increased flexibility, we also introduced an optional dynamic check. We also distinguish between read methods and write methods where read a method never modifies the state of its enclosing object when invoked on a read references. The statically checkable rules ensure that all methods in a program are annotated correctly.

The system is designed for class-based object-oriented programming languages and should be realisable not only in Java. For Java however, it seems reasonable to implement the mode system as an extension of the type system since all variables, method return declarations and formal parameters are associated with modes and since modes also further define the range of values a variable can assume. The presence of inheritance and method overriding in Java affects the criteria of the write method. Here, for simplicity, we require method overriding to preserve the modes of the overridden method. Thus, in addition to the original criteria, a method is a write method also if it overrides a write method.

The mode system is formalised as a type system with judgments, mode rules etc. Our work in progress is the completion of the system with operational semantics for the dynamic behaviour of *e.g.* **caseModeOf** and completion of the proofs of the mode system.

We also plan to extend our Java subset to get closer to the Java Language Specification [4].

References

1. J. Boyland, J. Noble, and W. Retert. Capabilities for aliasing: A generalization of uniqueness and read-only. Accepted to ECOOP 2001. Under revision.

2. M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR 97-293, Rice University, 1997, revised 6/99. Original in Formal Syntax and Semantics of Java, LNCS volume 1523 (1999), Available from `http://www.cs.rice.edu/CS/PLT/Publications/tr97-293.pdf`.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995.
4. J. Gosling, B. Joy, G. L. Steele Jr., and G. Bracha. *The Java Language Specification Second Edition.* Addison-Wesley Publishing Company, 2000. Available from `http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html`.
5. G. Hamilton, editor. *JavaBeans.* Sun Microsystems Inc, 1997. Available from `http://java.sun.com/products/javabeans/docs/spec.html`.
6. J. Hogg. Islands: Aliasing protection in object-oriented languages. In A. Paepcke, editor, *OOPSLA '91 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 271–285. ACM Press, 1991.
7. G. Kniesel and D. Theisen. Java with transitive access control. In *IWAOOS'99 – Intercontinental Workshop on Aliasing in Object-Oriented Systems. In association with the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, June 1999. Available from `http://cui.unige.ch/~ecoopws/iwaoos/papers/index.html`.
8. P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001. Available from `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.
9. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998.
10. M. Skoglund and T. Wrigstad. A mode system for read references. Technical report, Department of Computer and Systems Sciences, Stockholm University/Royal Institute of Technology, 2001. To appear.
11. B. Stoustrup. *The C++ Programming Language Third Edition.* Addison-Wesley Publishing Company, 1997.

A Mode-checking rules

The rules are shown in Appendixes A.2-A.4 and briefly explained below. The system is a modification of ClassicJava, [2]. Included here are the most integral parts, a complete description of the mode-checking system is included in [10]. The most significant changes with respect to ClassicJava are the integration of the mode system, the **caseModeOf**-statement and the treatment of local variables.

For simplicity, we assume the following: All classes are declared only once, no inheritance chains are cyclic, no method is declared in a class more than once, no field is declared in a class more than once, every class is an extension of another declared class or Object, parameter names and local variable names in the same method do not conflict, method and field overriding preserves all modes and types. Note that we also assume that all methods bodies end with an expression. The result of a method invocation is the result of the last expression in the method body.

We write $P, F, m_c \vdash \langle p \rangle$ to denote that given the program text P, a mapping from local variable names to modes and types F and the mode of the context m_c, we may conclude $\langle p \rangle$. We write $e : (m, t)$ to denote that the expression e can be associated with a mode m and a type t. Note that in rule amcmo, we add an additional premise on the LHS of the $\vdash$ stating the existence of a field *fd* with mode m and type t in the type *t'* replacing an existing field in *t'* with the same name.

All expressions return values. The expressions $vn = e$ and **this**.*fd* $= e$ both return the value of the RHS allowing the expressions to be chained and used as arguments to methods.

We subscript $\vdash$ as $\vdash_{\mathsf{d}}$ for validation of class declarations, as $\vdash_{\mathsf{m}}$ for validation of method declarations, as $\vdash_{\mathsf{t}}$ for validation of mode/type pairs and as $\vdash_{\mathsf{s}}$ for type subsumption and mode conversion. Plain $\vdash$ denotes checks of expressions, statements or sequences of statements and expressions.

In addition to the non-terminals of the abstract syntax, we use the m to denote any mode (*e.g.* $m \in \{\mathsf{read}, \mathsf{write}, \mathsf{any}, \mathsf{context}\}$). We use standard '′' (prime) notation and subscription to

distinguish between meta variables of the same kind, *e.g.* m and m' are both modes, not necessarily the same. The symbol m_c always denotes the mode of the current context, read or write.

A.1 Explanation of the integral parts of the integral rules

Expressions Rule null states that null can be given any mode and thus be stored in every variable or field. Rule new states that references to objects created by new have mode write. Rule rcast states that any result of an expression may be cast to read. Rule lget states that the result of an access of a local variable has the variable's declared mode and type; $var \in dom(F)$ states that var is declared in F and $F(var)$ retrieves the mode and type for a variable name. Rule lset states that an assignment to a local variable requires that both sides have the same mode or that the RHS is a member with mode write and the LHS has mode context. Rule fget states that the result of an access to a field in **this** has the field's declared mode if the mode of **this** is write, else it has the mode read. Rule fset states that assignments to members requires that the mode of the context is write and that the mode on the RHS can be converted to the declared mode of the field on the LHS. Rule call states that write methods may only be invoked on write references, read methods may be invoked on a reference regardless of its mode. It also states that the modes of the arguments must be possible to convert to the same modes as the formal parameters.

Declarations Rule meth states that a method is well-moded if its body is well-moded with its declared mode as the mode of the context. The rule also states that if the method overrides a method with mode write, the method must be declared as write even if it is valid when the declared mode is read. Note that this does not allow methods that would compile as read methods to be declared as write methods and vice versa, except for the case of method overriding because of the difference between the most specific method selected in compile-time and the actual method bound to in run-time.

Statements Rule amcmo states that **caseModeOf** on a member variable declared as any is valid if the expressions in the write-block are valid with the variable as write and vice versa for the read-block. Rule alcmo states the same, but for local variables. Rule ccmo states that **caseModeOf** on a variable declared as context is valid if the expressions in the write block are valid with in with the mode of the context as write and vice versa for the read block.

Sequence Rule sseq states that a sequence of statements or expressions is well-moded if all substatements or subexpressions are well-moded. Note the s here, meaning that statements may be contained in sequences. Also note that an expression is also a statement.

Mode and type rules Rule mconv states that a mode m may be converted to m, any or, if m is context, to the mode of the current context, m_c. Rule mode/type states that mode and type pair is valid if the mode is in the set {read, write, any, context} and the type is declared in P. The rule sub states that the result of an expression with mode m and type t can be subsumed to be of a supertype to t and another mode to which m may be converted. Note that conv/sub accepts sequences of statements. While this might seem slightly unorthodox, it allows smooth treatment of method bodies, which always end with an expression.

A.2 Abstract syntax

$class$:= **class** t **extends** t { $field^*$ $meth^*$ }
$field$:= m_f t vn
$meth$:= m_l t $md((m_a\ t\ vn)^*)$:m_m { $body$ }
$local$:= m_l t vn
$body$:= $local^*$ s_{seq}
s_{seq} := s; | s; s_{seq}
e := null | **new** c | var | $vn = e$ | **this**.fd
| **this**.$fd = e$ | $e.md(e^*)$ | **read** e
s := e | **caseModeOf** ((**this**.fd | var)) {**w:** s_{seq} **r:** s_{seq}}

t := a classname or Object
m_f := read | any | context
m_l := read | write | any | context
m_a := read | write | any
m_m := read | write
vn := a variable name
var := vn | **this**
fd := a field descriptor
md := a method descriptor

A.3 Explanation of symbols (denotes relations)

$<:$ $t <: t'$ iff t' is a superclass to t or t'.
$\in_{\mathsf{c}}$ $t \in_{\mathsf{c}} P$ iff t is a class declared in program P.
$\in_{\mathsf{f}}$ $(m, t, fd) \in_{\mathsf{f}} t'$ iff field fd with mode m and type t is declared in class t'.
FA(e) FA(e) iff $e \equiv$ **this** or $e \equiv$ **this**.fd
$\in_{\mathsf{m}}$ $(m', t', md, ((m, t)\ldots), m'') \in_{\mathsf{m}} t'$ iff method md with return mode/type- pair (m', t') and formal parameters $vn\ldots$ with mode/type-pairs $(m, t)\ldots$ method mode m'' is declared in class t''.

A.4 Mode/type elaboration

$$\text{null}\ \frac{P \vdash_{\mathsf{t}} (m, t)}{P, F, m_c \vdash \mathsf{null} : (m, t)} \qquad \text{new}\ \frac{t \in_{\mathsf{c}} P}{P, F, m_c \vdash \textbf{new}\ t : (write, t)} \qquad \text{rcast}\ \frac{P, F, m_c \vdash e : (m, t)}{P, F, m_c \vdash \textbf{read}\ e : (\mathsf{read}, t)}$$

$$\text{lget}\ \frac{var \in dom(F)}{P, F, m_c \vdash var : F(var)} \qquad \text{lset}\ \frac{\begin{array}{c}P, F, m_c \vdash vn : (m', t) \quad P, F, m_c \vdash_{\mathsf{s}} e : (m, t) \\ m = m' \vee (\text{FA}(e) \wedge m = \mathsf{write} \wedge m' = \mathsf{context})\end{array}}{P, F, m_c \vdash vn = e : (m, t)}$$

$$\text{fget}\ \frac{\begin{array}{c}P, F, m_c \vdash_{\mathsf{s}} \textbf{this} : (m', t) \qquad (m'', t', fd) \in_{\mathsf{f}} t \\ m' = \mathsf{write} \Rightarrow m = m'' \qquad m' \neq \mathsf{write} \Rightarrow m = \mathsf{read}\end{array}}{P, F, m_c \vdash \textbf{this}.fd : (m, t)} \qquad \text{fset}\ \frac{\begin{array}{c}P, F, m_c \vdash_{\mathsf{s}} \textbf{this} : (\mathsf{write}, t) \\ (m', t', fd) \in_{\mathsf{f}} t \qquad P, F, m_c \vdash_{\mathsf{s}} e : (m', t')\end{array}}{P, F, m_c \vdash \textbf{this}.fd = e : (m', t')}$$

$$\text{call}\ \frac{\begin{array}{c}m' = \mathsf{write} \vee m'' = \mathsf{read} \qquad P, F, m_c \vdash_{\mathsf{s}} e : (m', t') \\ P, F, m_c \vdash_{\mathsf{s}} e_j : (m_j, t_j) \text{ for } j = [1, n] \\ (m, t, md, ((m_1, t_1)\ldots(m_n, t_n)), m'') \in_{\mathsf{m}} t'\end{array}}{P, F, m_c \vdash e.md(e_1\ldots e_n) : (m, t)} \qquad \text{mode/type}\ \frac{\begin{array}{c}t \in_{\mathsf{c}} P \\ m \in \{\mathsf{read}, \mathsf{write}, \mathsf{any}, \mathsf{context}\}\end{array}}{P \vdash_{\mathsf{t}} (m, t)}$$

$$\text{mconv}\ \frac{m = m' \vee m' = \mathsf{any} \vee (m = \mathsf{context} \wedge m' = m_c)}{m_c \vdash_{\mathsf{m}} m \leq_M m'} \qquad \text{class}\ \frac{\begin{array}{c}P \vdash_{\mathsf{t}} (m_j, t_j) \text{ for } j = [1, n] \\ P, c \vdash_{\mathsf{m}} meth_k \text{ for } k = [1, p]\end{array}}{\begin{array}{l}P \vdash_{\mathsf{d}} \textbf{class}\ c\ \ldots\ \{m_1\ t_1\ fd_1\ldots \\ \quad m_n\ t_n\ fd_n\ meth_1\ldots meth_p\}\end{array}}$$

$$\text{conv/sub}\ \frac{\begin{array}{c}P, F, m_c \vdash s_{\mathsf{seq}} : (m', t') \\ m_c \vdash_{\mathsf{m}} m' \leq_M m \qquad t' <: t\end{array}}{P, F, m_c \vdash_{\mathsf{s}} s_{\mathsf{seq}} : (m, t)} \qquad \text{meth}\ \frac{\begin{array}{c}t_0 <: t_1 \qquad P \vdash_{\mathsf{t}} (m, t) \qquad m'' \in \{\mathsf{read}, \mathsf{write}\} \\ P \vdash_{\mathsf{t}} (m_j, t_j) \text{ for } j = [1, n+p] \qquad P, \kappa, m'' \vdash_{\mathsf{s}} s_{\mathsf{seq}} : (m, t) \\ (m, t, md, ((m_1, t_1)\ldots(m_n, t_n)), \mathsf{write}) \notin_{\mathsf{m}} t_1 \Rightarrow m'' = m' \\ \kappa = [\textbf{this} : (\mathsf{context}, t_0), v_1 : (m_1, t_1)\ldots v_{n+p} : (m_{n+p}, t_{n+p})]\end{array}}{\begin{array}{c}P, t_0 \vdash_{\mathsf{m}} m\ t\ md(m_1\ t_1\ v_1\ldots m_n\ t_n\ v_n) : m' \\ \{m_{n+1}\ t_{n+1}\ v_{n+1}\ \ldots m_{n+p}\ t_{n+p}\ v_{n+p}\ s_{\mathsf{seq}}\}\end{array}}$$

$$\text{amcmo}\ \frac{\begin{array}{c}P, F, m_c \vdash \textbf{this}.fd : (\mathsf{any}, t) \qquad P, F, m_c \vdash \textbf{this} : (\_, t_1) \\ (\mathsf{write}, t, fd) \in_{\mathsf{f}} t_1, P, F, m_c \vdash s'_{\mathsf{seq}} : (m', t') \\ (\mathsf{read}, t, fd) \in_{\mathsf{f}} t_1, P, F, m_c \vdash s''_{\mathsf{seq}} : (m'', t'')\end{array}}{\begin{array}{c}P, F, m_c \vdash \textbf{caseModeOf}\ (\textbf{this}.fd) \\ \{\textbf{w:}\ s'_{\mathsf{seq}}\ \textbf{r:}\ s''_{\mathsf{seq}}\} : (\mathsf{any}, t)\end{array}} \qquad \text{alcmo}\ \frac{\begin{array}{c}P, F, m_c \vdash vn : (\mathsf{any}, t) \\ P, F[vn : (\mathsf{write}, t)], m_c \vdash s'_{\mathsf{seq}} : (m', t') \\ P, F[vn : (\mathsf{read}, t)], m_c \vdash s''_{\mathsf{seq}} : (m'', t'')\end{array}}{\begin{array}{c}P, F, m_c \vdash \textbf{caseModeOf}\ (vn) \\ \{\textbf{w:}\ s'_{\mathsf{seq}}\ \textbf{r:}\ s''_{\mathsf{seq}}\} : (\mathsf{any}, t)\end{array}}$$

$$\text{ccmo}\ \frac{\begin{array}{c}P, F, m_c \vdash e : (\mathsf{context}, t) \\ P, F, \mathsf{write} \vdash s'_{\mathsf{seq}} : (m', t') \qquad P, F, \mathsf{read} \vdash s''_{\mathsf{seq}} : (m'', t'')\end{array}}{P, F, m_c \vdash \textbf{caseModeOf}\ (e)\ \{\textbf{w:}\ s'_{\mathsf{seq}}\ \textbf{r:}\ s''_{\mathsf{seq}}\} : (\mathsf{context}, t)} \qquad \text{sseq}\ \frac{\begin{array}{c}P, F, m_c \vdash s : (m, t) \\ P, F, m_c \vdash s_{\mathsf{seq}} : (m', t')\end{array}}{P, F, m_c \vdash s; s_{\mathsf{seq}} : (m', t')}$$

Static Analysis of Java Cryptographic Applets

Pierre Boury[1] and Nabil Elkadhi[2]

[1] GIE Dyade,
Domaine de Voluceau - Rocquencourt - B.P. 105
78158 Le Chesnay Cedex France,
e-mail: Pierre.Boury@dyade.fr,
home page: http://www.dyade.fr/fr/actions/VIP/pb_homepage.html
[2] EPITECH,
14-16 rue Voltaire,
94270 Le Kremlin Bicêtre France
e-mail: nelkadhi@club-internet.fr
home page: http://www.epita.fr/~el-kad_n

Abstract. Secure Java applications such as JavaCard applets often rely on cryptography for implementing security functions such as authentication, or the creation of confidential channels. The question addressed by this paper is to provide automated support for the verification of such applications. We describe here our experience in the design and implementation of *StuPa*, a prototype of a static analyzer performing verification of Java cryptographic applets in order to prevent the disclosure of specified sensitive data. We give an overview of the formal models it is based on, particularly those used for modeling cryptographic knowledge. Finally, we review the scope and limitations of the current prototype, and its interest for practical applications. [1]

1 Introduction

JavaCard ([10]) is a simplified subset of Java, which has been lightened and secured to run on smartcards. JavaCard applets are typically used for authentication and electronic commerce applications and have high security requirements. Our objective in this paper is to statically detect potential flaws in an untrusted applet. We are mainly interested in confidentiality properties: while a trusted program is interacting with an untrusted environment, we want to ensure that secret data cannot be disclosed, inadvertently or mischievously. Several formal models for the analysis of cryptographic protocols have been proposed, notably [3] and [13]. These methods allow for the verification of various cryptographic properties such as freshness, authentication and confidentiality. Moreover, some other recent approaches such as [1], [9] and [7], are fully automated.

The point addressed by this paper is the adaptation of these methods to the verification of confidentiality properties of JavaCard applets. Such a task raises several problems. First, these methods generally require a complete specification

[1] This work was partially supported by the TASSC ITEA project

of protocols, including a description of the role of each participant, whereas in the applications we consider, we only possess a Java program. Second, in contrast with these approaches which only apply to dedicated specification formalisms, we need, for practical reasons, to be able to deal with real Java programs in a highly automated way.

The focus of the following work is to derive suitable formal models that comply with these requirements. This task has been done following the methodology of abstract interpretation, as described by [6].

The points addressed in the following are twofold. First, we review in section 2 the formal approach we have designed for the static analysis of confidentiality. This approach has been derived from [3] and is applicable to general cryptographic algorithms. Second, in section 3 we explain how we have adapted this approach to the static analysis of Java. The necessity of dealing with real Java program instead of algorithms specified into a dedicated language raises several concerns we consider below. We explain in sub-section 3.1 how we have dealt with static analysis issues proper to Java such as managing statically unknown values and references. In section 3.2 and 3.3, we consider the problem of identifying and modeling cryptographic actions of Java applets. Finally, our design choices and solutions have been tested and experimented in *StuPa*, a prototype static analyzer. We will briefly summarize the first results of these investigations in section 4.

2 Automated verification of confidentiality properties

To begin with, we designed a convenient formal framework for the static analysis of cryptographic programs. Our approach is an adaptation of Bolignano's method [3], which is oriented toward automated verification of confidentiality. We have modified this method to make it automated and more easily applicable to real programs. The techniques we use to achieve these goals are derived from proof-theoretical results about the evolution cryptographic knowledge. We follow the abstract interpretation methodology in order to translate these results into an automated verification method. The idea of using abstract interpretation in the area of cryptographic protocol verification to get automated procedures has been applied independently in [4] and [9].

Instead of considering Java byte-code at first, we use a simple language of cryptographic actions in order to set up the formal foundations of static cryptographic verification. This language is limited to what is essential and sufficient to describe cryptographic programs: its has simple testing and branching instructions in order to describe program control, and its elementary instructions include cryptographic actions such as equality tests, cryptographic message construction and decomposition, nonce and key generation, and message reception and emission (see table 1 below). Known or public messages are those that could be constructed by an intruder in an untrusted environment using only initially known data and data having leaked from user program. As more and more data get disclosed along program execution, the set of public messages is

$a \in A ::= \epsilon$		*null action*
	$x = y$	*equality test*
|	$\neg\ x = y$	*non-equality test*
|	$x := y$	*assignment*
|	$x := op(x_1, x_2, \ldots, x_n)$	*elementary operation*
|	$y := [x_1; x_2; \ldots, x_n]$	*tuple construction*
|	$detuple(x, [x_1; x_2; \ldots, x_n])$	*tuple read*
|	$x := ?y$	*channel read*
|	$x\ !\ y$	*channel write*
|	$x := encrypt_y^\alpha z$	*encryption*
|	$x := decrypt_y^\alpha z$	*decryption*
|	$decryptfail_y^\alpha z$	*decryption failure*
|	$x := fresh$	*nonce generation*
|	$x, y := keys^\alpha$	*keys generation*

Table 1. *Cryptographic actions*

growing. This set is somehow an approximation of the *cryptographic knowledge* of a hostile intruder. Within this model, we have defined a deduction relation describing when a message is constructible from a set of known messages. What is needed for program verification is an automated procedure to statically compute the evolution of cryptographic knowledge during program execution. We have investigated the formal properties of the deduction relation and derived sound verification procedures from it. A complete description of this approach can be found in [8]. We will only mention that, in this approach, the concrete meaning of cryptographic actions is defined as terms of transformations acting on cryptographic environement made of an environment carrying information about messages content, and of a set containing all disclosed messages describing the state of knowledge of an untrusted intruder. Over that concrete model is defined a static abstract model in order to statically compute the evolution of cryptographic knowledge. Variables are associated to message components, and constraints on these variables express assertions about the structure of messages, and about the set of public data. Sets of constraints approximate cryptographic knowledge and are interpreted as sets of admissible cryptographic environments. Cryptographic actions are interpreted as particular transformers of constraints sets.

We can see how this approach works by considering the following piece of cryptographic protocol (defined in table 2): a message x is read, decrypted into a message z using a secret key k_1, then encrypted into a message w using a secret key k_2, and finally sent. In that example, after stage 1, the constraint $\neg\text{known}(x; k_2)$ means that key k_2 is confidential assuming message x is public, and before stage 3, the constraint $w \approx \{z\}_{k_2}^a$ means that message x is equal to message z encrypted using key u, which is the inverse key of k_1. Table 2 below

action	cryptographic constraints
0. (initial assumptions)	$\kappa_0 = \{is\_simple(k_1),\ \neg known(;k_1),\ is\_simple(k_2),\ \neg known(;k_2)\ \}$
1. $x := ?ic$	$\kappa_1 = \kappa_0 \cup \{is\_simple(k_1),\ \neg known(x;k_1),\ is\_simple(k_2),\ \neg known(x;\ k_2)\ \}$
2. $z := decrypt^a_{k_1} x$	$\kappa_2 = \kappa_1 \cup \{is\_simple(u),\ Inv^a(u,k_1), x \approx \{z\}^a_u\ \}$
3. $w := encrypt^a_{k_2} z$	$\kappa_3 = \kappa_2 \cup \{w \approx \{z\}^a_{k_2}\ \}$
4. $w\,!oc$	$\kappa_4 = \{is\_simple(k_1),\ \neg known(x;k_1),\ is\_simple(k_2),\ \neg known(x;k_2),\ is\_simple(u),\ Inv^a(u,k_1),\ x \approx \{z\}^a_u,\ w \approx \{z\}^a_{k_2}\ \}$

Table 2. *A part of a cryptographic protocol*

shows the evolution of cryptographic constraints resulting from the execution of the protocol.

More generally, this approach has been implemented and tested on programs implementing parts of well-known cryptographic protocols, and it gave sensible results. We use this formal approach in section 3.3 below in order to model cryptographic information in Java programs.

3 Formal models for Java programs

The formal approach of section 2 is ignoring Java-specific problems such as the matching of Java instructions to cryptographic actions, which we consider in sub-sections 3.2 and 3.3 below. In order to set up convenient formal models for Java, we have adapted its former approach to the particular structures of Java Virtual Machine runtime data and values.

On one hand, we have to deal with general issues of Java static analysis such as how to handle unknown values. On the other hand, we have to integrate our general model of cryptographic knowledge in a seamless way into our Java-oriented framework of static analysis. Formal models encoding Java Virtual Machine configurations have to be augmented with cryptographic information, and cryptographic actions have to be identified inside Java programs.

In the following, we explain how we deal with these issues in *StuPa*. As an illustration, we will use a Java implementation of the piece of protocol presented in table 2 above. *StuPa* takes as input a set of Java class-files, and additionnal cryptographic information, and returns as output static information about the cryptographic knowledge at different control points of the program. Table 3 below shows the source code and the byte-code of the *main* method of our example. As additionnal information, we have set the assumptions that the static fields `k1` and `k2` hold confidential keys at their initialization. We also have declared the cryptographic methods `put`, `get`, `encrypt` and `decrypt` so that they are submitted to a particular processing during static analysis. We give some details about this point below.

3.1 Static approximations

The Java Virtual Machine, or JVM, is described in [12], and more formally in [2]. As usual in static analysis, we have to approximate the JVM values which are unknown at compile-time. We make the following choices: unknown JVM numerical values (*int, long, double, float*) are approximated by their type, and unknown references are approximated by their set of possible sites of creation, or *locations*, which are the program control points where they may have been created. The main consequences of these modeling decisions are that actual classes of instances can be resolved accurately, but different instances created at a same control point are modeled as a single abstract instance (corresponding instances fields must be merged).

We also have to make decisions about the modeling of control information. For the sake of simplicity, methods calls are in-lined, which is sufficient in the

```
public static void main() {
  int x_size = 5;    // read incomming message
        // 0 iconst_5
        // 1 istore_0
  int[] x = Env.get(x_size);
        // 2 iload_0
        // 3 invokestatic #8 ⟨Method int get(int)[]⟩
        // 6 astore_1

  int[] z = decrypt(x, k1);    // decipher incomming message
        // 7 aload_1
        // 8 getstatic #9 ⟨Field int k1⟩
        // 11 invokestatic #5 ⟨Method int decrypt(int[], int)[]⟩
        // 14 astore_2

  int[] w = encrypt(z, k2);    // compose outgoing message
        // 15 aload_2
        // 16 getstatic #10 ⟨Field int k2⟩
        // 19 invokestatic #6 ⟨Method int encrypt(int[], int)[]⟩
        // 22 astore_3

  Env.put(w);    // send outgoing message
        // 23 aload_3
        // 24 invokestatic #11 ⟨Method void put(int[])⟩
  return;
        // 27 return
}
```

Table 3. *main method of the Java applet*

absence of recursion, as it is the case in most JavaCard applets. Otherwise, method calls require further standard treatment we don't wish to describe here.

As one can expect, this way we can faithfully model statically-defined instructions: static method call and static fields access and local access on the JVM stack are interpreted without loss of information. Computations on static data are also translated accurately. On the contrary, the abstract interpretation of dynamic instructions induces non-determinism, which arises from branches and method invocation.

3.2 Models of data leakage

In order to be able to state and check confidentiality properties, we must be able to distinguish, inside the program to be analyzed, which part is trusted and allowed to hold secret data, and which part belongs to the untrusted external environment, and toward which no secret information should leak. Note that this requirement differs from our previous cryptographic framework [8], in which such definitions are defined as primitive notions.

Our approach to that point consists in defining a frontier between the user program and its untrusted environment, following an approach inspired from [11]. This is done by declaring a set of trusted classes and by observing the JVM stack. Potential leaks of information are detected as follows. Numerical values are typed and partionned into three sets: surely public messages, possibly confidential messages, and values without cryptographic content (which are not messages). A leak is detected when a trusted method writes a confidential data on a field belonging to an untrusted class, or returns a confidential value to a calling method of an untrusted class, and when a method of an untrusted class gets confidential values as parameters, or reads a confidential data from some field.

For instance, in the example of table 3 above, the definition class of methods `encrypt` and `decrypt` is trusted, whereas the definition class of methods `put` and `get` is not. When an *int* array is passed as argument to the untrusted `put` method, the values contained in the array are checked, and a leak of data is detected for the values which are typed as confidential (values without cryptographic content would produce a type error). This means that this method call will be interpreted as a sequence of *channel write* action on these values, as we will see in sub-section 3.3 below. Similar checks are applied to field access intructions.

In so doing, we define a single confidentiality domain, the limits of which are set at the level of classes (by the distinction made between trusted and non-trusted classes). This approach has appeared to be convenient in practice. One should also note that in order to perform the static analysis of a program, we not only have to supply a set of class-files (to be loaded and launched), but also additional information that specify trusted classes and initially secret data (designated as particular confidential static fields).

This model of data leakage has some similarities to the approach of [11] which considers the dual problem of access to sensitive references. This latter

method consists in using a dedicated type system which assigns a particular typing to references in order to detect access to sensitive references. This approach is applied to a ML-like language with references and functional closures. The type soundness properties of this language are formally investigated in [11] and they allow to use type inference algorithms in order to make the detection. Our approach to data leakage detection is similar in that it also assigns particular typing to sensitive values. But it differs from it in that it uses data flow analysis, rather than type inference, in order to propagate typing information. We have chosen a different approach because, in the case of JavaCard applets, which are somehow more restricted than ML programs, this choice leads to simpler static analysis algorithms.

3.3 Models of cryptographic knowledge evolution

In section 2, we referred to a general model of cryptographic knowledge we have designed for automated verification. We have now to incorporate this formal into a Java-oriented framework of static analysis.

First, we need to define a meaning for cryptographic notions such as messages or cryptographic knowledge within the concrete JVM model of execution. This task raises no particular difficulty. This is done by augmenting the JVM runtime structures with additional cryptographic information. To the JVM global configuration is associated a cryptographic environment in the same way as environments are associated to cryptographic programs in [8]. JVM numerical values are associated with messages showing their cryptographic content.

Second, we have to define what is the cryptographic counterpart of the instructions of this augmented JVM, that is to say how is propagated and modified cryptographic information along execution. This point is less easy, because in the JVM model of execution, there is no simple counterpart to the elementary cryptographic actions defined in our cryptographic model. We proceed as follows: message emission is modeled following the method described in sub-section 3.2 for detecting data leakage; other cryptographic actions such as encryption, decryption, nonce or key creation, and message construction or decomposition are associated to calls to particular library methods. The cryptographic meaning of these methods is hard-coded inside our analyzer as a set of specifications that must be supplied once and for all with each cryptographic API.

Finally, we have to abstract this augmented JVM model of execution into a static abstract model, well-adapted to automated analysis. Following the approach initiated in [8] and presented in section 2, this stage of abstraction poses not particular problem.

We achieve this in our abstract model by associating variables to numerical values having a crytographic content, and by augmenting the JVM configuration state with a set of cryptographic constraints. Admissible cryptographic environments are statically modeled as a set of cryptographic contraints as in the example of table 2 above. During the static analysis of execution, a fresh variable is associated to each new value. Cryptographic constraints on that variable

are generated and transformed along the execution of each instruction which happens to be interpreted as a cryptographic action.

We can get some insights into this approach by looking at the example of table 3.

initial assumptions
$\kappa_0 = \{is\_simple(k_1),\ \neg known(;k_1),\ is\_simple(k_2),\ \neg known(;k_2)\}$
read incomming message 0 iconst_5 1 istore_0 2 iload_0 3 invokestatic #8 ⟨Method int get(int)[]⟩ 6 astore_1
$\kappa_1 = \kappa_0 \cup \{is\_simple(k_1),\ \neg known(x;k_1),\ is\_simple(k2),$ $\neg known(x;k_2),\ \neg known(x;k_2),\ x \approx [x_1;x_2;x_3]\}$
decipher incomming message 7 aload_1 8 getstatic #9 ⟨Field int k1⟩ 11 invokestatic #5 ⟨Method int decrypt(int[], int)[]⟩ 14 astore_2
$\kappa_2 = \kappa_1 \cup \{is\_simple(u),\ Inv^a(u,k_1),\ x' \approx [x_1;x_2;x_3],$ $x' \approx \{z\}^a_u,\ z \approx [z_1;z_2;z_3]\}$
compose outgoing message 15 aload_2 16 getstatic #10 ⟨Field int k2⟩ 19 invokestatic #6 ⟨Method int encrypt(int[], int)[]⟩ 22 astore_3
$\kappa_3 = \kappa_2 \cup \{\ z' \approx [z_1;z_2;z_3],\ w \approx \{z'\}^a_{k_2},\ w' \approx [w_1;w_2;w_3]\ \}$
send outgoing message 23 aload_3 24 invokestatic #11 ⟨Method void put(int[])⟩ 27 return
$\kappa_4 = \{is\_simple(k_1),\ \neg known(x;k_1),\ is\_simple(k_2),\ \neg known(x;k_2),$ $is\_simple(u),\ Inv^a(u,k_1),\ x \approx [x_1;x_2;x_3],\ x' \approx [x_1;x_2;x_3],$ $x' \approx \{z\}^a_u,\ z \approx [z_1;z_2;z_3],\ z' \approx [z_1;z_2;z_3],\ w \approx \{z'\}^a_{k_2},$ $w \approx [w_1;w_2;w_3]\ \}$

Table 4. *knowledge evolution*

For instance, in table 3, at method call `Env.put(w)` near byte-code 23, the int array argument `w` is scanned for leakage detection, and for each leaking value (here, for each value), the associated variables are extracted and cryptographic *write channel* actions are generated. Another illustration is given by method call `encrypt(z, k2)` near bytecode 19: in that case, the values contained in the array argument `z` are fetched, the sets of their associated variables z_1, z_2 and z_3 are read, and the following cryptographic actions are generated: first, a tuple construction of the message to

be encrypted: $z' := [z_1; z_2; z_3]$; second, an encryption of that message (using a fresh variable w as result): $w{:=}\ encrypt^a_{k_2}\, z'$; and third, a tuple reading of the resulting message (using fresh variables w_1, w_2 and w_3 for the contained values): $w := [w_1; w_2; w_3]$. (The tupling and detupling actions are necessary in order to set the lengths of the arrays taken as argument and returned as result.) Table 4 shows the kind of cryptographic information that *StuPa* can deduce from the static analysis of the applet. In particular, we can check data confidentiality by tracking constraints $\neg known(...; ...)$ and $\neg old(...; ...)$. The example above shows that the keys k_1 and k_2 remain confidential up to the end of the execution of the protocol.

4 First implementation results

We have reviewed above the underlying formal models used as basis for the design of *StuPa*, an automated tool for the static verification of confidentiality of Java cryptographic applets.

At the current stage of development, a prototype is working, and is under experimentation. Up to now, it has been tested on small cryptographic applets such as one implementing the user's role in the Yalahom protocol described in [5]. On these examples, our tool yields good results in terms of leak detection and verification of confidentiality. Experiments on larger applets are on the way. The main task required for carrying them is to incorporate cryptographic specifications of the JavaCard API into the analyzer.

After some usage, the main limitations that appear are related to loss of information in the process of static analysis. Some limitations result from design decisions. As it is oriented toward the verification of JavaCard applets, our formal model doesn't support features of the Java language which are un-used in this Java subset, such as threads and reflection. Other sources of information loss are related to the existing trade-off between accuracy of analysis and its cost. For instance, we could get more information by unfolding iterations more. For those cases, it is possible to fine-tune the analyzer.

5 Conclusions

An innovative aspect of *StuPa* is undoubtedly the way it handles cryptographic knowledge: this tool rests on a formal framework that provides both an accurate description of cryptography, and a well-fitted model for automated analysis. It enables us to formally relate Java implementations of cryptographic mechanisms to their design requirements, and to verify their conformance in an automated way.

Much care has been taken in achieving a sound design, in accordance with the methodological principles of the abstract interpretation framework. In particular, formal foundations of the model of cryptographic knowledge have been extensively investigated ([8]).

StuPa sets the focus of static analysis on JavaCard-compliant applets, for which first experiments gave conclusive results. The small size of these applets

and the restricted subset Java they are implemented on mostly account for this success. For more general Java programs, more look is needed to deal with recursion (which poses no theoretical difficulties), thread and reflection (which are harder to deal with).

Nevertheless, even as limited to JavaCard, this approach seems promising, because there are today are high needs of formal verification on JavaCard applets, which have high security requirements.

References

1. Monniaux, D. Abstracting cryptographic protocols with tree automata. In *Static Analysis* (1999), A. Cortesi and G. Filé, Eds., vol. 1694 of *Lecture Notes in Computer Science*, Springer, pp. 149–163.
2. Bertelsen, P. Dynamic semantics of Java bytecode. In *Workshop on Principles of Abstract Machines* (Pisa, Italy, Sept. 1998).
3. Bolignano, D. An approach to the formal verification of cryptographic protocols. In *3rd ACM Conference on Computer and Communications Security* (New Delhi, India, Mar. 1996), C. Neuman, Ed., ACM Press, pp. 106–118.
4. Bolignano, D. Using abstract interpretation for the safe verification of security protocols. In *Electronic Notes in Theoretical Computer Science* (2000), M. M. Stephen Brookes, Achim Jung and A. Scedrov, Eds., vol. 20, Elsevier Science Publishers.), pp. 77–87.
5. Burrows, M., Abadi, M., and Needham, R. A logic of authentication. *ACM Transactions on Computer Systems 8*, 1 (Feb. 1990), 18–36.
6. Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles* (New York, NY, 1977), ACM, pp. 238–252.
7. Lowe, G. Casper: A compiler for the analysis of security protocols. In *10th IEEE Computer Security Foundations Workshop (CSFW '97)* (Washington - Brussels - Tokyo, June 1997), IEEE, pp. 18–30.
8. Elkadhi, N. Automatic verification of confidentiality properties of cryptographic programs. *Networking and Information Systems* (2001), pp. 4–15, Available at url: `http://www.epita.fr:8000/~el-kad_n/Hermes.ps`
9. Goubault-Larrecq. A method for automatic cryptographic protocol verification. In *SPDP: IEEE Symposium on Parallel and Distributed Processing* (2000), ACM Special Interest Group on Computer Architecture (SIGARCH), and IEEE Computer Society.
10. Sun Microsystems, Inc. *The JavaCard 2.2.1 Platform Specification.* Palo Alto/CA, USA, May 2000.
11. Leroy, X., and Rouaix, F. Security properties of typed applets. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California* (New York, NY, Jan. 1998), ACM, pp. 391–403.
12. Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, Jan. 1997.
13. Meadows, C. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security 1*, 1 (1992), 5–36.

6 Authors background and expectations

Nabil EL KADHI is professor of computer science at Paris EPITA/EPITECH. Among his research interests are the security of cryptographic protocol and the static analysis of program confidentiality.

He is the author of a Phd thesis on the "automatic verification of confidentiality properties of cryptographic programs".

Pierre BOURY is research engineer at GIE Dyade, a common BULL-INRIA subsidiary. His main research interests are formal methods applied to software engineering. He has recently been involved in a project of cryptographic protocols verification using the Coq proof assistant, and is today developing static analysis tools for the automated verification of Java cryptographic applets.

Our expectations toward the Workshop on Formal Techniques for Java Programs are to share ideas about formal verification of security properties in the domain of Java, and particularly to bring our contribution to the field of automated verification of cryptographic security.

Modular Specification of Frame Properties in JML

Peter Müller[1], Arnd Poetzsch-Heffter[1], and Gary T. Leavens[2]

[1] FernUniversität Hagen, 58084 Hagen, Germany
{Peter.Mueller, Arnd.Poetzsch-Heffter}@Fernuni-Hagen.de
[2] Iowa State University, Ames, Iowa, 50011-1040, USA
leavens@cs.iastate.edu

Abstract. We present a modular specification technique for frame properties. The technique uses modifies clauses and abstract fields with declared dependencies. Modularity is guaranteed by a programming model that restricts aliasing, and by modularity requirements for dependencies. For concreteness, we adapt this technique to the Java Modeling Language, JML.

1 Introduction

In an interface specification language, a *frame property* describes what locations a method may modify, and, implicitly, what locations it may not modify [BMR95]. This is often specified using a *modifies clause* [GHG+93,Win87].

We address three problems for specification and verification of frame properties: (1) Information hiding—The concrete (e.g., private) fields of a class should be hidden from its clients, even in specifications; yet the frame properties of (public) specifications must somehow permit those locations to be modified. (2) Extended state—When a subclass overrides a method, it may need to modify additional fields it declared; yet the demands of behavioral subtyping (e.g., [LW94,DL96]) would seem to prohibit modification of these additional fields [Lei98]. (3) Modularity—A modular solution to the frame problem must allow one to precisely specify the frame properties of methods and to verify their implementations, without knowing the context in which the methods will be used. However, in general one cannot know what locations might be found in a program that extends or uses a given class or interface.

Leino's work [Lei95] solves problems (1) and (2) by introducing abstract fields with explicitly declared dependencies and a refined semantics of modifies clauses (see below). This paper explains part of Müller's thesis [Mül01], which builds on Leino's work and provides a modular sound solution to problem (3).

1.1 Related Work

When modeling objects as records containing possibly abstract locations, one needs a way to specify the correspondence between abstract and concrete locations. To do this, Leino introduced depends and represents clauses [Lei95,Lei98].

A represents clause says how an abstract location's value is determined from various concrete locations. To a first approximation, a depends clause says what concrete locations are used to determine the abstract location's value. More precisely, a dependency declaration allows dependees to be modified whenever the abstract location is named in a modifies clause. Thus, in JML, "`depends absloc <- concloc`" says that `concloc` can be modified whenever `absloc` is modifiable.

To support the specification of extended state, a subtype may declare that an inherited abstract field depends on the fields it declares. Such dependencies allow overriding methods in subclasses to modify their extended state.

Leino and Nelson distinguish *static dependencies*, of the form "`depends f <- g`", and *dynamic dependencies*, of the form "`depends f <- p.g`", in which abstract field `f` depends on field `g` of the *pivot object* `p`. Leino and Nelson handle static and dynamic dependencies in different ways, that is, by different desugaring of modifies clauses, and different modularity rules. Although Müller's thesis [Mül01] treats both cases uniformly, in this paper, to avoid introducing additional concepts, we also distinguish them.

Leino and Nelson use scope-dependent depends relations [Lei95], which lead to a scope-dependent meaning of modifies clauses. Soundness is not immediate, because proofs for smaller scopes do not necessarily carry over to larger scopes; indeed, Leino and Nelson have not yet proved modular soundness of their technique for dynamic dependencies. See [Mül01, Section 5.5.1] for a detailed comparison between our approach and Leino's and Nelson's work.

1.2 Approach

To solve the first two problems described above, we follow Leino and Nelson [Lei95,LN00], using abstract fields and explicitly declared dependencies. We explain the ideas by applying them to the Java Modeling Language (JML) [LBR01,LBR99], which allows the specifier to declare abstract fields by using the modifier "`model`". JML also allows one to declare dependencies, although it does not yet incorporate the restrictions we propose here.

Our solution to the modularity problem entails three steps: (1) We define a programming model that hierarchically structures the object store into so-called contexts and restricts references between contexts [MPH00,MPH01,Mül01].

(2) Dependency declarations generate a theory for dependencies declared in a given set of modules. This depends relation does not specify dependencies for extensions to the given set of modules. Because of this underspecification one can only prove properties about a module that hold in well-formed extensions. Thus modular soundness is much simpler to prove than with a scope-dependent semantics of the modifies clause. The restricted programming model guarantees that this weaker semantics is still strong enough to verify method invocations.

(3) We impose three modularity requirements to restrict the permissible dependencies of abstract locations. These restrictions allow us to prove a modularity theorem that makes modular verification of frame properties possible.

A detailed presentation of these ideas, including all formalizations and proofs, but not their application to JML, is found in [Mül01].

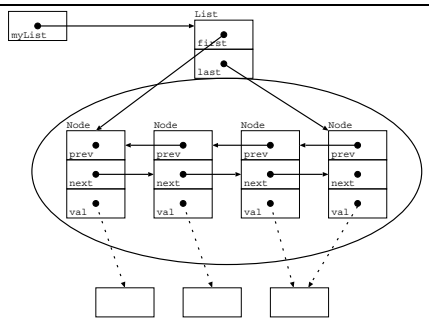


Fig. 1. Nodes in a context (the oval). The owner object sits atop the context it owns.

2 The Programming Model

To achieve modularity, dependencies must be controlled. There are two problems, both of which involve aliasing: (1) *Representation exposure* occurs when objects inside the representation of an object X may be referenced by objects outside of X's representation. (2) *Dependencies on argument objects* occur when an object X's abstract value is determined by the abstract values of objects, called *argument objects*, outside X's representation. Both problems allow modification of an object's abstract value in ways that cannot be controlled by its implementation.

To prevent representation exposure, the object store is structured into a hierarchy of contexts. *Contexts* are disjoint groups of objects. There is a root context. All other contexts have an *owner object* in their parent context. Aliasing is controlled by the following invariant: Every reference chain from objects in the root context to an object in a context C passes through C's owner. Thus, an owner object can control access to objects in its context. This structure of the object store is called the *ownership model* [CPN98].

The ownership model is not sufficient to prevent dependencies on argument objects because it allows objects inside a context to reference argument objects in ancestor contexts. We refined the ownership model in two ways [MPH01,Mül01]: (a) references to argument objects are made explicit by marking them `readonly`, and (b) readonly references can point to any object, not only to objects in ancestor contexts. Access via readonly references is restricted to reading operations without side-effects. This *refined ownership model* is more general than the original one. In this refined model, we prevent dependencies on argument objects by forbidding dependencies via readonly references.

Figure 1 illustrates our *refined ownership model*. The nodes of a linked list are contained in a context owned by the list header. The objects stored in the list are outside the context and are referenced readonly (dashed arrows). Consequently, abstract fields of the list must not depend on fields of these objects.

To enforce the refined ownership model's invariant, we use the universe type system [MPH01,Mül01]. Besides tagging types as readonly, this type system also

distinguishes between references that remain inside a context and references to objects that belong to the *descendant context* owned by the `this`-object. References of the latter kind are tagged with the keyword `rep` [CPN98].

3 Specification of frame properties in JML

3.1 Data abstraction in JML

Data abstractions in JML are specified using abstract locations, i.e., model fields. For example, consider the specifications of `List` in Figure 2 and `Node` in Figure 3.

```
//@ model import edu.iastate.cs.jml.models.*;
public abstract class List {
    //@ public model non_null JMLObjectSequence listValue;
    protected /*% rep %*/ Node first, last;
    //@ protected depends listValue <- first, first.values, last;
    /*@ protected represents listValue <-
      @    (first == null ? new JMLObjectSequence() : first.values);  @*/

    /*@ public normal_behavior
      @   requires o != null;
      @   modifies listValue;
      @   ensures listValue.equals(\old(listValue.insertBack(o)));  @*/
    public void append(/*% readonly %*/ Object o) {
        if (last==null) {
            last = new /*% rep %*/ Node(null, null, o);
            first = last;
        } else {
            last.next = new /*% rep %*/ Node(null, last, o);
            last = last.next;
        }
    }

    /* ... */
}
```

Fig. 2. A JML specification of the Java class `List`, of doubly-linked lists.

The class `List` declares a public model field `listValue`, which describes the abstract value of a `List` object. In the class `Node`, the model field `values` forms part of the abstract value of `Node` objects. In JML, method specifications precede the method header, preconditions are introduced by the keyword `requires` and postconditions by the keyword `ensures`. For example, in the specification of `List`'s method `append`, the postcondition describes the abstract effect of `append` on the model field `listValue`.

```
//@ model import edu.iastate.cs.jml.models.*;
public class Node {
    //@ public model non_null JMLObjectSequence values;
    public Node next, prev;
    public /*% readonly %*/ Object val;
    //@ public depends values <- next, next.values, prev, val;
    /*@ public represents values <-
      @   (next == null ? new JMLObjectSequence(val)
      @                 : next.values.insertFront(val));  @*/

    Node(Node nextp, Node prevp, /*% readonly %*/ Object valp) {
        next = nextp;  prev = prevp;  val  = valp;
    }
}
```

Fig. 3. The JML specification of the Java class `Node`.

3.2 Explicit dependencies in JML

Although Müller's thesis [Mül01] uses a quite general form of dependencies, we use a syntax for depends clauses like that in Leino's thesis [Lei95]. Besides simplicity, this syntax also permits the restrictions discussed in Section 4 to be statically checked easily. We leave extensions to this syntax as future work.

For example, in the class `List`, the model field `listValue` is represented by a sequence determined by `first` and `first.values`. Hence `listValue` is also declared to depend on these fields. Although the represents clause for `List` does not use the field `last`, that field is listed in the depends clause, to permit it to be modified whenever `listValue` is modifiable. Similarly, in class `Node`, the model field `values` depends on `next`, `next.values`, `prev`, and `val`.

3.3 Modifies Clauses in JML

An example of a modifies clause in JML appears in the specification of `List`'s `append` method. It says that the method may modify `listValue`.

The semantics of the modifies clause is that all relevant locations that either are named in the clause or on which such locations depend may be modified. A location is *relevant* to the execution of a non-static method m if it is either in the context that contains m's receiver or a descendant context of the one that contains m's receiver. For example, if `myList` is an object of type `List`, then for the call `myList.append(o)`, the relevant locations are those in the context that contains `myList`, and locations in descendant contexts. Since the field `first` in `List` is declared using the keyword `rep`, the object `myList.first` points to is in the context owned by `myList` (see Figure 1), which is thus a descendant context of the context that contains `myList`. Since the `next` fields of `Node` objects are not

```
//@ model import edu.iastate.cs.jml.models.*;
public abstract class Set {
    //@ public model non_null JMLObjectSet setValue;
    protected /*% rep %*/ /*@ non_null @*/ List theList;
    //@ protected depends setValue <- theList, theList.listValue;
    /*@ protected represents setValue \such_that
      @   (\forall Object o; o != null;
      @       theList.listValue.has(o) <==> setValue.has(o));  @*/

    /*@ public normal_behavior
      @   requires o != null;
      @   modifies setValue;
      @   ensures setValue.has(o);  @*/
    public void insert(/*% readonly %*/ Object o) {
        if (!theList.contains(o)) { theList.append(o); }
    }
}
```

Fig. 4. The JML specification of the Java class `Set`.

declared using `rep`, the objects reachable via `next` are all in the same context. It follows that all the nodes are in the context owned by `myList`, and hence that the fields of these nodes are also relevant locations. That is, the call may modify `myList.first`, `myList.first.values`, `myList.last`, and all the fields of the nodes reachable from `myList.first` via the `next` field.

To explore the modularity consequences of this semantics, consider an extended program, in which the type `List` is used to implement the type `Set`, specified in Figure 4. `Set`'s model field `setValue` depends on its concrete field `theList` and `theList.listValue`. Since the specification of `Set`'s `insert` method lists `setValue` in its modifies clause, a call such as `mySet.insert(o)` may modify `mySet.setValue` and all the other relevant locations on which it depends. Since `theList` is declared using `rep`, it is in the context owned by `mySet`, and so is in a descendant context of the one containing `mySet` (see Figure 5). Therefore `mySet.theList` is a relevant location, and since it is also a dependee, it can be modified. Similarly, `mySet.theList.listValue`, `mySet.theList.first`, and the fields of the nodes are relevant, and so these dependees can be modified.

The modularity of the semantics is shown by the call `theList.append(o)` in `Set`'s `insert` method. How does the semantics allow `List`'s `append` method to modify the set's model field `setValue`, which it does when it modifies the abstract value of `theList`? The semantics allows this because it underspecifies the locations that `append` can modify, since it only describes the modification of relevant locations, and `setValue` is not relevant for the call `theList.append(o)`. The reason for this is that a context's owner is not contained in the context it owns, and `theList` is in the context owned by the receiver in `Set`'s `insert`

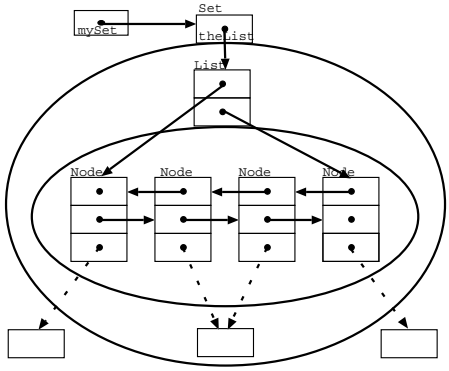


Fig. 5. Object Structure for a `Set` object.

method (see Figure 5). Hence in `Set`'s `insert` method, `this.setValue` is not a relevant location for the call to `theList.append(o)`.

Responsibility for verifying frame properties is divided. A method's implementor is responsible for the locations relevant to its executions, as specified in its modifies clause, and the method's caller is responsible for other locations. For example, `append`'s implementor is responsible for verifying the frame properties in its modifies clause. When verifying the call to `append` in `Set`'s `insert` method, one uses `append`'s modifies clause and `Set`'s depends clauses to reason about modification of `Set`'s fields `theList` and `setValue`.

4 Modularity and Dependencies

To achieve modularity, we impose three requirements on dependencies:

Locality Requirement: Abstractions of an object X can only depend on locations in the context that contains X or its descendants. That is, they may depend on locations in X's representation, but not on argument objects.

Authenticity Requirement: The declaration of an abstract location L in a context C must be visible in every scope that contains a method m that could—if invoked on a target object in C—modify L. Thus the verifier of m can determine all relevant locations that m might modify.

Visibility Requirement: Whenever two locations are declared in a scope S, the dependencies in S must allow one to determine whether these locations depend on each other or not.

We enforce these requirements by statically checking the following rules for single depends clauses of the form "`depends` f `<-` g" or "`depends` f `<-` $p.g$".

Locality Rule: For dynamic dependencies, the pivot field must not hold a read-only reference; that is, p must not be of a `readonly` type.

Authenticity Rule: For static dependencies and for dynamic dependencies where the pivot field is not of a `rep` type, f must be declared in the scope of g. For dynamic dependencies where the pivot field is of a `rep` type, f must be declared in the scope of the owner type of p. In most implementations such as in our examples, the *owner type* of a field is its declaration type (see [Mül01] for a precise definition).

Visibility Rule: Static and dynamic dependencies where the pivot field is not of a `rep` type must be declared in the scope of g. Dynamic dependencies where the pivot field is of a `rep` type must be declared in the scope of p's owner type.

To verify frame properties of a method m, one has to prove that m leaves all relevant locations that are not covered by m's modifies clause unchanged. This proof obligation can be shown for those locations that are declared in the scope of m by referring to their representations and dependencies. For all other relevant locations, the locality and authenticity requirements guarantee that they are not modified by m, as stated by the following modularity theorem:

A method m can only modify relevant locations that are declared in m's scope.

A sketch of this theorem's proof is contained in the appendix. A formalization of the theorem and the full proof can be found in [Mül01]. The modularity theorem's proof shows that the modularity requirements in combination with the universe programming model are strong enough to enable modular verification of frame properties. Similar requirements are used in [LN00].

5 Conclusions

We extended the Java Modeling Language by constructs to specify frame properties in a modular way. The extension is based on a refined ownership model: The programmer can hierarchically structure the object store into contexts to which only designated owner objects have direct access. All other references crossing context boundaries have to be declared readonly. The ownership model is enforced by the universe type system. It provides the basis to refine the semantics of the modifies clause and to define context conditions that guarantee the modularity of specification and verification of frame properties.

The JML extensions are based on a more general framework that was developed for modular verification of Java programs [Mül01]. In that work, these ideas are also applied to the modular treatment of class invariants, by considering invariants to be boolean-valued abstract fields. Thus these ideas also lead to modular specification and verification of invariants.

Although our technique can express common implementation patterns such as containers with iterators and mutually recursive types [Mül01], some extensions might be useful in practice. For instance, unique variables would allow objects to migrate from one context to another, and less restrictive modularity rules would provide better support for inheritance [Mül01]. We leave such extensions for future work.

Acknowledgments

The work of Leavens was supported in part by the US NSF under grant CCR-9803843, and was done while Leavens was visiting the University of Iowa.

References

[BMR95] Alex Borgida, John Mylopoulos, and Rayomnd Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[CNP01] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for containment. In *European Conference on Object-Oriented Programming, ECOOP 2001*, Lecture Notes in Computer Science. Springer-Verlag, 2001. (to appear).

[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.

[DEJ+00] Sophia Drossopoulou, Susan Eisenbach, Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter. Formal techniques for Java programs. In Jacques Malenfant, Sabine Moisan, and Ana Moreira, editors, *Object-Oriented Technology. ECOOP 2000 Workshop Reader*, volume 1964 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2000.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[GHG+93] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, NY, 1993.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde 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.

[LBR01] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06m, Iowa State University, Department of Computer Science, February 2001. See `www.cs.iastate.edu/~leavens/JML.html`.

[Lei95] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[Lei98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[LH94] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies.* The Object-Oriented Series. Prentice Hall, New York, NY, 1994.

[LN00] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. Technical Report 160, Compaq Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, 2000.

[LW94] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[MPH00] Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. Published in [DEJ+00]., 2000.

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, FernUniversität Hagen, 2001.

[Mül01] Peter Müller. *Modular Specification and Verification of Object-Oriented programs*. PhD thesis, FernUniversität Hagen, Germany, March 2001.

[SBC92] Susan Stepney, Rosalind Barden, and David Cooper, editors. *Object Orientation in Z*. Workshops in Computing. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[Win87] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

A Sketch of the Modularity Theorem's Proof

In the following, we sketch the field update case of the proof of the modularity theorem from Section 4. The proof for method invocations is similar.

Proof Sketch. Let m be executed in context C (i.e., the receiver is in C). If m updates $Y.g$, the universe type system guarantees that Y is in C or in one of C's immediate descendants. Consider an abstract location $X.f$ that is relevant for m. If $X.f$ does not depend on $Y.g$, $X.f$ is not affected by updates of $Y.g$. Otherwise, we show that f is declared in m's scope:

Case 1: Y is in C. If $X.f$ is relevant for m, then by the locality rule X is in C. Thus, X and Y are in the same context, and the authenticity rule ensures that f is declared in g's scope. Since g is accessible in m, f is in m's scope.

Case 2: Y is in an immediately-descendant context D of C. Due to locality, X is in D or in C. The former case is analogous to Case 1. In the latter case: a dynamic dependency must be involved with a pivot field p of a `rep` type, the owner type of p is in the scope of m (by the universe type system), and f is declared in the scope of p's owner type (by the authenticity rule). Thus, f is declared in m's scope.

Compositional specification and verification of control flow based security properties of multi-application programs

Gilles Barthe[1], Dilian Gurov[2], Marieke Huisman[1]

[1] INRIA Sophia-Antipolis, France
{Gilles.Barthe,Marieke.Huisman}@sophia.inria.fr
[2] SICS, Sweden dilian@sics.se

Abstract. Jensen *et al.* present a simple and elegant program model, within a specification and verification framework for checking control flow based security properties by model checking techniques. We generalise this model and framework to allow for compositional specification and verification of security properties of multi-application programs. The framework contains a program model for multi-application programs, and a temporal logic to specify security properties about such programs.

1 Introduction

Formal verification of security properties becomes more and more important. An important and interesting class of security properties are control flow based security properties. Jensen *et al.* [7] present a simple and elegant program model which is used to check these kind of properties (using finite-state model checking). This program model is language independent, but it can easily be instantiated for Java or JavaCard. With the program model, also a language to specify security properties is presented. A drawback of this approach is that to check "real world" programs, the state space can become very large.

Verification of security properties is in particular important for the new generation multi-application smart cards. Typical for such multi-application smart cards is that applets can be loaded post-issuance, *i.e.* after initialisation of the card. Therefore, one would like to do the verifications in a compositional way, stating which properties should be satisfied by the components of the system, to ensure the global correctness of the system. When issuing a new applet on the card, one has to check that this new applet satisfies these required properties, in order to know that other applets can safely cooperate with it.

In this paper, we present a framework for compositional verification of multi-application programs, which is a generalisation of the model presented by Jensen *et al.* The framework, which consists of a program model and a specification language, is language-independent, but can easily be instantiated for JavaCard (as in [7]). The framework enables specification and reasoning in a compositional style, and is thus more suited to verify security properties for multi-application smart cards. The program model is designed to be as abstract as possible, while

it still accurately describes the method call behaviour. Further we propose a set of temporal logic patterns which can be used to specify properties over these programs. The temporal logic patterns can be translated into different logics, including the modal μ-calculus [8]. For this logic, a proof system is under development which will allow one to decompose system properties into properties over the individual applets. This verification method fits in well with the nature of smart cards, where applets can be loaded post-issuance, and it makes verification more manageable by reducing the state space. This paper focuses on appropriate specifications of multi-application programs, and on how to specify properties over such programs in such a way that compositional verification can be achieved.

The model and the logic enable us to reason about smart cards at a behavioural level, *i.e.* at the level of method calls. We feel that this is the right level to talk about applet interaction: for the global correctness of the system it is important to know that the components have a certain interface behaviour, and it does not matter how this behaviour is achieved. Only when showing that an applet satisfies the required properties, one has to look at its implementation.

Example: electronic purse To illustrate our approach we discuss an example from [1], which presents a typical verification problem for smart cards. An electronic purse is presented, which contains three applets: a Purse applet `P`, and two loyalty applets: AirFrance `AF`, and RentACar `RaC`. The owner of an electronic purse smart card can decide to join a loyalty program of some company, and load the appropriate applet on his card. The loyalty applets need to be informed about the purchases done with the card, in order to compute the loyalty points.

For efficiency reasons, the electronic purse keeps a log table of bounded size of all credit and debit transactions, and the loyalty applets can request the information stored in this table. For example, if the user wishes to know how many loyalty points he/she has, the loyalty applet will update its local balance first, before returning an answer. Updating the local balance of a loyalty applet consists of two phases: asking the entries of the log table of the purse, and asking the balances of loyalty partners (to compute an extended balance).

In order to ensure that loyalties do not miss any of the logged transactions (if the log table is full, entries will be replaced by new transactions), they can subscribe to the so-called `logFull` service. This service signals all subscribed applets that the log will be emptied soon, and that they should thus update their local balance. In the example, the AirFrance applet is subscribed to this service, but the RentACar applet is not. However, RentACar might be able to implicitly deduce that the log is full, from the fact that AirFrance asks RentACar for its balance information, every time AirFrance gets the `logFull` message. A malicious implementation of the RentACar loyalty applet might therefore request the information stored in the log table, before returning the value of its local balance to AirFrance. This is unwanted, because it might be the case that applets pay for the `logFull` service, and the owner of the purse applet would not want other applets to get this information for free.

Thus, one would like to specify and verify that only applets that are subscribed to the `logFull` service update their balance, until the log is emptied; in particular one would like to specify that the bad scenario, depicted as a message sequence chart in Fig. 1 (where the solid lines indicate method invocations and the dashed lines indicate method returns) can not happen.

The property depicted in Fig 1 can be formulated as: *an invocation of* `AF.logFull` *in the AirFrance applet should not trigger an invocation of* `P.getTrs` *in the Purse applet by the RentACar applet* `RaC`. Below, in Section 2, we will specify this property formally, and we will also show that to establish that this property holds for the system, it is sufficient to show that `AF.logFull` only calls `P.getTrs` and `RaC.getBalance`, while these methods do not call other methods (hence `RaC` never calls `P.getTrs` when `AF.logFull` is called).

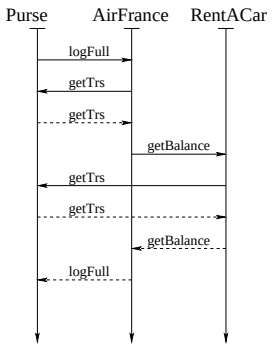


Fig. 1. Electronic purse: bad scenario

The remainder of this paper is organised as follows. Section 2 introduces the temporal logic patterns and show how these are used to specify properties. It also discusses the decomposition theorem. Section 3 discusses the compositional program model, which extends the model of Jensen *et al.* Finally, Section 4 concludes and discusses future work. Throughout the paper, the case study described above will serve as a motivating example.

2 Specifying properties for multi-application programs

Typical properties that are of interest for multi-application programs can often be expressed as temporal logic formulae, stating *e.g.* that a particular event only occurs after some other event has happened. We take the following approach to specification. First we specify the global property (as a temporal logic formula) that should be satisfied by the program. Then we specify which properties should hold for the individual applets (or components) of the program, and we prove formally that if the components satisfy these properties, the global program satisfies the global specification.

The specifications of the global system and the applets are described using temporal specifications patterns, following the approach taken for the Bandera specification language [3]. These patterns have proven useful to specify properties, and can easily be translated into formulae in a particular logic. Typical example patterns that we use are ALWAYS ϕ, WITHIN m ϕ, where m is a method, and A CALLS $\mathcal{M}$, where A is an applet, and $\mathcal{M}$ a set of methods. The temporal logic framework is rich enough to express security properties like the absence of bad scenarios as illustrated above, and it allows a wide range of other important behavioural correctness properties of multi-application programs to be specified.

Using these temporal logic patterns we can specify correctness properties for the electronic purse. As mentioned above we want that an invocation of

`AF.logFull` in the purse does not trigger a call from `RaC` to `P.getTrs`. Formally, we can specify this as $\mathsf{SPEC}_{\mathtt{EP}}(\mathtt{P}, \mathtt{AF}, \mathtt{RaC})$, where:

$$\begin{array}{l}\mathsf{SPEC}_{\mathtt{EP}}(X, Y, Z) \stackrel{\text{def}}{=} \\ \quad \mathsf{ALWAYS}\,. \\ \quad\quad \mathsf{WITHIN}\; Y.\mathtt{logFull}\,. \\ \quad\quad\quad \mathsf{NOT}(Z\;\mathsf{CALLS}\;\{X.\mathtt{getTrs}\})\end{array}$$

where X, Y, Z are variables ranging over applets. This specification states that for any (reachable) state in which the method $Y.\mathtt{logFull}$ has been invoked, but not been finished, there should be no call from the Z applet to $X.\mathtt{getTrs}$.

Based on this specification, we give specifications per applet in such a way that it is sufficient to prove for each applet that it satisfies its local specification, in order to deduce that the global system satisfies the global specification. Finding the local specification requires insight into the system. We specify the purse applet as $\mathsf{SPEC}_{\mathtt{P}}(\mathtt{P})$, the AirFrance applet as $\mathsf{SPEC}_{\mathtt{AF}}(\mathtt{AF}, \mathtt{P}, \mathtt{RaC})$, and the RentACar applet as $\mathsf{SPEC}_{\mathtt{RaC}}(\mathtt{RaC})$, where $\mathsf{SPEC}_{\mathtt{P}}$, $\mathsf{SPEC}_{\mathtt{AF}}$, and $\mathsf{SPEC}_{\mathtt{RaC}}$ are defined as follows.

$$\begin{array}{l}\mathsf{SPEC}_{\mathtt{P}}(X) \stackrel{\text{def}}{=} \\ \quad \mathsf{ALWAYS}\,. \\ \quad\quad \mathsf{WITHIN}\;(X.\mathtt{getTrs})\,. \\ \quad\quad\quad X\;\mathsf{CALLS}\;\{\}\end{array}$$

$$\begin{array}{l}\mathsf{SPEC}_{\mathtt{AF}}(Y, X, Z) \stackrel{\text{def}}{=} \\ \quad \mathsf{ALWAYS}\,. \\ \quad\quad \mathsf{WITHIN}\;(Y.\mathtt{logFull})\,. \\ \quad\quad\quad Y\;\mathsf{CALLS}\;\{X.\mathtt{getTrs}, Z.\mathtt{getBalance}\}\end{array}$$

$$\begin{array}{l}\mathsf{SPEC}_{\mathtt{RaC}}(Z) \stackrel{\text{def}}{=} \\ \quad \mathsf{ALWAYS}\,. \\ \quad\quad \mathsf{WITHIN}\;(Z.\mathtt{getBalance})\,. \\ \quad\quad\quad Z\;\mathsf{CALLS}\;\{\}\end{array}$$

The specification for the purse applet states that the method $X.\mathtt{getTrs}$ does not invoke any other method. The specification for AirFrance specifies which methods are invoked by $Y.\mathtt{logFull}$. The specification for RentACar specifies that $Z.\mathtt{getBalance}$ should not invoke any other method. Notice that these specifications do not fully specify the behaviour of the applets, they only describe the necessary behaviour in order to satisfy the global property.

Given the global specification $\mathsf{SPEC}_{\mathtt{EP}}$ for the electronic purse, and given the specifications for the individual applets `P`, `AF` and `RaC`, we establish the following theorem, presented as a Gentzen-style sequent, where free variables are (implicitly) universally quantified (where $X : \phi$ is an assertion meaning that applet X satisfies property ϕ).

$$X\colon \mathsf{SPEC}_{\mathtt{P}}(X), Y\colon \mathsf{SPEC}_{\mathtt{AF}}(Y, X, Z), Z\colon \mathsf{SPEC}_{\mathtt{RaC}}(Z) \;\vdash\; X \mid Y \mid Z\colon \mathsf{SPEC}_{\mathtt{EP}}(X, Y, Z)$$

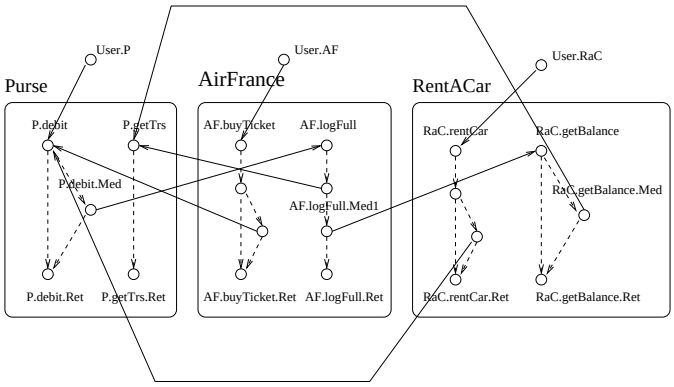


Fig. 2. Compositional model for the purse

Using this theorem one can reduce the proof of the global correctness assertion $\mathtt{P} \mid \mathtt{AF} \mid \mathtt{RaC}\colon \mathsf{SPEC}_{\mathtt{EP}}(\mathtt{P}, \mathtt{AF}, \mathtt{RaC})$ to proving the local correctness assertions $\mathtt{P}\colon \mathsf{SPEC}_{\mathtt{P}}(\mathtt{P}, \mathtt{AF})$, $\mathtt{AF}\colon \mathsf{SPEC}_{\mathtt{AF}}(\mathtt{AF}, \mathtt{P}, \mathtt{RaC})$ and $\mathtt{RaC}\colon \mathsf{SPEC}_{\mathtt{RaC}}(\mathtt{RaC})$ of the individual applets. Notice that we thus have two different kind of verification tasks in our framework, namely model-checking the local properties of the individual applets, and proving property decompositions correct. The use of general temporal logic patterns allows us to use different verification techniques. For example, we can model check the "local" applet properties, by translating the specifications into CTL (*e.g.* as input for NuSMV [2]) or LTL (*e.g.* as input for SPIN [6]), while we can use the modal μ-calculus [8] to prove the correctness of the property decomposition.

3 A program model for multi-application programs

To verify the properties as described above, we need a formal model, representing multi-application programs, with a formal (operational) semantics. This model is designed in such a way that it is suited for compositional verification. Based on the approach taken by Jensen *et al.* [7], we model a program as a transfer graph, modelling intra-procedural control flow, and a call graph, modelling method calls. A special set of vertices is identified, which are the return vertices, where a method hands back control to the caller. A function $\alpha\colon V \rightharpoonup A$ exists, which attributes vertices to applets. This is a partial function, as we allow vertices that do not belong to applets; these are the external vertices that model the environment. To illustrate the model, Fig. 2 shows the electronic purse formalised in this way. A suggestive naming and notation is used, to attribute vertices to applets (the function α), and to suggest the control flow in the methods. For clarity of presentation, in the picture we did not name all the intermediate

vertices. The dashed arrows are edges in the transfer graph, the solid arrows are edges in the call graph.

Every applet has a local state, which is a list of pairs of vertices, representing the control stack in the current program point. For example, given an applet a with local vertices v_2 and v_5, its local state $(v_1, v_2) \cdot (v_2, v_3) \cdot (v_4, v_5)$ can be interpreted as: vertex v_1 (which is external to a) invoked a vertex in a, during whose execution v_2 is reached. Next, v_2 invoked the vertex v_3 in some other applet. Execution continued in this other applet, but eventually somewhere in some applet a vertex v_4 is reached, which invoked a vertex in a again, and during the execution of this vertex, the vertex v_5 is reached.

The operational semantics of individual applets as well as of sets of applets is given compositionally, in terms of labelled transition systems induced by a set of transition rules. The latter are grouped in two parts: transition rules defining the behaviour of individual applets (that is, singleton applet sets), and transition rules for combining behaviours of applet sets.

The transition labels are denoting method invocations and returns. We distinguish between perfect and imperfect actions, the former being either intra-procedural control flow actions (left unlabelled) or method invocations/returns internal to a given applet set (labelled with call and ret, respectively), and the latter being method invocations/returns involving vertices external to the applet set (labelled with call?/ret? for input and call!/ret! for output action, respectively). Imperfect actions can form the corresponding perfect actions by synchronisation in the global trace of the system (thus leaving only the labels call and ret).

Applet transition rules Figure 3 gives the transition rules per applet. In this figure the applet name a is fixed, and π denotes the local state of applet a. We use $v_1 \rightarrow^T v_2$ to denote edges in the transfer graph, modelling intra-procedural control flow, and $v_1 \rightarrow^C v_2$ to denote edges in the call graph, respectively.

We use an applet-state predicate active_a and vertex predicates local_a and return_a, which are defined as follows.

$$\begin{aligned}
\mathsf{active}_a(\pi) &\stackrel{\text{def}}{=} \exists \pi', v, v'.(\pi = \pi' \cdot (v, v')) \ \wedge \ \mathsf{local}_a(v') \\
\mathsf{local}_a(v) &\stackrel{\text{def}}{=} \alpha(v) \in \mathsf{dom}(\alpha) \ \wedge \ \alpha(v) = a \\
\mathsf{return}_a(v) &\stackrel{\text{def}}{=} v \in V^R \ \wedge \ \mathsf{local}_a(v)
\end{aligned}$$

Thus, an applet is active if the second vertex in the last pair of π is local to this applet.

The first three rules describe transitions local to the applet. The rules send call and receive call describe the state transitions when a call to a different applet is made (either from an external vertex, or from applet to applet). Similarly, the rules send return and receive return describe the state transitions if a call over method borders is completed. The receive return transition is enabled if the return is sent by the same applet as the one the corresponding call was send to, there are no requirements on the local state of this applet. This is in accordance with the restrictions on compositional reasoning.

$$[\text{local call}]\ \frac{v_1 \rightarrow^C v_2 \qquad \mathsf{local}_a(v_1) \qquad \mathsf{local}_a(v_2)}{\pi \cdot (v, v_1) \xrightarrow{v_1\ \mathsf{call}\ v_2} \pi \cdot (v, v_1) \cdot (v_1, v_2)}$$

$$[\text{local return}]\ \frac{v_1 \rightarrow^T_a v_2 \qquad \mathsf{return}_a(v_3)}{\pi \cdot (v, v_1) \cdot (v_1, v_3) \xrightarrow{v_3\ \mathsf{ret}\ v_1} \pi \cdot (v, v_2)}$$

$$[\text{local transfer}]\ \frac{v_1 \rightarrow^T_a v_2 \qquad v_1 \not\rightarrow_C}{\pi \cdot (v, v_1) \longrightarrow \pi \cdot (v, v_2)}$$

$$[\text{send call}]\ \frac{v_1 \rightarrow^C v_2 \qquad \mathsf{local}_a(v_1) \qquad \neg\mathsf{local}_a(v_2)}{\pi \cdot (v, v_1) \xrightarrow{v_1\ \mathsf{call!}\ v_2} \pi \cdot (v, v_1) \cdot (v_1, v_2)}$$

$$[\text{receive call}]\ \frac{v_1 \rightarrow^C v_2 \qquad \neg\mathsf{local}_a(v_1) \qquad \mathsf{local}_a(v_2) \qquad \neg\mathsf{active}_a(\pi)}{\pi \xrightarrow{v_1\ \mathsf{call?}\ v_2} \pi \cdot (v_1, v_2)}$$

$$[\text{send return}]\ \frac{\mathsf{return}_a(v_2) \qquad \neg\mathsf{local}_a(v_1)}{\pi \cdot (v_1, v_2) \xrightarrow{v_2\ \mathsf{ret!}\ v_1} \pi}$$

$$[\text{receive return}]\ \frac{v_1 \rightarrow^T_a v_2 \qquad \neg\mathsf{local}_a(v_3) \qquad \alpha(v_3) = \alpha(v_4)}{\pi \cdot (v, v_1) \cdot (v_1, v_3) \xrightarrow{v_4\ \mathsf{ret?}\ v_1} \pi \cdot (v, v_2)}$$

Fig. 3. Applet transition rules

In all rules except receive call it is implicit whether applet a is active or not. The two receive rules are the only two rules that can apply when applet a is not active. Notice how the active applet changes when methods are called and returned: the applet that sends a call has to be active to be able to make the call, and as a result becomes inactive, while the applet that receives the call becomes active. A similar thing applies to the return transitions.

Using these transition rules, one can derive for example the trace fragment in Fig. 4 for the AirFrance applet.

Composing applets Applets can be composed into larger system components. Composite states are sets of local states, with the following restrictions:

- at most one applet is active,
- at most one external vertex is mentioned in the trace, and in this case this vertex occurs as the first component of the first pair of the trace.

The last condition ensures that we can only get single execution threads (which is for the time being appropriate for JavaCard). Computations are always started by the environment, they do not begin spontaneously. External vertices can only invoke methods, and wait for their return. By requiring that external vertices only occur at the beginning of the trace, we enforce that the environment only invokes a method in an applet, if there is no active applet. If necessary this

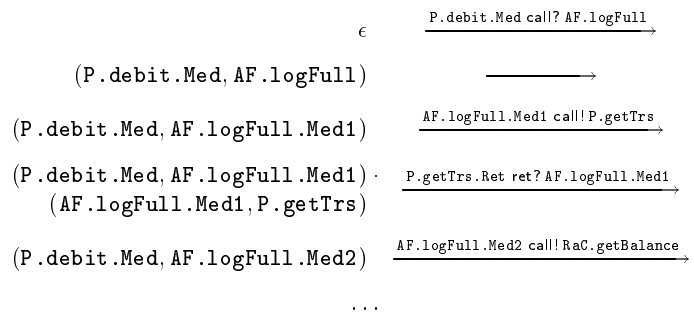


Fig. 4. Local trace AirFrance applet

$$[\mathsf{synchro}]\ \frac{\mathcal{A}_1 \xrightarrow{v_1\ \ell?\ v_2} \mathcal{A}_1' \qquad \mathcal{A}_2 \xrightarrow{v_1\ \ell!\ v_2} \mathcal{A}_2'}{\mathcal{A}_1 \mid \mathcal{A}_2 \xrightarrow{v_1\ \ell\ v_2} \mathcal{A}_1' \mid \mathcal{A}_2'}\ \ell \in \mathsf{call}, \mathsf{ret}$$

$$[\mathsf{propagation}]\ \frac{\mathcal{A}_1 \xrightarrow{\ell} \mathcal{A}_1'}{\mathcal{A}_1 \mid \mathcal{A}_2 \xrightarrow{\ell} \mathcal{A}_1' \mid \mathcal{A}_2}\ \mathsf{perfect}(\ell) \text{ or } \neg\mathsf{involved}_{\mathcal{A}_2}(\ell)$$

Fig. 5. Transition rules for composite states

restriction can be relaxed to allow multi-threading. For the global state, *i.e.* the set of all applets, we strengthen the last restriction and require that the first component in the first pair of the trace is an external vertex. In this way, we ensure that it is always an external vertex that triggers the global execution.

The way the labelled transitions of composite states are induced by the labelled transitions of its subsets is defined through the rules given in Fig. 5. In these rules $\mathcal{A}_1$ and $\mathcal{A}_2$ denote disjoint sets of applet states. Symmetric counterparts exist for both rules. The transition rule synchro applies when both sets of applets can do a transition, labelled with an imperfect action, and when these imperfect actions can synchronise into one perfect action (a perfect action is labelled with call or ret only, it does not contain tags ? or !). This results in a single transition in the composite system, labelled with the corresponding perfect action. The propagation transition rule applies when one set of applets can do a transition, labelled with ℓ, such that ℓ is a perfect action, or ℓ does not involve vertices from applets in the other set. The notion of being involved is defined as follows (where $\mathcal{A}$ is a set of applets).

$$\mathsf{involved}_{\mathcal{A}}(\rho) \stackrel{\text{def}}{=} \exists v1,\ v2 \in V. \exists \ell \in \{\mathsf{call}, \mathsf{ret}\}.\ (\rho = v_1\ \ell?\ v_2 \vee \rho = v_1\ \ell!\ v_2) \wedge (\alpha(v_1) \in \mathcal{A}\ \vee \alpha(v_2) \in \mathcal{A})$$

Using these transition rules, one can find *e.g.* the global trace fragment for the electronic purse, depicted in Fig. 6.

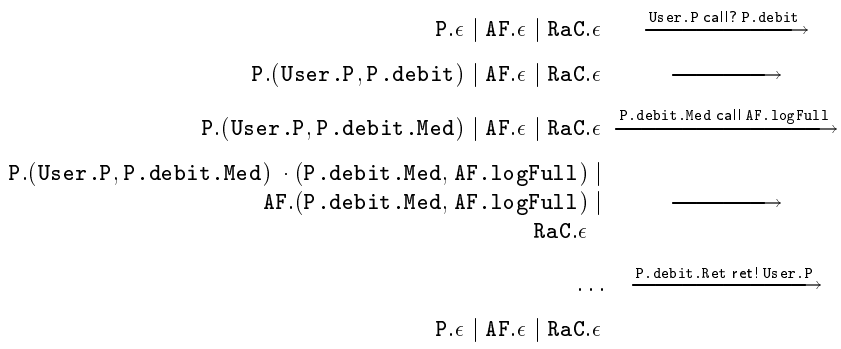


Fig. 6. Fragment of the global trace

4 Conclusions & future work

We have outlined a compositional program model, which will help us to verify security properties over multi-application smart cards. Further we have shown how typical properties of multi-application programs can be specified, and decomposed into specifications over the applets. The program model and logic are language-independent, but can easily be instantiated for JavaCard applications, as is illustrated by the purse example.

Future work The work presented here is only a first step towards a specification and verification framework for (security) properties of multi-application smart cards. Future work will concentrate on the following topics.

- Based on [4, 5] a proof system will be developed (and proven sound and complete) which will allow one to prove the correctness of the decomposition.
- At the moment the program model only deals with the control flow structure of the program. To be able to express integrity properties as *the balance of the purse is not changed by any action in the loyalty applet* one also needs to be able to talk about data. To this end, the program model has to be extended with data. Every applet will contain several variables (or fields), and for each program step it has to be described how these variables might be affected.
- After decomposing the global property, it remains to be shown that the individual applets satisfy the required properties. When dealing with control flow based security properties only, we can fall back on the model checking techniques developed by Jensen *et al.* [7], but after extending the model with data, more sophisticated techniques will be required. Abstraction techniques will be used to simplify the applets and the properties in such a way that they can be checked by model checking.

References

1. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Electronic purse applet certification: extended abstract. In S. Schneider and P. Ryan, editors, *Proceedings of the workshop on secure architectures and information flow*, volume 32 of *Elect. Notes in Theor. Comp. Sci.* Elsevier Publishing, 2000.
2. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *Software Tools for Technology Transfer (STTT)*, 2/4:410–425, 2000.
3. J. Corbett, M. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, number 1885 in LNCS. Springer, 2000.
4. M. Dam and D. Gurov. Compositional verification of CCS processes. In D. Bjorner, M. Broy, and A.V. Zamulin, editors, *Perspectives of System Informatics '99*, number 1755 in LNCS, pages 247–256. Springer, 1999.
5. M. Dam and D. Gurov. μ-calculus with explicit points and approximations. In *FICS 2000*, 2000.
6. G. Holzmann. The model checker SPIN. *Transactions on Software Engineering*, 23(5):279–295, 1997.
7. T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security policies. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 89–103. IEEE Computer Society Press, 1999.
8. D. Kozen. Results on the propositional μ-calculus. *Theor. Comp. Sci.*, 27:333–354, 1983.

Formal Models of Java at the JVM Level
A Survey from the ACL2 Perspective

J Strother Moore
Robert Krug
Hanbing Liu
George Porter

Department of Computer Sciences, University of Texas at Austin
moore@cs.utexas.edu
http://www.cs.utexas.edu/users/moore

Abstract. We argue that a practical way to apply formal methods to Java is to apply formal methods to the Java Virtual Machine (JVM) instead. A Java system can be proved correct by analyzing the bytecode produced for it. We believe that this clarifies the semantic issues without introducing inappropriate complexity. We say "inappropriate" because we believe the complexity present in the JVM view of a Java class is inherent in the Java, when accurately modeled. If it is desired to model a subset of Java or to model "Java" with a slightly simpler semantics, that can be done by formalizing a suitable abstraction of the JVM. In this paper we support these contentions by surveying recent applications of the ACL2 theorem proving system to the JVM. In particular, we describe how ACL2 is used to formalize operational semantics, we describe several models of the JVM, and we describe proofs of theorems involving these models. We are using these models to explore a variety of Java issues from a formal perspective, including Java's bounded arithmetic, object manipulation via the heap, class inheritance, method resolution, single- and multi-threaded programming, synchronization via monitors in the heap, and properties of the bytecode verifier.

1 ACL2 Background

ACL2 [11] is a functional programming language based on Common Lisp, a first-order mathematical logic with induction and recursive definition, and a mechanical theorem prover in the style of the Boyer-Moore theorem prover NQTHM [2, 4]. Among other successful industrial uses of ACL2 is the verification of the hardware designs for the elementary floating-point arithmetic operations on the AMD Athlon microprocessor [21] and the formalization of the first silicon version of the JVM [8, 9]. See [10] for other case studies.

In this paper we advocate the use of formal models of the JVM [13] to verify Java programs. Some readers may think this is an impractical suggestion. But work by Yu [5] with NQTHM (the predecessor of ACL2) supports our suggestion. Yu developed an operational formal model of the Motorola 68020 and then

verified C programs from the Berkeley C String Library by verifying the machine code produced by `gcc`. Since the conceptual gap between C and 68020 machine code is much greater than the gap between Java and JVM bytecode, we believe it is reasonable to follow an analogous strategy to deal with Java programs.

2 Our Basic Approach

Our models of the JVM are operational ones. The state of the machine is represented by a list containing, say, a thread table, a heap, and a class table. The thread table is a list containing an entry for each thread. The entry includes the thread's call stack, scheduled status and other information. A call stack is a stack (list) of frames, each of which contains a program counter, the method body, a map from local variable names to values, an operand stack, and a flag indicating whether the method is synchronized. The heap is a finite mapping from reference "addresses" to instance objects. The class table is a list describing the superclasses, fields and methods and other attributes of each class. We then define in ACL2 the function that "steps" such a state, producing the next state. We finally define a function, `run`, that "runs" a state, by stepping it repeatedly. Such an ACL2 model of the JVM may be thought of as a system of Lisp programs that simulates the JVM.

We have produced several such models of the JVM, so that we can explore ways to prove various kinds of properties. Before discussing the variety of formal models we have, we will use one of them to illustrate the foregoing sketch. The model we use is a multi-threaded JVM with unbounded arithmetic. It support classes, instances, instance methods, monitors and synchronization, but not arrays, floats and certain other data types. It also completely ignores class loading, constructor methods, exceptions, the JVM's provisions for type safety, and a variety of other issues.

For each JVM opcode supported in the model we define an ACL2 function that produces the corresponding state change. Here, for example, is that part of the formal model for an instruction called `LOAD`, which is analogous to the JVM's family of typed load instructions `ILOAD`, `ALOAD`, `DLOAD`, etc. In this function, `inst` is a symbolic form of the particular load instruction to be executed; its value will be a list of the form `(LOAD` *var*`)`, where *var* is a variable name. The variable `th` identifies which thread is to be stepped and `s` is the JVM state.

```
(defun execute-LOAD (inst th s)
  (make-state
   (modify-tt th
    (push (make-frame (+ 1 (pc (top-frame s th)))
                      (locals (top-frame s th))
                      (push (binding (arg1 inst)
                                     (locals (top-frame s th)))
                            (stack (top-frame s th)))
                      (program (top-frame s th))
                      (sync-flg (top-frame s th)))
```

```
          (pop (call-stack s th)))
    'SCHEDULED
    (thread-table s))
   (heap s)
   (class-table s)))
```

Informally, this function returns a new state obtained by changing the thread table of `s` at thread `th`. The topmost item on the call stack of that entry is popped off and replaced by a new frame in which the program counter has been advanced and the value of *var* has been pushed onto the operand stack of that frame.

Here is our bytecode for the instance method

```
public int fact(int n){
 if (n<=0) return 1;
   else return n*fact(n-1);}
```

except in our model arithmetic is not bounded.

```
("fact" (N) NIL                    ;  Method int fact(int)
 (LOAD N)                          ;   0 iload_1
 (IFGT 3)                          ;   1 ifgt 6
 (PUSH 1)                          ;   4 iconst_1
 (XRETURN)                         ;   5 ireturn
 (LOAD N)                          ;   6 iload_1
 (LOAD THIS)                       ;   7 aload_0
 (LOAD N)                          ;   8 iload_1
 (PUSH 1)                          ;   9 iconst_1
 (SUB)                             ;  10 isub
 (INVOKEVIRTUAL "Alpha" "fact" 1)  ;  11 invokevirtual ...fact...
 (MUL)                             ;  14 imul
 (XRETURN))                        ;  15 ireturn
```

Because our model is an ACL2 program, it can be executed on concrete data to produce concrete results. Because ACL2 is a mathematical logic, it is possible to prove the following theorem:

```
(implies (poised-to-invoke-fact th s n)
         (equal (top
                 (stack
                  (top-frame
                   th
                   (run (fact-sched n th) s))))
                (factorial n)))
```

which says that, given any state poised, in thread `th`, to execute the `fact` bytecode the natural number `n`, the execution of a certain number of instructions in thread `th` will leave `n`! on top of the operand stack in thread `th`. The number of instructions required is given by the function `fact-sched`, which returns a

schedule adequate to compute the method on input `n`. We have proved similar theorems about other arithmetic methods, methods manipulating the heap in destructive ways [15], and insertion sort implemented in a list processing class [12]. Insertion sort is discussed briefly below.

3 A Survey of Our Models

We have several JVM models and are in the process of building others. All of our current models ignore floats, class loading and initialization, exceptions, and interfaces. We do not consider floats a problem; there is so much work in modeling floating-point arithmetic in ACL2 (see for example [21]) that we have extensive floating-point models and libraries about them. Aspects of class loading and initialization, exceptions and interfaces have been modeled by others [19, 1]. Garbage collection is invisible on the JVM and so need not be modeled.

3.1 Single-Threaded/Non-Safe/Unbounded

Our basic model is a single-threaded JVM in which we ignore typing issues and support unbounded integer arithmetic only. Using this model we have proved a variety of theorems about bytecode programs, including a single-threaded version of the factorial theorem above and theorems involving the overriding of methods and the destructive modification of instance objects in the heap [15]. Using this model we can explore basic issues of code specification and verification, including control flow and data operations, instance object creation and manipulation, class inheritance, and method resolution and invocation.

For example, we have used the model to prove the correctness of a bytecoded insertion sort method that copies a linked list of numbers in the heap, producing a permutation of it in which the elements appear in ascending order. To state the theorem we had to define the sense in which a reference (into a given "non-circular" heap) denotes some structure. The theorem we proved says that if the `isort` method (not shown here) is invoked on a reference, ref_0 and allowed to run for a certain number of instructions, returning a reference ref_1, then the list denoted by ref_1 in the final heap is an ordered permutation of the list denoted by ref_0 in the original heap. The preconditions imposed certain constraints on the non-circularity of the initial reference [12]. Here is the theorem proved.

```
(implies (poised-to-invoke-isort s0)
         (let* ((x0 (top (stack (top-frame s0))))
                (heap0 (heap s0))
                (n0 (isort-clock x0 heap0))
                (s1 (run n0 s0))
                (x1 (top (stack (top-frame s1))))
                (heap1 (heap s1)))
           (let ((list0 (deref* x0 heap0))
                 (list1 (deref* x1 heap1)))
            (and (ordered list1)
```

```
(perm list1 list0)))))
```

One can prove theorems about non-terminating computations in ACL2. If one adds to the model an instruction for explicitly indicating the normal termination of a program (e.g., add a halt flag to the state and arrange for a bytecode instruction, e.g., `halt`, to set it and for the machine not to proceed afterwards), one can prove theorems about the conditions under which a program halts normally, including that halting never occurs. One can also eliminate the use of "clocks" [14].

3.2 Single-Threaded/Non-Safe/Bounded

We have produced a version of the simple machine that supports Java's `int` and `long` (bounded) arithmetic. It also supports arrays. Using this model we have a proved the analogous theorem about the bounded factorial method. The code for this method is like that shown for `fact` above, except that the arithmetic operations are those for 32-bit twos complement. The theorem states that the final answer is equal to the result of converting to an `int` the factorial of the input. This theorem correctly characterizes the actual behavior of the Java program for `fact` shown above.

The user input required to prove the bounded factorial is exactly analogous to that required to prove the unbound factorial, justifying our belief that the unbounded model is a simpler (though technically inaccurate) test bed. The "new" reasoning, about modular arithmetic, is handled automatically by an ACL2 library of lemmas. We are continuing the development of ACL2's already extensive collection of arithmetic theorems.

3.3 Multi-Threaded/Non-Safe/Unbounded

An orthogonal variation of the basic model introduces multiple threads [17].

Each entry in the thread table lists a unique thread number, a call stack, a status flag (e.g., indicating whether the thread has been started), and a reference to the instance object representing the thread object in the heap. We do not model the scheduler, which is unspecified in [13], but provide an "oracle" to the operational semantics.

With this model we have proved an interesting theorem about the Java classes shown in Figure 1. Inspection of the code shows that the `main` method in class `Apprentice` starts an unbounded number of `Jobs`, each of which is contending for a shared object called the `Container`. Each `Job` is in an infinite loop incrementing the `counter` field of the `Container`. Each such increment is done within a synchronized block. (The model supports unbounded arithmetic.)

One might think that it is obvious that the value of the `counter` field of the `Container` increases monotonically. However, this is a nontrivial observation that requires showing that each `Job` has mutually exclusive access to the counter. Again, the naive Java user may think this mutual exclusion property is obvious.

```
class Container {
    public int counter; }
class Job extends Thread {
    Container objref;
    Object x;
    public Job incr () {
        synchronized(objref) {
            objref.counter = objref.counter + 1; }
        return this; }
    public void setref(Container o) {
        objref = o; }
    public void run() {
        for (;;) {
            incr(); } } }
class Apprentice {
    public static void main(String[] args){
        Container container = new Container();
        for (;;) {
            Job job = new Job();
            job.setref(container);
            job.start(); } } }
```

Fig. 1. The Apprentice Class: Unbounded Parallelism

We have had several programmers dismiss our theorem as trivial and claim that it may be observed merely by looking at the text

```
synchronized(objref) {
    objref.counter = objref.counter + 1; }
```

in the code for class `Job`. This claim is false.

A few changes to the `main` method of the `Apprentice` class can cause mutual exclusion to be violated and can permit the `counter` value to decrease under some scheduling regimes. These changes do not involve writing to the `counter` field of the `Container` or changing the `Job` class at all. The pathological behavior (of the counter decreasing) is ultimately manifested by the very assignment statement shown above. The changes we have in mind can cause that "synchronized" assignment statement to clobber the counter without owning the monitor for it.

Since many readers insist that it is "obvious" that the `Apprentice` class causes the counter to increase monotonically, we will not explain here how to cause the bad behavior. Ask someone who thinks it is obvious. Or try to prove it from a detailed formal model of multi-threaded Java. Our discussion of the problem and our proof is reported in [16].

Our multi-threaded model includes all of the functionality of our basic machine (e.g., classes, heap-allocated instance objects, virtual method invocation, etc.) plus support for the `Thread` class (including the significance of the `run`

method for an extension of the `Thread` class, the native methods `start` and `stop`, monitors on all `Objects`, the opcodes `MONITORENTER` and `MONITOREXIT`, and support for synchronous methods.

4 Relations Between Models

So far we have only discussed theorems about particular bytecoded methods under the semantics formalized in particular models. Because our models are formal, we can reason about the models themselves and even relate them. Lack of space precludes much discussion.

4.1 Single- versus Multi-Threaded Models

We have proved [18] a theorem relating the single-threaded model to the multi-threaded one. If the multi-threaded machine is being used to do what is essentially a single-threaded computation, the single-threaded machine may be used instead. We formalize the hypothesis so that we are concerned with states in which only one thread is scheduled (meaning the `start` method has been called on only one thread) and the bytecode running in that thread does not create or interfere with other threads. The conclusion is a "commuting diagram" stating that the "same" computation could be done on the single-threaded model by transforming the states appropriately. The theorem allows us to "lift" certain verified programs from the single-threaded model to the multi-threaded model.

Ultimately we hope to be able to reason formally about "independent" concurrent threads by reasoning about each on the single-threaded model. The biggest problem will be combining the "independent" effects of the two threads on the shared heap. This involves reasoning not unlike that already done in analyzing the denotation of the object references in the heap produced by the insertion sort method.

4.2 Single-Threaded/Type Safe/Unbounded

We have developed a "type safe" version of the basic machine. Before each instruction is executed, this machine checks that the state is suitable for the execution of the instruction. For example, if an `ADD` instruction is to be executed, then the machine dynamically checks that the operand stack has at least two items on it and that the top two items are numbers. The machine sets a flag in the state and halts if the next instruction is to be executed in an unacceptable situation.

We are developing a formal version of the Java bytecode verifier described by [13] that crawls over a class declaration and does a certain syntactic check of the code therein. Our goal is to prove a theorem relating the type safe machine to the unsafe machine, namely, the two are "equivalent" on code that has been accepted by the bytecode verifier. This work can be thought of as leading towards the formal statement of the correctness of the bytecode verifier and the mechanized verification that for a particular verification algorithm.

5 Related Work

The earliest formal mechanized JVM model we know of was Cohen' "defensive JVM" [6], formalized in ACL2. Our series of models evolved from his: Moore and Cohen simplified Cohen's model and developed the series of successive elaborations to make it easier to teach at the undergraduate level.

Projects formalizing the JVM are ongoing in other mechanized logics with considerable success. The soundness of a bytecode verification algorithm is addressed in Isabelle/HOL in [20, 19]. The approach follows closely the class file format of [13] and model aspects of interfaces, signatures and exceptions, all of which we ignore. As in [6] and (some of) our work, type information is stored with data and instructions are modeled as state transforming functions. The Isabelle/HOL work is the first published mechanically checked proof of the soundness of a bytecode verifier.

Somewhat closer to our work is that done with Coq and described in [1]. In this work, an operational model of the entire JavaCard VM is presented. They provide a tool for converting class files into their formal format. They also verify a bytecode verifier mechanically. The authors of [1] stress the importance of executability – an emphasis with which we agree. They do not discuss the efficiency with which their model can be implemented.

ACL2 was used to model the Rockwell JEM1 microprocessor, the world's first silicon JVM, now marketed by Ajile Systems, Inc. The formal ACL2 model was actually used in the standard test bench on which Rockwell engineers tested the chip design against the requirements by executing compiled Java programs. The ACL2 model executed at approximately 90% of the speed of the previously used C model [8, 9]. In [7], Wilding and Greve describe how microprocessor models in ACL2 are made to execute fast. The model there executes at approximately 3 million simulated instructions per second on a 733 MHZ Pentium III host running Allegro Common Lisp.

As far as we know, ours is the first formal thread model for the JVM. In addition, the emphasis of our work is on the verification of bytecode programs with respect to the operational semantics. This is surely within the reach of the related work above, but has not, apparently, been a focus of their work. Because of the way previously proved lemmas in the ACL2 library can be used to configure ACL2 to do proofs automatically in a given domain, we anticipate that the continued development of correctness proofs for individual bytecoded methods will increase the ease with which new methods can be verified.

6 Conclusion

We have described a variety of formal models of the JVM and discussed Java and JVM programs that we have verified with respect to these models. We have also discussed formally verified relationships between some of our models.

These examples support the contention that with formal operational semantics of the JVM one can

- specify and verify Java code with respect to a detailed and accurate semantics,
- reuse much previously developed formal work,
- explore the specifications of code under various refinements of the semantics of Java,
- establish properties of the semantic models,
- formally relate different semantic models, and
- specify and verify the bytecode verifier.

Our models are inadequate for practical Java: among other omissions are floating point, exceptions, and class loading. But there is ample evidence [10] that ACL2 is rugged enough to permit the models to be sufficiently elaborated.

Among the compelling reasons to base a formal semantics of Java on an operational semantics of the JVM are the following. First, the Java compiler takes care of many subtle static semantics issues. Second, the operational semantics of the JVM can be executed, meaning it is possible to test the semantics against accepted implementations of the JVM. Third, the operational semantics is easily unwound by standard symbolic evaluation and induction techniques [3]. Fourth, and most important, the semantics is rendered *formally*, so it can be inspected by language experts and used directly by the verifier.

7 Acknowledgments

Our JVM models owe much to Rich Cohen who used ACL2 to formalize a single-threaded version of the "defensive JVM" [6]. We are grateful to Rich for his pioneering effort into the JVM formalization, as well as to the entire ACL2 and NQTHM communities for their development of techniques to formalize and reason about such machines. We are also grateful to David Hardin and Pete Manolios, who have each made many valuable suggestions in the course of this work.

References

[1] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A formal executable semantics of the JavaCard platform. In D. Sands, editor, *ESOP 2001*, volume LNCS 2028, pages 302–319. Springer-Verlag, 2001.

[2] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[3] R. S. Boyer and J S. Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 147–176. MIT Press, 1996.

[4] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*. Academic Press, New York, 1997.

[5] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[6] R. M. Cohen. The defensive Java Virtual Machine specification, version 0.53. Technical report, Electronic Data Systems Corp, Austin Technical Services Center, 98 San Jacinto Blvd, Suite 500, Austin, TX 78701, 1997.

[7] D. Greve, M. Wilding, and D. Hardin. High-speed, analyzable simulators. In Kaufmann et al. [10], pages 113–136.

[8] D. A. Greve and M. M. Wilding. Stack-based Java a back-to-future step. *Electronic Engineering Times*, page 92, Jan. 12, 1998.

[9] David A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[10] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[11] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[12] M. Kaufmann and J S. Moore. A flying demo of ACL2. Technical Report `http://www.cs.utexas.edu/users/moore/publications/flying-%-demo/script.html`, Computer Sciences, University of Texas at Austin, 2000.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition)*. Addison-Wesley, 1999.

[14] P. Manolios and J S. Moore. Partial functions in acl2. Technical Report `http://-www.cs.utexas.edu/users/moore/publications/defpun/% -index.html`, Computer Sciences, University of Texas at Austin, 2001.

[15] J S. Moore. Proving theorems about Java-like byte code. In E.-R. Olderog and B. Steffen, editors, *Correct System Design – Recent Insights and Advances*, pages 139–162. LNCS 1710, 1999.

[16] J S. Moore and G. Porter. Mechanized reasoning about Java threads via a JVM thread model. *(submitted for publication)*, 2000. `http://www.cs.utexas.edu/users/moore/publications/m4/proofs.ps.gz`.

[17] J S. Moore and G. Porter. An executable formal JVM thread model. In *Java Virtual Machine Research and Technology Symposium (JVM '01)*, 2001 (to appear). `http://www.cs.utexas.edu/users/moore/publications/m4/model.ps.gz`.

[18] G. Porter. A commuting diagram relating threaded and non-threaded jvm models. Technical report, Honors Thesis, Department of Computer Sciences, University of Texas at Austin, 2001.

[19] Cornelia Pusch. Formalizing the Java virtual machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für Informatik, Technische Universiät München, 1998. See URL `http://www.in.tum.de/~pusch/`.

[20] Cornelia Pusch. Proving the soundness of a Java bytecode verifier in Isabelle/HOL. Technical report, Institut für Informatik, Technische Universiät München, 1998. See URL `http://www.in.tum.de/~pusch/`.

[21] D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998.

A Type System for Checking Applet Isolation in Java Card

Peter Müller and Arnd Poetzsch-Heffter

FernUniversität Hagen, 58084 Hagen, Germany,
{Peter.Mueller, Arnd.Poetzsch-Heffter}@Fernuni-Hagen.de

Abstract. A Java Card applet is, in general, not allowed to access fields and methods of other applets on the same smart card. This *applet isolation* property is enforced by dynamic checks in the Java Card Virtual Machine. This paper describes a refined type system for Java Card that enables mostly static checking of applet isolation. With this type system, most firewall violations are detected at compile time.

1 Introduction

The Java Card technology allows applications written in a subset of Java—so-called Java Card applets—to run on smart cards [Che00]. Several applets can run on a single card and share a common object store. Since the applets on a card may come from different, possibly untrusted sources, a security policy ensures that an applet, in general, cannot inspect or manipulate data of other applets. To enforce this *applet isolation* property, the Java Card Virtual Machine establishes an *applet firewall*, that is, it performs dynamic checks whenever an object is accessed, for example, by field accesses, method invocations, or casts. If an access would violate applet isolation, a `SecurityException` is thrown.

Dynamically checking applet isolation is unsatisfactory for two reasons: (1) It leads to significant runtime overhead. (2) Accidental attempts to violate the firewall are detected at runtime, that is, after the card with the defective applet has been issued, which could lead to enormous costs. In this paper, we sketch a refined type system for the Java Card language that allows one to detect most firewall violations statically by checks on the source code level. This type system serves three important purposes:

1. It reduces the runtime overhead caused by dynamic checks significantly.
2. Most firewall violations are detected at compile time. At runtime, only certain casts can lead to `SecurityException`s. These casts point programmers and verifiers at the potentially critical spots of a program.
3. The refined type information provides formal documentation of the kinds of objects handled in a program such as entry point objects, global arrays, etc., and complements informal documentation, especially, of the Java Card API.

Overview. In the remainder of this introduction, we describe the applet firewall and explain our approach. Section 2 presents the refined type system. The context conditions that replace dynamic checking of applet isolation are explained in Section 3. We offer some conclusions in Section 4.

1.1 Applet Firewall

The applet firewall essentially partitions the object store of a smart card into separate protected object spaces called *contexts* [Sun00, Sec. 6]. The firewall is the boundary between one context and another. It allows object access across contexts only in certain cases. In this subsection, we describe contexts, object access across contexts, and the dynamic checks that enforce the firewall.

Contexts. There is one context for each applet installed on a smart card (we neglect group contexts for brevity). The context for an applet `A` contains all `A` objects and all objects created by methods executed on objects in that context. The operating system of the card is contained in the Java Card Runtime Environment (JCRE) context. At any execution point, there is exactly one *currently active context* (in instance methods, this context contains `this`). When an object of context C invokes a method m on an object in context D, a *context switch* occurs, that is, D becomes the new currently active context. Upon termination of m, C is restored as the currently active context

Class objects do not belong to any context. There is no context switch when a static method is invoked. Objects referenced by static fields are ordinary objects, that is, they belong to an applet or to the JCRE context.

Firewall Protection. We say that an object is *accessed* if it serves as target for a field access or method invocation, or if its reference is used to evaluate a cast or `instanceof` expression (we do not treat exceptions and arrays here). In general, an object can only be accessed if it is in the currently active context (see below for exceptions to this rule). To enforce this rule, the Java Card Virtual Machine performs dynamic checks. If an object is accessed that is not in the currently active context, a `SecurityException` is thrown.

Object Access Across Contexts. The Java Card applet firewall allows certain forms of object access across contexts: (1) Applets need access to services provided by the JCRE. These services are provided by *JCRE entry point objects*. These objects belong to the JCRE context, but can be accessed by any object. In this paper, we only consider permanent entry point objects (PEPs for short). An extension to temporary entry point objects and global arrays is straightforward. (2) To support cooperating applets, applets can interact via *shareable interface objects* (SIOs for short). An object is an SIO if its class implements the `Shareable` interface. An applet can get a reference to an SIO of another applet by invoking a static method of the JCRE. It can then invoke methods on this SIO [Sun00, Sec. 6]. (3) The JCRE has access to objects in any context.

Example. In the following, we explain the dynamic checks for an invocation `e.m(...)` of a dynamically-bound method. We assume that `T` is the compile time type of `e` and that the evaluation of `e` yields an object X. Before the

invocation is executed, it is checked whether at least one of the following cases applies[1]. If no case applies, a `SecurityException` is thrown.

D1: X is in the currently active context;
D2: X is an entry point object and `T` is a class;
D3: `T` is an interface that extends `Shareable`;
D4: The JCRE context is the currently active context.

We illustrate these checks by a faulty implementation of two cooperating applets. Fig. 1 shows the implementation of a client applet. We assume that the client and a server applet are installed on the same card. The following interaction is initiated by method `Client.process`: The client requests an SIO from the server by invoking `JCSystem.getAppletShareableInterfaceObject`, which yields an SIO that is cast to the shareable interface `Service`. The client then invokes `doService` on the SIO. This invocation yields a new `Status` object that is used to check whether the service was rendered successfully.

In our implementation, this interaction leads to a `SecurityException`: The client and server applets reside in different contexts. The `Service` SIO and the `Status` object belong to the context of the server. Since `Status` does not implement `Shareable`, the `Status` object is not an SIO. When the invocation `sta.isSuccess()` is checked as explained above, cases **D1**–**D4** do not apply. Thus, the access is denied and the exception is thrown. To correct this error, one would have to use an interface that extends `Shareable` instead of class `Status`; then case **D3** would apply.

1.2 Approach

To detect firewall violations at compile time, we adapt type systems for alias control such as ownership types and universes [CPN98,MPH01,Mül01]. Whereas these type systems focus on restricting references between different contexts, we permit references between arbitrary contexts, but restrict the operations that can be performed on a reference across context boundaries.

Our type system augments every reference type of Java with context information that indicates (1) whether the referenced object is in the currently active context, (2) whether it is a PEP, or (3) whether it can belong to any context. Type rules guarantee that every execution state is well-typed, which means especially that the context information is correct.

We use downcasts to turn references of kind (3) into references of more specific types. For such casts, dynamic checks guarantee that the more specific type is legal. Otherwise, a `SecurityException` is thrown.

To check an applet with our type system, its implementation as well as the interfaces of applets it interacts with and of the Java Card API must be enriched

[1] The checks correspond to the rules explained in the above paragraphs. We have adopted them from [Sun00, Sec. 6.2.8] although **D2**'s requirement that `T` be a class seems overly restrictive. Please refer to [Sun00] for a more detailed explanation of the checks.

```
public class Status {
  private boolean success;
  public Status(boolean b)   { success = b; }
  public boolean isSuccess() { return success; }
}

public interface Service extends Shareable {
  Status doService();
}

public class Client extends Applet {
  private Client() { register(); }
  public static void install(byte[] a, short o, byte l) { new Client(); }

  public void process(APDU apdu) {
    AID     svr = ...;              // server's AID
    Shareable s = JCSystem.getAppletShareableInterfaceObject(svr,(byte)0);
    Service ser = (Service)s;       // cast is legal
    Status  sta = ser.doService();  // invocation is legal
    if (sta.isSuccess())            // leads to SecurityException
      ...
  }
}
```

Fig. 1. Implementation of a client applet. All classes are implemented in the same package. `package` and `import` clauses are omitted for brevity. We assume that a server applet is implemented in a different package.

by refined type information. This information is used to impose additional context conditions for field accesses, method invocations, casts, and `instanceof` expressions that guarantee that the firewall is respected.

In the execution of a program that is type correct according to our type system, only the evaluation of downcast expressions requires dynamic firewall checks and might lead to `SecurityException`s. Thus, casts point programmers at the critical spots in a program, which simplifies code reviews and testing. Moreover, they allow standard reasoning techniques to be applied to show that no `SecurityException` occurs.

In theory, our type system can replace almost all dynamic firewall checks. However, if only some applets on a card are checked by our type system, the dynamic checks have to stay in place to prevent applets from untrusted sources from violating the firewall. Still, our type system is useful to detect possibly fatal program errors at compile time, which simplifies reasoning and reduces costs.

In the following sections, we present the refined type system and some of the additional context conditions. For brevity, we focus on a subset of Java Card and omit exceptions and arrays. Moreover, we do not treat temporary entry points and global arrays. An extension of our work to these features is straightforward.

2 The Type System

A type system expresses properties of the values and variables of a programming language that enable static checking of well-definedness of operations and their application conditions, in this case, Java Card's firewall constraints.

Tagged Types. In order to know whether an operation is legal in Java Card, we need information about the context in which the operation is executed. The basic idea of our approach is to augment reference types by context information.

In this paper, we are only interested in checking applet code and do not consider the JCRE implementation. Thus, statements and expressions are either executed in an applet context (in case of instance methods) or outside all contexts (in case of static methods). From the point of view of an applet context C, we can distinguish (a) internal references to objects in C, (b) PEP references, and (c) external references to objects in applet contexts different from C or to non-PEP objects in the JCRE.

In the type system, we reflect this distinction by the *context tags* `i` for internal, `p` for PEP, and `a` for any. The `a`-tag expresses the fact that it is not known whether a reference is internal, PEP, or external. A special tag for external is dispensable, because all operations that are allowed on external references are allowed on "any" reference. Since static class members do not belong to any context, the `i`-tag must not be used for types of static fields or in static methods. Let TypeId denote the set of declared type identifiers of a given Java Card program; then the tagged type system comprises the following types:

$$\text{TaggedType} = \{\text{booleanT}, \text{intT}, ..., \text{nullT}\} \cup (\{\texttt{i}, \texttt{p}, \texttt{a}\} \times \text{TypeId})$$

Except for the null-type that is used to type the `null` literal, all reference types in the tagged type system are denoted as a pair of a tag and a Java type. The subtype relation $\preceq$ on tagged types is the smallest reflexive, transitive relation satisfying the following axioms, where G is a tag, $S, T \in \text{TypeId}$, and $\preceq_J$ denotes the subtype relation on TypeId:

$$(G, S) \preceq (G, T) \Leftrightarrow S \preceq_J T \qquad (G, T) \preceq (\texttt{a}, T) \qquad \text{nullT} \preceq (G, T)$$

Tagged Type Rules. We illustrate our approach by presenting the most interesting rules for the tagged type system. Since the type rules for statements are trivial, we focus on expressions. A type judgment of the form $E \vdash e :: TT$ means that expression e has tagged type TT in the declaration environment E of the method enclosing e.

The type judgment for instance creation expressions without parameters is $E \vdash \text{new T}() :: (\texttt{i}, \text{T})$. The `i`-tag indicates that the created object belongs to the currently active context.

The tagged type of a cast expression is the type (H, T) appearing in the cast operator (see below). For simplicity, we only consider downcasts, that is, (H, T) has to be a subtype of the expression type. Note that we allow a reference tagged

"any" to be cast into an internal or PEP reference. Recall that dynamic checks guarantee that the more specific tag is appropriate (see Subsec. 1.2)

$$\frac{E \vdash e :: (G,S) \;\;,\;\; (H,T) \preceq (G,S)}{E \vdash (\,(H,T)\,)\,e :: (H,T)}$$

The most interesting and complex rule is the one for invocation expressions. For simplicity, we assume that methods have exactly one parameter of tagged type (F_P, T_P) and a (tagged) return type (F_R, T_R):

$$\frac{E \vdash e1 :: (H,T) \;\;,\;\; E \vdash e2 :: (G,S) \;\;,\;\; (H,T) * (G,S) \preceq (F_P,T_P)}{E \vdash e1.m(e2) :: (H,T) * (F_R,T_R)}$$

The operator $* : \text{TaggedType} \times \text{TaggedType} \to \text{TaggedType}$ is defined as follows: $(H,T) * (G,S) = (\mathtt{a},S)$, if $H \neq \mathtt{i}$ and $G = \mathtt{i}$; $(H,T) * (G,S) = (G,S)$ in all other cases. The $*$-operator tags the parameter or result as "any" when the invocation could lead to a context switch ($H \neq \mathtt{i}$) and an internal reference is passed to or returned by the method ($G = \mathtt{i}$). This is necessary since an internal reference is external to the new currently active context, as illustrated by the following example.

Fig. 2 shows the `Service` interface and the `Client` class with tagged type information. The return type of `Service.doService` is internal since the method creates a new `Status` object in the context in which it is executed (the context of the server applet). When `doService` is invoked from the client context (see method `Client.process`, Fig. 2), the returned `Status` object is external to the client context and must, thus, be tagged "any". This adaption of the tag is described by the $*$-operator.

Type Safety. We proved that the tagged type system is type safe in the following sense: If the evaluation of an expression e starts in a type correct state, it yields a type correct state upon termination and the resulting value belongs to the tagged type of e. Informally, this property states that all references are correctly tagged, which is a prerequisite for statically checking applet isolation (see Sec. 3). The type safety proof is based on an operational semantics the states of which, in particular, contain a variable that keeps track of the currently active context.

The type safety proof runs by rule induction. An interesting aspect is the treatment of context information and their relation to tagged types. Each class instance "knows" the context it belongs to and whether it is a PEP. Based on this information, we can assign a tagged type to an object X in a context C ($\mathit{ctyp}(X)$ denotes the TypeId of X's class):

$$\mathit{ttyp} : \text{Object} \times \text{Context} \to \text{TaggedType}$$
$$\mathit{ttyp}(X,C) = \begin{cases} (\mathtt{p},\, \mathit{ctyp}(X)), & \text{if } X \text{ is a PEP} \\ (\mathtt{i},\, \mathit{ctyp}(X)), & \text{if the context of } X \text{ is } C \text{ and } X \text{ is not a PEP} \\ (\mathtt{a},\, \mathit{ctyp}(X)), & \text{otherwise} \end{cases}$$

3 Checking Applet Isolation

Tagged types provide a conservative approximation of runtime context information. This information can be used to impose *static checks* that guarantee that an applet respects the applet firewall at runtime. In the following, we present these checks for method invocations and argue why they enforce applet isolation.

Static Checks. We perform the following static checks for each invocation statement of the form `e.m(...)`, where (H, T) is the static tagged type of `e`. *class(T)* yields whether T denotes a class.

S1: $H = \texttt{p} \;\Rightarrow\; \mathit{class}(T)$
S2: $H = \texttt{a} \;\Rightarrow\; \neg\mathit{class}(T) \wedge T \preceq_J \texttt{Shareable}$

To show that these static checks prevent `SecurityExceptions`, we explain for each possible tag of `e`'s type that one of the cases **D1**–**D3** of the dynamic checks presented in Sec. 1 applies. (**D4** is not relevant here since we only consider applet code, not the implementation of the JCRE.) We assume that `e` evaluates to an object X:

$H = \texttt{i}$**:** Well-typedness guarantees that X is in the currently active context; that is, case **D1** applies.
$H = \texttt{p}$**:** Well-typedness guarantees that X is a PEP, and static check **S1** guarantees that T is a class; thus, case **D2** applies.
$H = \texttt{a}$**:** Static check **S2** guarantees that T is an interface that extends `Shareable`; thus, case **D3** applies.

In summary, well-typedness and the above static checks guarantee that execution of a method invocation does not violate the firewall at runtime. Therefore, the checks can be used to enforce applet isolation statically. The static checks for other expressions are analogous.

We proved the following *firewall lemma*: Each Java Card program with tagged types that passes the static checks behaves like the corresponding Java Card program with dynamic checks. That is, every Java Card program that can be correctly tagged does not throw `SecurityExceptions` (except for the checks for casts). The proof of the firewall lemma is based on two operational semantics: one semantics with dynamic checks, the other without them. It runs by rule induction and exploits type safety of the tagged type system and the static firewall checks.

Example. Fig. 2 shows `Client`'s `process` method with tagged type information. Since `getAppletShareableInterfaceObject` is a static method that does, in general, not return a PEP, its return type is `any Shareable`. The `any`-tag carries over to `sta` (the status is extern). Therefore, the invocation `sta.isSuccess()` does not pass static check **S2** since `Status` is a class that does not implement `Shareable`. That is, the firewall violation caused by this invocation would be detected at compile time.

```
public interface Service extends Shareable {
  intern Status doService();
}

public class Client extends Applet {
  ...
  public void process(any APDU apdu) {
    intern AID    svr = ...;     // server's applet id is intern
    any Shareable   s =
                JCSystem.getAppletShareableInterfaceObject(svr,(byte)0);
    any Service   ser = (any Service)s;     // ser is in general extern
    any Status    sta = ser.doService();    // sta is also extern
    if (sta.isSuccess())                    // static check fails
      ... }
  ...
}
```

Fig. 2. `Service` interface and `Client` class with tagged type information. In the concrete syntax, we use the keywords `intern`, `pep`, and `any` as tags.

4 Conclusions

We presented a refined type system for Java Card that allows one to check applet isolation statically. The type system is based on the more general approach of universe types that was developed for the modular verification of Java programs. Our future goal is to formally verify interesting properties of Java Card applets such as the absence of `SecurityException`s. To show this property, we would check the applet by the tagged type system and prove benevolence of the casts. Since we are interested in source code verification, we designed our checking technique for the source level, whereas other approaches focus on the byte code level [BCG+00,CHS01]. It could be adapted to the byte code level as well.

References

[BCG+00] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. In *ESORICS*, 2000.

[Che00] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, 2000.

[CHS01] D. Caromel, L. Henrio, and B. Serpette. Context inference for static analysis of Java Card sharing. Available from `www-sop.inria.fr/oasis/personnel/Ludovic.Henrio/JavaCardSharingAnalysis.ps.gz`, 2001.

[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.

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[Mül01] P. Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001.

[Sun00] Sun Microsystems, Inc. *The Java Card 2.1.1 Runtime Environment (JCRE) Specification*, May 2000.

Biographies

Peter Müller is a member of the VerifiCard project that is concerned with the formal verification of Java Card applets. He recently received a Doctorate in Computer Science from FernUniversität Hagen with a thesis on *Modular Specification and Verification of Object-Oriented Programs.*

Arnd Poetzsch-Heffter is associate professor at FernUniversität Hagen. He received a Doctor in Computer Science from the Technical University of Munich in 1991 with a thesis about programming language specification. During his postdoc year at Cornell University, he began research on the integration of program specification and verification techniques for OO-programs. In his Habilitation thesis, he developed the formal foundations for such an integration. Currently, he is especially engaged in the development of specification and verification techniques and tools for OO-programs.

The Interdependence of Effects and Uniqueness*

John Boyland

Department of EE & CS, University of Wisconsin-Milwaukee
boyland@cs.uwm.edu

Abstract. A good object-oriented effects system gives the ability to define abstract regions (or "data groups") of state within objects that can be extended in subclasses. Then one can specify (for instance) read and write effects on these abstract regions. Additionally, effects on "wholly owned subsidiary" objects should be seen as effects on regions of the owning object. For instance, an assignment within a bucket of a hash table should be seen as an effect on the hash table alone. Correctness of this transfer of effects depends on the bucket being accessible only through the hash table; it must be *unique*.

Uniqueness can be guaranteed using *destructive reads* (in which a unique variable can be used at most once). Destructive reads are inconvenient, so most uniqueness systems permit *borrowing reads* as well, in which a temporary alias of a unique variable is permitted. But if the unique variable is read during the lifetime of this alias, the uniqueness invariant fails. So we wish to ensure that this read effect does not happen. For modularity reasons, we use effects annotations on methods to check for such read effects.

Thus we see that effects and uniqueness depend on each other. Our position is that the use of annotations breaks the cyclic dependence as long as the annotations are given semantics independent of the analyses. As a semantics of uniqueness annotations is already available, we then sketch a semantics of effects annotations independent of uniqueness. Thus decoupled, one can prove the correctness of a uniqueness analysis and an effects analysis without regard for the other.

1 Interdependence

Properties of code effects and alias confinement are important when analyzing the meaning of complex programs. Putting checked annotations on methods aids sound modular reasoning, because a component of a system can then be independently verified.

* Work supported in part by the National Science Foundation (CCR-9984681) and the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF under contract F30602-99-2-0522. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

```
class Point {
  public region Position;
  private int x in Position;
  private int y in Position;
  public scale(int sc)
    writes Position
  {
    x *= sc;
    y *= sc;
  }
}

class ColorPoint extends Point {
  public region Appearance;
  private int color in Appearance;
}
```

(a)

```
class Point3D extends Point {
  private int z in Position;
  public void scale(int sc)
    writes Position
  {
    super.scale(sc);
    z *= sc;
  }
}
```

(b)

Fig. 1. The definitions of (a) classes `Point` and `ColorPoint`, and (b) class `Point3D`

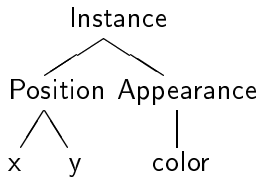


Fig. 2. Hierarchy of regions for class `ColorPoint`

In this section we motivate effects and uniqueness systems and show how each may depend on the other in the context of a hash table example. In Section 2, we discuss how the interdependence can be resolved by giving independent semantics to uniqueness and effects annotations. Since a semantics of uniqueness already exists, we propose the rough outline of a semantics of effects annotations.

1.1 Effects

In an object-oriented effects system (such as of Greenhouse and Boyland [8] or Leino [11]) the mutable fields of the object are abstracted as "regions" of state (the term used in this paper) or "data groups" [11]. This abstraction permits the effects of public methods to be declared using public abstractions that hide the names of actual fields. Regions are inherited along with fields and can be extended to include new regions and fields. Thus a subclass can extend the behavior of the superclass's methods to read or write additional fields. Figure 1 gives an example of a `Point` class with two subclasses illustrating extension

possibilities. The non-executable annotations are in *slanted* style. Figure 2 shows the hierarchy of regions within the `ColorPoint` class.

Often, not all the notional state of an object is actually contained directly within it. Part of the state of the object is contained in "wholly owned subsidiary" objects, private objects known only to the implementation of the class. Consider for example, Java's `Vector` class; the contents of the vector are stored in an array, which may need to be changed if the vector outgrows the array. The code for `addElement` is basically as follows:

```
public void addElement(Object elem) {
  ensureCapacity(size+1);
  contents[size] = elem;
  ++size;
}
```

where `size` and `contents` are private fields. Ignoring the effects of the call to `ensureCapacity`, this code has two side-effects: the array in `contents` is updated, and the private field `size` is changed. In this case, the `size` field is in a region `Size`. Thus the second change can be given as a (read and a) write to the `Size` region of the receiver (`this`) of the method call. The class has a second region `Elements` which contains the array field, but the array assignment does not actually change the array field itself, it changes the elements of the array that is stored in the field. The indirection causes a problem; it is not useful to the caller to say that "some array somewhere is changed" or even "some array which is referred to by a field in the region `this.Elements` is changed."

Instead of treating the array as a separate object, it is preferable to consider the array to be part of the vector, which is reasonable since the implementation arranges that every vector has its own array. So we use a *transfer of effect* from the array to the vector and declare the effects of the method as "state in this vector is changed." The transfer is declared in the `Vector` class using some syntax such as the following:

```
private Object[] list in Elements with [] in Elements;
```

Here the field is placed in the `Elements` region and the individual array elements (in a region `[]` of the array) are placed in the `Elements` region of the vector. Figure 3 gives a fragment of a hash table class using effects transfer. The effects on fields of the buckets are transitively transfered to the hash table object itself.

The soundness of this *transfer of effect* depends on several conditions. For our vector example, it is not enough that the array field be private; rather, the client must not have access to the array object through some other means. Otherwise, it could observe the effect on the array. For instance, two vectors must not use the same array, otherwise a change in one vector could be observed through the other vector. The invariant that we need is called a "uniqueness invariant." If an object is unique, effects on its regions can be safely mapped to effects on the object which refers to it. The effects analysis checks an effects transfer using uniqueness annotations.

```
class Bucket {
  public region Key, Value, Structure;
  Object key in Key, value in Value;
  Bucket next in Structure with Key in Key,
                                Value in Value,
                                Structure in Structure;
  Bucket(Object k, Object v, Bucket n) { ... }

  Object get(Object k)
    reads this.Key, this.Value, this.Structure, any(Object).Equal
  {
     if (key.equals(k)) return value;
     else if (next == null) return null;
     else return next.get(k);
  }
  ⋮
}

public class Hashtable {
  region Key; region Value; region Structure;
  private Bucket[] buckets in Structure with [].Key in Key,
                                             [].Value in Value,
                                             [].Structure in Structure;
  private int size = 0 in Structure;

  public synchronized Object get(Object k)
    reads this.Key, this.Value, this.Structure, any(Object).Equal
  {
    int h = k.hashCode() % buckets.length;
    Bucket b=buckets[h];
    if (b == null) return null;
    else return b.get(k);
  }
  ⋮
}
```

Fig. 3. Hash tables with effects transfer

1.2 Uniqueness

There have been a number of uniqueness proposals: Islands [9], Linearity [2], Eiffel* [16], Balloons [1], Virginity [15], and Alias Burying [3]. Islands and Balloons are not useful for modeling a vector class because they would not permit a vector to hold any shared references. Essentially the contents of the vector would have to be unique or immutable references. Flexible Alias Protection [18] describes how to avoid these problems but uses ownership [6] rather than uniqueness.

Baker's Linearity and Minsky's Eiffel* (as well as Hogg's Islands) use *destructive reads* in which a read of a unique variable also implicitly stores null in it. Destructive reads ensure uniqueness or null; instead of having multiple aliases to an object, all but one will hold null. Destructive reads, however, are not a satisfactory solution. First of all, it is difficult to program using such "slippery" variables: many methods that take unique variables must also return them as well as their normal result. Figure 4 shows how to code the hash table fragment from Fig. 3 in a system with strict destructive reads (which are underlined). We assume that operations such as `==` and `[.]` are overloaded to work for unique objects, so that comparisons and array accesses can be made without destroying unique objects. The second problem is that the code now has many more side-effects than previously, which will be difficult to ignore in an effects analysis.

More importantly, the code must now be prepared for the fact that the unique variable may actually be null. For example, the hash table `get` method must be prepared for the possibility that the field holding the array, or the array element holding the head of the bucket chain might currently be null due to an ongoing `get` call. Such a situation may take place, for instance, if the `equals` method for an object requires looking something up in a hash table. Aliasing errors may thus be transmuted into null pointer errors. While possessing the virtue of immediate detection, null pointer errors can still have devastating run-time consequences for the program. More desirable is a static checking system that can flag potential alias errors at compilation time. For instance, the hash table class does not put any fields into the `Equals` region (that may be read by the `equals` method). If an `equals` method tries to access a hash table, effects analysis would flag the access as an error.[1]

For these reasons, some systems with destructive reads (such as Eiffel* and Islands) provide "non-consuming" or "borrowing" reads in which certain kinds of so-called "dynamic" aliases of unique variables are permitted. Methods can be written to take borrowed receivers or parameters and promise not to store them anywhere. At the conclusion of the call, then, uniqueness is restored automatically. Alias Burying similarly supports controlled aliasing, but as soon as the unique variable is read, the aliases must be dead ("buried"). In particular, if a method call may read the unique field, even indirectly, it may not be passed

[1] Unfortunately the new collections framework requires `equals` on maps to compare contents. Personally, I believe it was a mistake to require mutable containers to override `equals` and `hashCode`.

```
class Bucket {
  Object key, value;
  unique Bucket next;

  Bucket(Object k, Object v, unique Bucket n) { ... }

  Pair<Object,unique Bucket> get(Object k) unique
  {
     if (key.equals(k)) return value;
     else if (next == null) return null; // we assume == doesn't destruct
     else {
       Pair<Object,unique Bucket> p = next.get(k);
       next = p.second;
       return new Pair<p.first,this>;
     }
  }
  ⋮
}

public class Hashtable {
  private unique Bucket unique [] buckets;
  private int size = 0;

  public synchronized Object get(Object k)
  {
    // NB: if buckets is null, we die with a NullPointerException
    Pair<int,unique Bucket unique []> p = buckets.unique_length();
    buckets = p.second;
    int h = k.hashCode() % p.first;
    unique Bucket b = buckets[h];
    Object result;
    if (b == null) {
      // NB: b might be null due to an ongoing 'get' call
      result = null;
    } else {
      Pair<Object,unique Bucket> p2 = b.get(k);
      result = p.first;
      b = p.second;
    }
    // need to restore array element
    buckets[h] = b;
    return result;
  }
  ⋮
}
```

Fig. 4. Hash tables with destructive reads (underlined)

an alias of that same field. With Alias Burying the uniqueness annotations given in Fig. 4 can be used with the code of Fig. 3; we do not need destructive reads.

Virginity and Alias Burying both provide ways to check uniqueness statically without changing the underlying language semantics. In an interesting convergence, both systems require "reads" clauses to ensure that uniqueness is not compromised [3, 14].

When a unique pointer is passed out of the scope of the owning object in a call (for example by calling a method), there must not be another call on the owning object that uses the unique field until the first call is complete. Otherwise, the uniqueness invariant would be violated; in the case of destructive reads, a null pointer would be encountered unexpectedly. In the case of alias burying, one of the called method's parameters is suddenly no longer valid. Thus the static analysis needs to ensure that the unique field is not read during the dynamic lifetime of the call. The Alias Burying paper [3] suggests listing the complete set of fields read during every procedure but concedes that this requirement breaks information hiding. A much better solution is to use an object-oriented effects system, but that brings us full circle.

Thus we see effects analysis depends on uniqueness analysis which depends on effects analysis. Leino has also noticed this interdependence [12].

2 Resolving the Interdependence

In this section, we show how the interdependence can be resolved soundly by giving a semantics to the annotations. Then we show how one can give semantics to uniqueness annotations and finally we sketch an idea for giving meaning to effects annotations.

2.1 Separating the Analyses

The interdependence is a concern to us when we are trying to determine whether the effects analysis and uniqueness analysis are both sound, and if so, writing a proof. If both analyses depend on each other, we need to prove the correctness of both together. In our situation, however, we are working with annotations on methods and classes that summarize their behavior in particular ways. The annotations provide some indirection: in the same manner as type checking, analysis checks the annotations on an entity while using the annotations on other entities, or (in the case of self-reference) itself.

Proving the soundness of the analysis, then, naturally requires that the annotations be assigned some meaning against which the analysis is checked. Furthermore, to avoid tautologies, and to permit the substitution of more accurate analyses, the semantics should be defined at a lower level than the analyses. In particular, we wish to avoid giving an annotation a meaning such as "Analysis A gives result B when applied to this method." Rather we are interested in meanings such as safety properties: "Event E will never happen while executing this method as long as the program state at the point of entry satisfies predicate I."

Assuming then that we can place the semantics on a complete (semi-)lattice and define our analyses monotonically on this lattice, proving the correctness of the analysis becomes an application of the theory of abstract interpretation [7]. In particular, we can prove the soundness of a uniqueness analysis separate from any effect analysis, and vice versa.

2.2 Semantics for Uniqueness

In a current paper [4], we give a capability-based low-level language that can be used to give a meaning to uniqueness annotations. Uniqueness is expressed through the exclusive holding of read and write access rights. Uniqueness invariant failures are converted into capability failures, so that any analysis that ensures the absence of capability failures ensures the correctness of the uniqueness annotations. Furthermore, the only way lack of an access right can be observed is through a capability failure, and thus if a program is guaranteed to never have failures, it can be executed in an environment that ignores access rights completely. In particular, no space is needed to store the access rights.

2.3 Semantics for Effects

A useful semantics of effects is not yet available. Our earlier paper [8] gives an indirect basis for defining the soundness of effects annotations: whether a conflict detection analysis using the effects annotations always catches data dependencies between adjacent program portions. But it does not give a semantics to the annotations directly, and the conflict detection analysis is overly conservative in several situations.

Leino's abstract variables [10] (from which data groups were derived) have a clear meaning in the context of a program specification. However, because of effects transfer, when a module specification uses `modifies` clauses[2] the client of a module cannot make interesting use of the information in a sound manner [13]. Any problem in effects transfer shows up as unsoundness in the client, which is "unfair" because the client has no control or even awareness of the transfer.

The semantics of effects could be specified using uniqueness or ownership [17, 5] but that would tie us to a particular system, not directly related to effects. Uniqueness systems find it difficult to model doubly linked lists and ownership systems find it difficult to model transfer. Thus we prefer an effects semantics not depending on the particular method for alias containment.

We suggest instead that the implementation of a method be charged with ensuring the soundness of effects transfer. In particular, the state thus mapped into the state of another object must not be available to any caller that does not have access to the object through which the transfer is effected. Our position is that this intuition can be formalized through a run-time hierarchy of actual regions. Effects transfer is implemented by mutations on this tree.

[2] Leino does not currently use `reads` clauses.

At the start of a program, the entire state (the set of all fields) is available for reads and writes. Whenever a method with an effects annotation is called, the available state is pared down to the intersection of the currently available state and the state implied by the annotations. The available state is then restored after the call. When a new object is created, all of its fields are made available for both reads and writes. When a field with an effects transfer is assigned to, we check that the object's state is fully available for reads and writes (except perhaps for immutability), and then transform the region hierarchy. If a read of a field outside the permitted area occurs, the state is presumed immutable, because reads of immutable state need not be declared. It is marked as such for the duration of the program. When a write occurs, the system checks that the field is in the area currently permitted for writes and that the field is not marked immutable. An error causes evaluation to get stuck.

In this way, we achieve a semantics of effect annotations and transfer that does not depend on a particular definition of uniqueness or ownership. The details of this suggested semantics remain to be worked out.

3 Summary

Two apparently different problems, (1) describing the reads and writes of methods and (2) upholding uniqueness invariants, are nonetheless interdependent. The connectivity presses us to define the semantics of effects separately from the semantics of uniqueness. We currently have a promising semantics for uniqueness, but an effects semantics which has the desired properties remains to be fully fleshed out.

Acknowledgments

I thank Aaron Greenhouse, Bill Retert and the FTJP 2001 reviewers for their many useful comments. I also thank Rustan Leino for a good technical conversation which encouraged me to write up this interesting interdependence. All omissions and errors are strictly my own.

References

1. Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming, 11th European Conference*, Jyväskylä, Finland, June 9–13, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, Berlin, Heidelberg, New York, 1997.
2. Henry G. Baker. 'Use-once' variables and linear objects—storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, January 1995.
3. John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, May 2001.

4. John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalization of uniqueness and read-only. To appear in ECOOP 2001, 2001.
5. David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. To appear in ECOOP 2001, 2001.
6. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, *ACM SIGPLAN Notices*, 33(10):48–64, October 1998.
7. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, California, USA, pages 238–252. ACM Press, New York, January 1977.
8. Aaron Greenhouse and John Boyland. An object-oriented effects system. In Rachid Guerraoui, editor, *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, Lisbon, Portugal, June 14–18, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229. Springer, Berlin, Heidelberg, New York, 1999.
9. John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA'91 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Phoenix, Arizona, USA, October 6–11, *ACM SIGPLAN Notices*, 26(11):271–285, November 1991.
10. K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, Pasadena, California, USA, 1995. Available as Technical Report Caltech-CS-TR-95-03.
11. K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, *ACM SIGPLAN Notices*, 33(10):144–153, October 1998.
12. K. Rustan M. Leino. Some thoughts about rep exposure and alias confinement. December 2000.
13. K. Rustan M. Leino and Gren Nelson. Data abstraction and information hiding. SRC Research Report 160, Compaq Systems Research Center, Palo Alto, CA, November 2000.
14. K. Rustan M. Leino and Raymie Stata. Smothering rep exposure with reads clauses. November 1999.
15. K. Rustan M. Leino and Raymie Stata. Virginity: A contribution to the specification of object-oriented software. *Information Processing Letters*, 70(2):99–105, April 1999.
16. Naftaly Minsky. Towards alias-free pointers. In Pierre Cointe, editor, *ECOOP'96 — Object-Oriented Programming, 10th European Conference*, Linz, Austria, July 8–12, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer, Berlin, Heidelberg, New York, July 1996.
17. Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. In Sophia Drossopolou, Susan Eisenbach, Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *2nd ECOOP Workshop on Formal Techniques for Java Programs*, Nice, France, June 12. 2000.
18. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98 — Object-Oriented Programming, 12th European Conference*, Brussels, Belgium, July 20–24, volume 1445 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, New York, 1998.

Class Refinement for Sequential 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
alcc@cin.ufpe.br www.cin.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: class refinement, modular specification and verification, inheritance and dynamic binding, refinement calculi, semantics

1 Introduction

This extended abstract describes progress in an ongoing project on refinement calculus for sequential Java. Predicate transformer semantics is being used to validate correctness-preserving transformations for use in program development, verification, design refactoring, and compilation. We focus here on the semantics and its application in showing soundness of forward simulation for class refinement, the foundation of behavioral subclassing.

This section is an overview of project objectives and recent progress. Section 2 addresses the language and its semantics. Section 3 discusses class refinement, Section 4 presents our ideas for future work.

Our work is being done in the context of a collaboration involving others at UFPE (P. Borba and A. Sampaio) and Birmingham (U. Reddy), and our research assistants.[1] Our long-term goal is development of tools and methods for specification, construction, modular verification, restructuring, and compilation of Java programs. Current work uses an idealized language ROOL based on the sequential part of Java.

Refinement calculus is the unifying framework for the work. In refinement calculi, the specification statement $x : [pre, post]$ is treated as an "imaginary command". For commands c and c', the algorithmic refinement $c \sqsubseteq c'$ means that c' satisfies any specification that c does. Ordinary correctness is expressed using specification statements: we have that $x : [\phi, \psi] \sqsubseteq c$ holds just if c meets the specification "modifies x, requires ϕ, ensures ψ". Refinement laws formalize development by stepwise refinement from specifications [Mor94].

One of our objectives is to extend this method to encompass object-oriented programs, and in particular design patterns and refactoring transformations [Fow99], including those that involve several classes at once. In a case study applying our results, we restructure an object-oriented application to follow a

[1] The work is funded by National Science Foundation under Grant No. 9813854, and by CNPq under grants 520763/98-0 and 680032/99-1.

layered architecture [VB99]. Borba and his students are developing a tool to automate the application of design transformations.

Specification statements, including the special cases known as assertions and assumptions, provide flexible annotation of program fragments. This is useful not only for verification but also for static checking [DLNS98] and program transformation. Sampaio, Cavalcanti, and their students are developing a compiler based on the normal-form approach [HHS93,Sam97], which exploits specification statements in transformation of code fragments. In the past year, a normal form has been devised for a virtual machine based on JVM. Compilation is based on normal-form laws, and some of these have been proved using our semantics.

A major objective is to derive design and compilation laws from basic laws proved sound in predicate transformer semantics [BS00]. Weakest precondition semantics is of direct use in verification tools and it is well suited to proving refinement laws. Eventually we plan to prove soundness of this semantics with respect to an operational semantics, hopefully one already developed by another research group.

To this end, and in order to check proofs of laws and of results discussed in the sequel, we are using PVS to encode the typing system and semantics of our language. The encoding is purely definitional. We are using a deep embedding of program expressions including predicate in specification statements. In accord with the new semantics described in Section 2, commands act on state sets in PVS so this part is a shallow embedding.

Many design laws involve data refinement, for which we use an intrinsic definition [HHS86,dRE98], and behavioral subclassing, which is similar to a data refinement of coexisting classes. The primary means for establishing data refinement and behavioral subclassing is (forward) simulation. The existing literature falls short of the simulation results we need. Our new results on soundness and preservation of simulation are the main topic in the sequel.

2 Syntax and semantics

A program in our language is a sequence *cds* of Java-like class declarations followed by a main program c whose free variables may include objects of classes in *cds*. Attributes can be private, protected, or public, like in Java, and they can be mutually recursive. Methods are regarded as public. Mutual recursion between methods is not allowed, to simplify the semantics of method calls and the proof of laws. Methods are defined as parameterized commands [Bac87,CSW99] using call by value, result, and value-result (with copy semantics).

In [CN99,CN00a] we defined a weakest precondition semantics for ROOL. In that work, we regarded a predicate transformer as a function on formulae. We extended traditional weakest precondition semantics and gave an account of method calls that is both abstract and operationally intuitive. This semantics is appropriate for the proof of refinement laws, a work that is well under way [BS00].

For the proof of the soundness of simulation for data refinement, however, we find the syntactic approach to predicates to be a problem. In this context, it

is natural to consider the coupling invariant (or rather, a simulation relation) to be a formula relating the private attributes of the abstract and concrete classes. The proof of soundness of simulation requires a comparison of programs that differ only by the fact that the concrete class is substituted for the abstract one. We cannot, however, say that the semantics of the fixed client classes is equal in both programs. Since their semantics depends on the semantics of methods defined in the simulated classes, the proper relation between them is that of a simulation as well. To define this simulation, we need what we call a generalized coupling invariant to relate states of client classes.

We find it difficult to give a definition for this generalized invariant syntactically, but on the other hand, its definition as a relation on states is very intuitive and straightforward [CN00b]. Also, a data refinement proof technique should involve the definition of the coupling invariant by the developer, but not the definition of the generalized coupling invariant. So there is not really a justification to have it as a formula. For this reason, we have given a new semantics to our language where predicates are regarded as sets of states, and predicate transformers as functions on these sets.

The definitions in this new semantics are very similar to those of our previous work. We use type-theoretic techniques to organize the semantic definitions. If a command c can occur in the methods of a class N, we use a typing judgement $\Gamma, \Sigma, N \rhd c$. The typing environment Γ records the classes in context, including N, and the signature Σ includes the variables in scope for c: attributes of N, parameters, and local variables. The typing rules reflect Java's restrictions on scope and subsumption. The semantics is defined by induction on typing derivations.

As expected, the challenge was the definition of the semantics of method calls. As before, we have an environment that records the semantics of methods and that is defined by a fixpoint construction. The semantics recorded is that of the behaviour of the method when called from inside the class where it is available. We use this semantics directly to define the meaning of calls $\mathbf{self}.m(e)$. For a call of the form $x.m(e)$, it must be adapted.

At the point where the call $x.m(e)$ occurs, the state space includes x as well as attributes of the calling object, parameters of the calling method, and locals of the calling method. In a state where the dynamic type of x is N', the environment η gives a meaning $\eta N' m$ for the called method, but that meaning acts on the state space consisting of attributes of N' and parameters of m. So we have to adjust the postcondition at the point of call so that $\eta N' m$ is applicable. Roughly, this adjustment extracts the attributes of x to get a state of the right kind and ensures that state variables other than x are unchanged. The definition for a pre-state σ and a postcondition ψ is as follows.

$$\sigma \in [\![\Gamma, \Sigma, N \rhd x.m(e) : \mathbf{com}]\!]\eta\ \psi \Leftrightarrow \{x\} \cup \mathit{rvrargs} \lhd \sigma \in \mathit{pt}\ (\mathit{adapt}\ \sigma\ \psi)$$

The environment η provides the transformer pt determined as $pt = \eta\ N'\ m\ \mathit{arglist}$, where N' is the class of x defined by σ, and *arglist* is the list of arguments resulting from evaluating the expressions e in σ. The predicate transformer pt is for a local signature that contains only the attributes of N' and the parameters. On

the other hand, the predicate ψ is on Γ, Σ, N, where Σ is the state space of the caller. As already said, we need to reconcile these differences before applying pt. This is the role of the function *adapt*. The method call can only affect the value of x and of the result and value-result arguments *rvrargs*. We require that the state resulting from the domain restriction ($\lhd$) of σ to x and *rvrargs* satisfies the precondition. The function *adapt* considers the conjunction of ψ with the predicate that requires that the value of all variables, except x and those in *rvrargs*, are the same as in σ. Moreover, it transforms the resulting predicate into another one on the attributes of N' and on the result and value-result parameters, by extracting the attributes of N' (or one of its subclasses) from $\sigma\ x$ and the value of the parameters from the arguments. This new semantics combines elements from [CN00a] and [Nau00].

3 Class Refinement

Algorithmic refinement of programs and commands is defined in the usual way as the pointwise order on predicate transformers. In [CN00a], we define two relations of class refinement. Here, we are focusing on the relation $cds \rhd cda \preccurlyeq_{=} cdc$ that captures the situation in which the abstract class *cda* is data refined by the concrete class *cdc* in the context of the sequence of class declarations *cds*. They both introduce the same class *Ns* with the same superclass.

Definition 1 (Class Refinement). *For a sequence of class declarations cds, and class declarations cda, and cdc, that introduce a class called Ns, for instance, we define* $cds \rhd cda \preccurlyeq_{=} cdc$ *if and only if*

- *the sequences of class declarations cds cda and cds cdc are both well-formed;*
- *for all commands c that use only methods in cds and cda and whose global components have types that are Ns-free, if c is a well-typed main program for cds cda, then*
  - *c is well-typed for cds cdc; and*
  - $(cds\ cda \bullet c) \sqsubseteq (cds\ cdc \bullet c)$.

A sequence of class declarations is well-formed if all methods, or rather, the commands in their bodies, are well-typed and there is no mutual recursion. The global components are the free variables, and, inductively, components of attributes of the object-valued free-variables. Intuitively, a type is N-free if any variable declared to have such a type cannot have attributes of the class N.

If c has global components that are not N-free, then the program refinement $(cdscda \bullet c) \sqsubseteq (cdscdc \bullet c)$ is not even well-defined because the programs act in different state spaces. For this reason, no global variables of object types are allowed in the result of [Nau01b], which is the closest result in the literature to what we need. There, structural subtyping is used, so there is no way to define a notion like N-free.

Forward simulation (including abstraction functions) is the standard proof technique for class refinement. We define class simulation in the context of private

attributes *avs* and *cvs* of *cda* and *cdc*, respectively, and of a coupling invariant *ci* defined as a relation from states of *cda* to states of *cdc*. The classes *cda* and *cdc* are assumed to provide exactly the same methods.

Coupling invariants have to satisfy certain healthiness conditions. For instance, only states for the same class can be related. Also, the initial states of the classes are related. More stringent conditions are motivated by the proof of soundness of simulation and are discussed later on.

Simulation for predicate transformers is defined in the usual way [GM91], but in terms of a generalized coupling invariant. First, if the class declarations *cda* and *cdc*, or rather, the states of these classes, are related by the coupling invariant *ci*, we define a relation $ogci\ T$, coupling values of a type T. If the type T is primitive, then $ogci\ T$ is the identity: the values of such a type are the same in both contexts. If T is either *Ns* or one of its subclasses, then $ogci\ T$ is the coupling invariant itself. Finally, if T is a class N that does not inherit from *Ns*, then it has the same attributes in both contexts. In this case, we relate an object o of N in the context of *cda* to an object o' in the context of *cdc*, if the values of the corresponding attributes of o and o' are related themselves.

The definition of the generalized coupling invariant for states is shown below.

Definition 2 (Generalized Coupling Invariant). *For a class N and local variables in scope vs, we define gci N vs to relate states σ for N and vs in the context of cda with states σ' for the same class and local variables, but in the context of cdc.*

$$
\begin{array}{ll}
(\sigma, \sigma') \in gci\ N\ vs \Leftrightarrow (\alpha(vs) \lhd \sigma, \alpha(vs) \lhd \sigma') \in ci \wedge & \\
\quad \forall x : \alpha(vs) \bullet (\sigma\ x, \sigma'\ x) \in ogci\ T & \text{if } N \text{ is a subclass of} Ns \\
(\sigma, \sigma') \in gci\ N\ vs \Leftrightarrow dom\ \sigma = dom\ \sigma' \wedge \sigma\ myclass = \sigma'\ myclass \wedge & \\
\quad \forall x : dom\ \sigma \setminus \{myclass\} \bullet (\sigma\ x, \sigma'\ x) \in ogci\ T & \text{otherwise}
\end{array}
$$

where T is the type of x in the context of N.

If N is a subclass of *Ns* we cannot simply define $gci\ N\ vs$ to be *ci* because of the extra local variables *vs*. If we disregard them, by considering the states $\alpha(vs) \lhd \sigma$ and $\alpha(vs) \lhd \sigma'$, then we can require the resulting states to be related by *ci*. The set $\alpha(vs)$ contains the local variables, as opposed to *vs* which is their declaration. We use the operator $\lhd$ (domain subtraction) to remove those variables from the states. The values assigned to the variables of *vs* have to be related by *ogci*. For the case in which N is not a subclass of *Ns*, we require the states to give values to the same variables ($dom\ \sigma = dom\ \sigma'$), to be for the same class ($\sigma\ myclass = \sigma'\ myclass$), and finally give related values to corresponding attributes. Besides declared attributes, a state σ has a special attribute *myclass* that designates its class. The states for a class include all the states for its subclasses.

To define simulation for the classes *cda* and *cdc* we consider the method

environments η and η' determined by *cds cda* and *cds cdc*.

Definition 3 (Class Simulation). *We define*

$$cds, avs, cvs, ci \rhd cda \preccurlyeq cdc$$

if and only if for each method m of cda and cdc, we have that

$$cds, cda, cdc, avs, cvs, ci, Ns \rhd (\eta\ Ns\ m) \preccurlyeq (\eta'\ Ns\ m)$$

We require that the meaning recorded in η for each method of *cda* and *cdc* is simulated by the meaning recorded in η'.

The meaning of a method recorded in the environment is a curried function from argument values to predicate transformers. Simulation for these functions is defined in terms of simulation of predicate transformers. We require that if the corresponding arguments are related by simulation, the resulting predicate transformers are as well. Simulation of arguments amounts to simulation of values, for value arguments, and the identity, for variables passed by result or value-result.

Our main theorem is stated below.

Theorem 1 (Soundness of Simulation). *If* $cds, avs, cvs, ci \rhd cda \preccurlyeq cdc$, *then* $cds \rhd cda \preccurlyeq_{=} cdc$.

The proof of this theorem relies mainly on two facts. The first is preservation: the semantics of the commands of the client classes of *cda* and *cdc* are related by simulation. This implies simulation for any main program. The second is an identity extension lemma: the generalized coupling invariant is the identity when the global components in context are *Ns*-free. Therefore, simulation of a main program implies algorithmic refinement, as required by Definition 1.

The identity extension result is simple and rather straightforward. The proof of preservation, on the other hand, brought to light a few surprises. The syntactic approach to the semantics requires the inclusion of equality on objects as a primitive function. We need that to define, for instance, the semantics of assignment. Such an expression, however, does not preserve data-refinement as it relies on equality of private attributes. Luckily it is not needed in the present semantics and it was eliminated from the language.

For variable blocks, result and value-result parameterization, and specification statements, the coupling invariant has to be surjective. The representation of an object value has to include values of private attributes, even though they are hidden. The semantics of a variable block, for instance, considers all initial values that a local object variable can have, including the different values for its private attributes. If a variable block declares a variable whose type is that being refined, then to relate the concrete block to the abstract block, we have to relate every possible concrete value of the variable to a corresponding abstract value. This requires the coupling relation to be surjective. This requirement is unnecessary, and incomplete, for simple imperative programs [HHS86,dRE98].

A way around this problem is to consider that variables are initialized. In that case, the semantics has to consider only those initial values, and the coupling invariant only needs to be surjective for values that can be expressed in

the language. The visibility restrictions and the simulation properties ensure that differences in values of hidden attributes are not relevant. This approach, however, does not work for specification statements.

We are going to investigate a solution in which each class has an invariant and the semantics quantifies over objects satisfying this invariant only. The coupling invariant is defined as a relation on states that satisfy the invariant and the surjectivity restriction is weaker. The user has to provide class invariants and discharge the corresponding proof obligations. Nevertheless, class invariants are normal practice and have independent justification. Another alternative is to change the semantics to quantify over object values obtained by applying the methods of its class to the initial values defined by the constructor. In other words, we use the weakest invariant determined by the program, rather than requiring an explicitly declared invariant.

Angelic variable (logical constant) blocks only preserve data refinement if the coupling invariant is total. If the coupling invariant is not total, in the concrete counterpart of the block, the angelic choice is restricted. As an example, consider the block (**avar** $x : T \bullet : [x = v, \mathrm{true}]$) using a specification with empty frame. In the abstract context, the block behaves like **skip** as the angelic choice can succeed in establishing the precondition of the specification statement by choosing x to be v. If v does not have a concrete counterpart, however, the concrete block is (**avar** $x : T \bullet : [\mathrm{false}, \mathrm{true}]$), which behaves like **abort**. The approaches above can also be used to avoid the totality restriction on coupling invariants.

In summary, forward simulation is sound for all the program constructs. To extend soundness to specification statements, uninitialized variable blocks, result and value-result parameters, and angelic variables, however, we need surjectivity and totality with respect to some form of class invariant.

4 Future Work

An immediate topic for further work is the investigation of the alternatives pointed out in the previous Section to generalize our results to arbitrary coupling invariants. Besides pursuing these approaches, we are going to adapt our results for the relation $cds \rhd cd \preccurlyeq\!\!\!\neq cd'$. This is the second class refinement relation introduced in [CN00a], which captures the situation in which cd and cd' introduce classes of different names. This subsumes the relation of behavioural subclassing.

Besides the specific goals of our project, we believe that our work complements the work of others in various ways. In particular, we are using a semantic model to justify simulation techniques that are often postulated as means to achieve behavioral subclassing. As a specific example, we plan to work with Gary Leavens to interpret the core constructs of JML using our semantics. On this basis, we expect to justify JML rules for behavioral subclassing.

In the first phase of our project we decided that the scope of the language would include core features of sequential Java, including visibility controls and recursion, but excluding concurrency, exceptions, and most contentiously,

pointers. Meanwhile, Reddy and his student Hongseok Yang have worked on modular reasoning for pointer programs, extending recent work of Reynolds [Rey01,RO01,IO01,Yan00,ROY01].

This work is based on a non-standard logic, but we have recently shown how a form of spatial conjunction can be used in the setting of standard logic and predicate transformers [Nau01a]. This work focuses on reasoning about fine-grained manipulation of pointers. In particular, it localizes reasoning using partitions of the heap that can have two-way interlinking, unlike disciplines such as Universe Types [MPH00] which focus on modular reasoning at the level of classes. In the next phase of our project we plan to deal with pointers using Universe Types together with spatial conjunction.

Variations of the specification statement are used in JML [LLP+00] as "model programs" which are particularly useful in specifying calling patterns of methods, including callbacks [BW99,RL00]. Up to now, our specification constructs include only the specification statements and "angelic variables" (logical constants) of Morgan's refinement calculus [Mor94]. In the next phase, we plan to add abstract attributes and dependencies for modular specification [LN00,Mül01].

References

[Bac87] R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo - Finland, 1987. Ser. A No. 55.

[BS00] P. H. M. Borba and A. C. A. Sampaio. Basic Laws of ROOL: an object-oriented language. In *3rd Workshop on Formal Methods*, pages 33 – 44, Brazil, 2000.

[BW99] Martin Büchi and Wolfgang Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, August 1999. http://www.abo.fi/~mbuechi/publications/TR297.html.

[CN99] 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.

[CN00a] A. L. C. Cavalcanti and D. A. Naumann. A Weakest Precondition Semantics for Refinement of Object-oriented Programs. *IEEE Transactions on Software Engineering*, 26(8):713 – 728, August 2000.

[CN00b] A. L. C. Cavalcanti and D. A. Naumann. Simulation and Class Refinement for Java. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, Fernuniversität Hagen, 2000. Available from `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.

[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.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report Report 159, Compaq Systems Research Center, December 1998.

[dRE98] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GM91] P. H. B. Gardiner and C. C. Morgan. Data Refinement of Predicate Transformers. *Theoretical Computer Science*, 87:143 – 162, 1991.

[HHS86] J. He, C. A. R. Hoare, and J.W. Sanders. Data refinement refined (resumé). In *European Symposium on Programming*, volume 213 of *Springer LNCS*, 1986.

[HHS93] C. A. R. Hoare, J. He, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.

[IO01] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*. ACM Press, 2001.

[LLP+00] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, October 2000.

[LN00] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. Technical Report 160, COMPAQ Systems Research Center, November 2000.

[Mor94] Carroll Morgan. *Programming from Specifications, second edition*. Prentice Hall, 1994.

[MPH00] Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *ECOOP Workshop on Formal Techniques for Java Programs*. Technical Report 269, Fernuniversität Hagen, 2000. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[Mül01] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[Nau00] David A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtyping. *Science of Computer Programming*, 2000. To appear.

[Nau01a] David A. Naumann. Ideal models for pointwise relational and state-free imperative programming. In *Principles and Practice of Declarative Programming*, 2001. `http: //www.cs.stevens-tech.edu/~naumann/relambda.ps`,.

[Nau01b] David A. Naumann. Soundness of data refinement for a higher order imperative language. *Theoretical Computer Science*, 2001. To appear.

[Rey01] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*. Palgrave, 2001.

[RL00] Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Proceedings of OOPSLA 2000*, October 2000.

[RO01] John C. Reynolds and Peter W. O'Hearn. Reasoning about shared mutable data structure. Slides from invited talk at SPACE 2001, January 2001.

[ROY01] John C. Reynolds, Peter W. O'Hearn, and Hongseok Yang. Local reasoning about shared mutable data structure. Slides for invited talk at APPSEM 2001 workshop, 2001.

[Sam97] Augusto Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *Algebraic Methodology and Software Technology*. World Scientific, 1997.

[VB99] E. Viana and P. Borba. Integrando Java com Bancos de Dados Relacionais. *III Simpósio Brasileiro de Linguagens de Programação*, pages 77–91, May 1999.

[Yan00] Hongseok Yang. An example of local reasoning in BI pointer logic:the schorr-waite graph marking algorithm. Draft, December 2000.

On the Role of Invariants in Reasoning about Object-Oriented Languages

Joachim van den Berg, Cees-Bart Breunesse, Bart Jacobs, Erik Poll

Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{joachim,ceesb,bart,erikpoll}@cs.kun.nl

Abstract The aim of this paper is to clarify the different roles that class invariants play in the verification of object-oriented programs, namely in method specifications as *proof obligations for method implementations* (assume the precondition and then prove the postcondition) and in specifications as *assumptions on method invocations* (prove the precondition, and then assume the postcondition), in order to prove the correctness of other methods. The standard proof obligation is that individual methods preserve the invariant of their class, as part of the method's specification. When trying to prove such an obligation one may have to be careful about the invariants that are assumed to be part of the precondition, at least if one wishes to use the same specification in other proofs. This is what we call the *conservative* approach. There is an associated more *liberal* approach which really makes a distinction between specifications as obligations and as assumptions. In the latter case the requirement to prove all invariants in the precondition is dropped. This considerably simplifies the verification work, but relies on a suitable meta-result about non-disturbance of invariants.

1 Introduction

Preconditions, postconditions and invariants are universally accepted as the basic ingredients for specification of OO programs. Invariants for classes play an important role in object-oriented languages ([11,14]) because they often express key data integrity properties, such as: this integer field is always non-negative, or this reference is never null. The idea is that invariants should always hold. But things are not so simple. Consider for example the invariant `i+j==0`. It does not hold in between the two statements `i++;j--`, even though it will hold after the composition, assuming it holds before. It is inevitable that some invariants are temporarily invalidated in a method body, but they should be re-established at some point.

Our perspective here—unlike for example [4]—is tool-assisted verification of object-oriented programs. More specifically, proving specifications for method implementations. The specifications we consider are written in the behavioural interface specification language JML, developed by Leavens *et al.* [6]. The implementations are written in Java. We have developed a special compiler, called

the LOOP tool (see [1]) which translates these JML specifications plus Java implementations into their semantics in the language of the proof tool PVS. Specifications become special predicates, which we try to prove for the (translated) implementations. In this context, we are forced to be very systematic and precise about the meaning of invariants—which is a delicate matter. The purpose of this short note is to make the main problems explicit, and to present the options that one has.

The verification of a specification for a method proceeds by stepping through the methods body via applying appropriate Hoare-like rules for JML, see [5], for each individual statement in this body. These statements may consist of method calls, like in `void m() { ... o.k(); ... }`. What we then do is *use* the specification given for the method `k`—which involves in particular showing that the precondition of this specification of `k` holds. The first point we wish to emphasise is that there are thus two different roles for method specifications in verification, namely as a proof obligation for the method implementation, and as an assumption about a method invocation. Ideally, specifications have the same meaning in these two roles, but that is possibly not the most convenient. Invariants play a key part in making such a distinction.

Method specifications in JML contain several clauses for the various modes of termination in Java, like divergence, normal termination, and exceptional termination (see [5] for details). These differences are not so relevant in this paper, and so we shall simplify this matter and use the standard triple notation from Hoare logic,

$$\{\ \mathtt{pre}\ \}\ \mathtt{m}\ \{\ \mathtt{post}\ \} \tag{1}$$

for partial correctness. Such a triple expresses that if the precondition `pre` holds in the pre-state, and if the method `m` terminates normally, then the postcondition `post` holds in the post-state. However, the triple (1) is too simple, because it does not involve the invariant $\mathtt{I_A}$ of the class `A` in which the method `m` is defined. Thus the triple (1) should be:

$$\{\ \mathtt{I_A}\ \&\&\ \mathtt{pre}\ \}\ \mathtt{m}\ \{\ \mathtt{I_A}\ \&\&\ \mathtt{post}\ \} \tag{2}$$

This is standardly taken as the meaning of a method specification, and is what should be proved for the method.

2 Invariants and inheritance

The two main requirements[1] of the behavioural approach to subtyping [10] are the following. For a subclass `B` of a class `A`,

1. The invariant $\mathtt{I_B}$ of `B` is stronger than the invariant $\mathtt{I_A}$ of `A`, *i.e.* $\mathtt{I_B} \Rightarrow \mathtt{I_A}$.
2. For each method `m` in class `A` that is overridden in the subclass `B`, the specification of `m` in `B` is stronger than the one in `A`. Usually this is expressed by the pair of implications:

$$\mathtt{pre_A} \implies \mathtt{pre_B} \qquad \text{and} \qquad \mathtt{post_B} \implies \mathtt{post_A}. \tag{3}$$

[1] Omitting constraints.

Notice that the second condition does not involve invariants. Simply adding them in the straightforward way, namely as:

$$\mathtt{I_A} \mathbin{\&\&} \mathtt{pre_A} \implies \mathtt{I_B} \mathbin{\&\&} \mathtt{pre_B} \qquad \text{and} \qquad \mathtt{I_B} \mathbin{\&\&} \mathtt{post_B} \implies \mathtt{I_A} \mathbin{\&\&} \mathtt{post_A} \tag{4}$$

is problematic, because the first implication does not hold in general—because $\mathtt{I_B}$ can be really stronger than $\mathtt{I_A}$.

The practice of actual verification [3,2] has taught us that a slightly different approach is more appropriate (and effective), in which the above implications (3) and (4) are not required. In order to explain this alternative we distinguish notationally between the version $\mathtt{m_B}$ of `m` in `B`, and the (overridden) version $\mathtt{m_A}$ in `A`. The meaning of the specification of $\mathtt{m_A}$ is as in (2) above. For $\mathtt{m_B}$ we use a conjunction of two requirements:

$$\begin{array}{l} \{\ \mathtt{I_B} \mathbin{\&\&} \mathtt{pre_B}\ \}\ \mathtt{m_B}\ \{\ \mathtt{I_B} \mathbin{\&\&} \mathtt{post_B}\ \} \\ \{\ \mathtt{I_B} \mathbin{\&\&} \mathtt{pre_A}\ \}\ \mathtt{m_B}\ \{\ \mathtt{I_B} \mathbin{\&\&} \mathtt{post_A}\ \} \end{array} \tag{5}$$

Notice that the first triple is the analogue of (2) for $\mathtt{m_B}$. The second one expresses that $\mathtt{m_B}$ should also satisfy the specification of the superclass `A`, *but with the invariant of its own class*. Indeed, this is what is often needed in practice, typically when the invariant $\mathtt{I_B}$ expresses certain safety conditions that are essential for the correct behaviour of $\mathtt{m_B}$. We shall use the second triple in (5) as interpretation of the requirement 2. of behavioural subtyping in the beginning of this section, namely that an overriding method should also satisfy the specification of the overridden method.

Several further remarks should be made at this point.

1. The formulations (5) are convenient in verification because they are in a form that can directly be used in proofs, when specifications are used as assumptions.
2. The first requirement in (5) follows from the second if we can establish the following adaptations of (4).

   $$\mathtt{I_B} \mathbin{\&\&} \mathtt{pre_A} \implies \mathtt{pre_B} \qquad \text{and} \qquad \mathtt{I_B} \mathbin{\&\&} \mathtt{post_B} \implies \mathtt{post_A}.$$

   And this is what we of course do, if possible, in order to prevent going through the method body in another lengthy proof.
3. One distinguishing feature of object-oriented languages (with inheritance) is that objects have both a static type and a dynamic (run-time) type—where the latter is a subtype of the former. This raises the question whether the invariant of an object refers to the invariant of the static or of the dynamic type of an object. This difference is relevant if Java's `instanceof` is used to get more information about an objects dynamic type. In order not to make matters more complicated than they already are, we shall ignore the difference between static and dynamic invariants and shall simply write `Inv(o)` for the invariant of an object `o`.

4. Another question is whether late binding should be used to interpret the (pure, side-effect free) methods that may occur in invariants. For example, in

```
class A {
      int i;
      //@ invariant i > min();

      //@ ensures \result >= 0;
      /*@ pure */ int min() { .. }
}

class B extends A {
      //@ ensures \result >= 10;
      /*@ pure */ int min() { .. }
}
```

   do we know, for an object `b` of class `B`, that `b.i > 0` or—by late binding—`b.i > 10` ?

3 Invariants for objects (other than `this`)

Besides the invariant of the class in which the method is defined (the invariant of `this`, as it is sometimes called), the correctness of a method's implementation may also rely on invariants of other objects used in the method body, such as parameters and (possibly static) fields. Therefore, these invariants are also important to prove the correctness of the implementation.

Like the class invariant (of `this`) that is assumed to hold in the pre-state, in most cases also the invariants of the parameters and relevant fields are needed in this state. The proof obligation for a method `m` with parameters $\overrightarrow{a}$ and defined in class `A` will then become

$$\{\ \texttt{Inv(this) \&\& Inv(}\overrightarrow{a}\texttt{) \&\& Inv(}\overrightarrow{f}\texttt{) \&\& pre}\ \}\ \texttt{m(}\overrightarrow{a}\texttt{)}\ \{\ \texttt{Inv(this) \&\& post}\ \},$$

where `Inv(this)` is what we have written as $\texttt{I}_{\texttt{A}}$ before (with `A` the class of `m`), `Inv(`$\overrightarrow{a}$`)` are the invariants of the reference parameters in $\overrightarrow{a}$, and `Inv(`$\overrightarrow{f}$`)` are the invariants of all the relevant reference fields. How it is determined which objects are relevant is not so important at this stage.

It may also be the case that a method returns a reference to an object[2]. This result object should also satisfy its invariant. Therefore, the proof obligation is further strengthened to:

$$\begin{array}{l}\{\ \texttt{Inv(this) \&\& Inv(}\overrightarrow{a}\texttt{) \&\& Inv(}\overrightarrow{f}\texttt{) \&\& pre}\ \}\\ \quad\ \texttt{m(}\overrightarrow{a}\texttt{)}\\ \{\ \texttt{Inv(this) \&\& Inv(\textbackslash result) \&\& post}\ \},\end{array}$$

[2] Also, the method may produce an exception object, which should also satisfy its invariant. But this case does not occur in our simplified Hoare logic dealing only with normal termination.

where `Inv(\result)` denotes the invariant for the return value.

In an object-oriented context, modular verification ([7,12]) is important, because one wishes correctness results to be robust with respect to addition of subclasses. Consider the following example in which a method `m` has a reference parameter `o` of class `A`:

```
void m (A o) { o.i=1; o.k(); }
```

One would like the specification of `m` to be correct for all possible future subclasses of class `A`—objects of which may be passed as actual parameter. In order to achieve such correctness, it should be assured that each subclass of `A` behaves as `A`. The assignment `o.i=1`, however, may cause trouble: it might break a possibly stronger invariant of a subclass of `A`.

In general terms, assignments may disturb the invariants of other objects via exposure of the state of an object, *e.g.* via access to public fields or aliasing. In order to control this, a meta-result about non-disturbance of other invariants is needed. Müller and Poetzsch-Heffter [13,12] propose to use suitable "universes" of objects that put restrictions on leaking references to the outside world. This will be needed to make our verifications modular.

4 Invariants in verification

It is common in the literature on object-oriented specification and verification to assume invariants, whenever needed. Explicitly, in the words of Liskov & Wing [10, p.13]: "We omit adding the invariant, because if it is needed in doing a proof it can always be assumed, since it is known to be true for all objects of its type." Such a hand-waving approach is not possible in a formalised semantics for theorem proving, and must be described explicitly. This will be done by explicitly distinguishing between proving and using method specifications.

For expository reasons we first consider some naive rules for invariants, mainly to illustrate the subtleties involved.

4.1 The black box approach

In an ideal world, every object would be a black box, that only interacts with its environment by method invocations. In such a setting, every object would take care of maintaining its own invariant only. The proof obligation for a method implementation would be as expected:

```
{Inv(callee) && pre}
  m()
{Inv(callee) && post}
```

At first sight, one might expect that users of a method may make the following assumption for method invocations:

```
{pre}
  callee.m()
{post}
```

Given that the object `callee` takes care of maintaining its own invariant, the caller can assume that the invariant of `callee` will hold.

However, the rule above is not correct, because the caller may have temporarily invalidated its invariant at the moment of the invocation `callee.m`. This is a problem because of possible *call-backs*: the execution of `callee.m()` may lead to other method invocations, including method invocations on the object `caller`, in which case execution of a method of `caller` would start in a state in which its invariant does not hold. The simplest scenario where this problem occurs is the case that `caller` and `callee` are the same object, *i.e.* if `callee.m()` is an invocation of the objects own methods. To prevent such problems, the `caller` should ensure that its own invariant holds whenever it invokes other methods:

$$\begin{array}{l}\{\texttt{Inv(caller) \&\& pre}\}\\ \quad \texttt{callee.m()}\\ \{\texttt{Inv(caller) \&\& post}\}\end{array}$$

This may be seen as formalisation of the (standard) solution for call-backs, see for example [14, Section 5.8]: "A simple solution would be to require all invariants of an object to be established before calling any method."

The big problem with this approach is that the underlying assumption—that objects are black boxes—is not true for typical object-oriented languages such as Java. For instance, invariants can be disturbed by assignments to public fields. More importantly, the invariant of an object typically depends not just on the states of that object (*i.e.* its fields), but often also depends on the states of other objects (in its "universe").

Another problem with this approach is that, in spite of the drastic simplifying assumption, insisting that the invariant is maintained whenever another method is invoked may be too strong a requirement in practice. As noted for instance in [14, Section 5.8], (re)establishing an invariant might require a sequence of method invocations. One place where this requirement is often unworkably strong is in constructors; typically the invariant is not established until the end of a constructor body, which means that we cannot invoke any methods (or super constructors) in constructor bodies [9].

4.2 The white box approach

Diametrically opposed to the black box approach discussed above, where every object is given the responsibility of maintaining only its own invariant, is what we call the white box approach: here every individual object is given the responsibility of maintaining all the invariants of all existing objects.

For this approach the proof obligation for a method implementation would be

$$\begin{array}{l}\{\forall i.\ \texttt{Inv(o}_i\texttt{) \&\& pre}\}\\ \quad \texttt{m()}\\ \{\forall i.\ \texttt{Inv(o}_i\texttt{) \&\& post}\}\end{array}$$

and the assumption on method invocations would be

$$\begin{array}{l}\{\forall i.\ \texttt{Inv(o}_i\texttt{) \&\& pre}\}\\ \quad \texttt{callee.m()}\\ \{\forall i.\ \texttt{Inv(o}_i\texttt{) \&\& pre}\}\end{array}$$

In this approach we know that at all visible states—all routine borders—all invariants of all objects hold. This approach has the advantage of being sound, as the proof obligation for method implementation and the assumption on method invocation are identical. However, the problem with this approach is that the proof obligation for method implementations is unworkably strong. In general there is no way of knowing whether a given assignment to a field does not invalidate the invariant of some other object. Additionally, we also have the problem mentioned in the last paragraph of the Subsection 4.1.

4.3 The liberal and conservative approach

In light of the fundamental problems with the two approaches above, we now consider two more pragmatic approaches to invariants, see Figure 1, called the liberal and conservative approach. In the conservative approach there is no distinction between obligation and assumption—although we use a slightly different formulation to emphasise that there are two objects involved when a specification is used as assumption. In the liberal approach one adds all invariants to the precondition in a proof obligation, but one does not need to establish these invariants when the specification is used as an assumption. The justification for this omission is like in Liskov & Wing [10], as cited above. More formally, it should be justified by a meta-result about non-disturbance of invariants, in the universes setting of Poetzsch-Heffter & Müller [13,12]. This is needed for the soundness of the liberal approach. But as we saw towards the end of Section 3, the meta-result is needed anyway to be able to work modularly.

	liberal	**conservative**

specification as obligation	$\{\forall i.\ \texttt{Inv(}o_i\texttt{) \&\& pre}\}$
	`m()`
	`{Inv(callee) && post}`
		{∀relevant i. `Inv(`o_i`) && pre`}
		`m()`
		`{Inv(callee) && post}`

specification as assumption	`{Inv(caller) &&`
	`Inv(callee) && pre}`
	`callee.m()`
	`{Inv(caller) &&`
	`Inv(callee) && post}`
		{∀relevant i. `Inv(`o_i`) && pre`}
		`callee.m()`
		`{Inv(callee) && post}`

Figure1. The proof obligations and assumptions for method specifications in the liberal and the conservative approach.

Some motivations for the rules in Figure 1:

- The reason for including `Inv(caller)` in the precondition of the liberal assumption rule is to allow for call-backs, as discussed in Subsection 4.1. In the conservative assumption rule, `Inv(caller)` should be included as one of the relevant objects if there is a call-back to it.
- The reason for including `Inv(caller)` in the postcondition of the liberal assumption rule is that it is needed to prove that a method implementation containing `callee.m()` preserves the caller's invariant. However, one could also rely on the modifiable clause (*i.e.* frame property) of the called method `m` to prove this; this is what has to be done in the conservative approach.

If one wishes to use a specification in the conservative approach one needs to prove explicitly that the invariants added to the precondition hold. Adding the invariants of *all* objects makes proofs unnecessarily difficult—and often impossible, when certain, irrelevant, invariants do not hold. This is the reason for restricting ourselves to what we call the invariants of all relevant objects. Still this puts a considerable burden on the verifier: in our experience this requirement gives a lot of work, which is in essence unnecessary. But the great advantage of this conservative approach is of course its soundness.

By having a weaker specification as obligation than as assumption in the liberal approach it is not hard to introduce logical inconsistencies in the back-end theorem prover, namely in cases where an invariant of an other object (than `this`) is necessary for a certain result value. Consider for example:

```
class A {
  int i;
  //@ invariant i > 0;

  //@ requires a != null;
  //@ ensures \result == true;
  boolean m(A a) { return a.i > 0; }

  //@ requires a != null && b != null;
  //@ ensures \result == true;
  boolean k(A a, A b) { b.i = 0; return a.m(b); }
}
```

Notice that `k` does not meet its specification. But it can be proven "correct" with the liberal approach: the specification of `m` as assumption does not require that the invariant for the parameters hold.

In the universe approach [13] certain constraints will have to be imposed, in order to prevent that the integer field `i` can become non-positive, *e.g.* by making it private or read-only. This disables invariant-disturbing assignments to `i`.

As mentioned, in the conservative approach only invariants of relevant objects are added to the precondition, whereas all invariants are added when proving a specification in the liberal approach. During such a "liberal" proof one quickly

restricts the invariants added to the precondition to only the relevant objects, because otherwise one pushes too strong a formula through the method body (via the rules for Hoare logic).

5 LOOP project

As stated in the introduction, the LOOP compiler translates Java classes plus JML annotations into PVS. The translation covers almost all of sequential Java, and this part is reasonably stable. The translation of JML is still under construction, but already covers a core part of JML: class invariants and constraints, method specifications including modifiable clauses, but, for instance, not yet model variables. The JML translation is being used for several case studies (notably for JavaCard), and is optimised on the basis of the resulting experiences.

Originally, the approach we used was to explicitly include invariants of objects other than `this` in pre- and postconditions, which is possible in JML using the `\invariant_for` keyword. For example, the method `m` in the example above could be specified as follows

```
//@ requires a != null && \invariant_for(a);
//@ ensures \result == true;
boolean m(A a) { return a.i > 0; }
```

to make it explicit that `m` relies on the invariant of its argument. The problem with this approach is that it quickly becomes cumbersome to have to explicitly include all these invariants in pre- and postconditions.

We then considered the conservative approach, mainly because this seemed to be the safest. It has given rise to several difficulties.

- How to determine which objects are relevant? Adding invariants for reference parameters is not problematic, but finding all other relevant objects in a method body is beyond static analysis. This is a problem in generating the semantics of a specification. ESC/Java uses some heuristic to choose the set of relevant objects [8, Sect. 2.4.1]. One way to tackle this problem might be to include a model variable `relevantObjects`,

  ```
  //@  public model JMLObjectSet relevantObjects;
  ```

  for every object, that keeps track of the set of relevant objects, *i.e.* `o.relevantObjects` is the set of objects on whose invariants the methods of the object `o` rely.
- Proving the invariants of all relevant objects when a method specification is used is very time consuming, and does not yield very much in terms of confidence because these invariants typically hold anyway.

On the basis of these experiences we are now moving to the liberal approach[3], also because there are now concrete plans to extend JML with the universe type system of Poetzsch-Heffter & Müller. The liberal approach makes verification easier, but it may lead to inconsistencies because the required meta-result about non-disturbance of invariants is not formalised in PVS. This possibly unsound approach goes very much against the very idea of formal verification, but seems to be the most pragmatic: we have assumptions which are strong enough for verifying non-trivial program properties, but which should not be abused.

6 Conclusion

Our work on tool-assisted verification of object-oriented programs forces us to be very explicit and precise about the meaning of all the constructs involved. This short note focuses on the role of invariants in this setting, and tries to clarify several issues that occur in various places in the literature, but are still confusing when it comes down to detailed formalisation.

References

1. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS01, Tools ans Algorithms for the Construction and Analysis of Software*, number 2031 in Lect. Notes Comp. Sci., pages 299–312. Springer, Berlin, 2001.
2. J. van den Berg, B. Jacobs, and E. Poll. Formal Specification and Verification of JavaCard's Application Identifier Class. In I. Attali and T. Jensen, editors, *Proceeding of the first JavaCard Workshop (JCW'2000)*, Lect. Notes Comp. Sci. Springer, Berlin, 2001. To appear.
3. 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. To appear in *Software Tools for Technology Transfer*, 2001.
4. K. Huizing, R. Kuiper, and SOOP. Verification of object oriented programs using class invariants. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci., pages 208–221. Springer, Berlin, 2000.
5. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, number 2029 in Lect. Notes Comp. Sci., pages 284–299. Springer, Berlin, 2001.
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 April 2001.

[3] We are also considering an equally unsound variation of the liberal approach which simply says (as an assumption in PVS) that all objects different from the caller (`this`) satisfy their invariant. This is even easier to use, because it avoids that the invariant assumptions at the beginning of proofs have to be carried along through the method body until the point where they are needed.

7. K.R.M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Inst. of Techn., 1995.
8. K.R.M. Leino, G. Nelson, and J.B. Saxe. Esc/java user's manual. Technical Report (2000/002), Compaq SRC, 2000.
9. K.R.M. Leino and R. Stata. Checking object invariants. Technical Report 97/007, Digital SRC, 1997.
10. B. Liskov and J. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, November 1994.
11. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.
12. P. Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001. Available at `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.
13. P. Müller and A. Poetzsch-Heffter. Universes: a type system for alias and dependency control. Tech. Rep. 279-1/2001, Fernuniversität Hagen, 2001.
14. C. Szyperski. *Component Software.* Addison-Wesley, 1998.

Formalising Dynamic Composition and Evolution in Java Systems

Claus Pahl

Dublin City University, School of Computer Applications
Dublin 9, Ireland
cpahl@compapp.dcu.ie

Abstract. The Java platform allows the dynamic establishment and closure of connections and also the composition and customisation of components at deployment time. In order to guarantee reliability and maintainability in dynamic evolving systems, we take a process-oriented view on composition and interaction. This is supported by a contract concept to formalise matching of suitable service provider and requestor.

1 Motivation

Forming a contract between service provider and service requestor can constrain the invocation of remote or unknown methods [1], leading to more reliable systems. Since dynamic loading of classes is possible in Java, objects can interact via RMI, and beans can use each others services [2] - possibly assembled and customised at deployment time -, respective means to control the use of services should be in place. A contract-based composition framework is sufficient for static systems, but systems do evolve over time. Requirements change and force contracts to be renegotiated. This can be addressed by embedding a concept of contracts into a model of change. Reasoning about the impact of dynamic composition and change is important to achieve reliability for evolving systems. We adapt a process calculus to capture the establishment and release of contracts.

2 Contracts and Connectors

The π-calculus shall be used to model the process of establishing contracts and connections between components. The π-calculus offers means to specify communication between agents in a distributed environment. Objects communicating through RMI and beans are agents in this sense. Modelling the process of change using a process calculus is justified by a similarity between mobility and evolution. Mobility in the π-calculus is defined as a change of neighbourhood, i.e., a change of the links that an agent has with its environment. In the same way evolution might require changes in connections between interacting objects or between beans. Requirements of a client, formulated using pre- and postconditions, need to be satisfied, or matched, by a service provider, formalised using the refinement calculus. Interfaces can be used to form contracts between a server

component and a client component. A requested and a provided method have to be matched based on their specifications (pre- and postconditions) to form a contract. The matching construct is refinement $\sqsubseteq$. The provider needs to satisfy the needs of the requestor, i.e., a provided method n should refine the requirements of m:

$$m \sqsubseteq n \stackrel{\triangle}{=} pre(m) \rightarrow pre(n) \wedge post(n) \rightarrow post(m) \tag{1}$$

The process of matching and creating a connector is described by:

$$\textsc{Req}\ \overline{cC}\langle m\rangle.C' | \textsc{Prov}\ cC(n).P' \stackrel{m \sqsubseteq n}{\longrightarrow} \textsc{priv}\ m\ (C' | P'\{^m/_n\}) \tag{2}$$

This rule is constrained by the refinement $m \sqsubseteq n$, i.e. it matches a service request m and a provided service n. This request is handled on a contract channel cC. This step establishes a (private) connection m between the provider and its client. Interactions between them, e.g. invocations of remote methods, can now be executed via the connector m. This rule can model contracts between client and server using RMI or between beans assembled to larger components. Connectors occur in two forms in Java. Firstly, as a remote computation, i.e., a service channel is used to invoke a remote method. Secondly, as a local computation, i.e., a data channel is used to load the class which contains the code to be executed. Connectors are an abstraction to capture *remote* and *mobile* code.

In order to formalise the constraint language within the dynamic framework, we need to see objects as entities with internal structure. We use hidden algebras to define semantical structures, embedding this into dynamic logic.

3 Management of Change

This framework for change and evolution in Java can be expanded into concepts to determine the effects of change and to manage evolving systems. Both specifications of service requests and available services might change due to changes in the overall requirements or the environment. Changes in one component might force changes in other components - change is propagated. A framework based on matching and internal correctness conditions can help to determine the effects of change. This framework can be defined based on the dynamic logic semantics used to embed pre- and postconditions. Matching is used to determine the effect of change to contracts. Internal component correctness relations form a measure for the effect of contract changes on a component implementation.

References

[1] L.F. Andrade and J.L. Fiadero. Interconnecting Objects via Contracts. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language*. Springer Verlag, LNCS 1723, 1999.

[2] S. Cimato and P. Ciancarini. A formal approach to the specification of java components. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Tech. Rep. 251, University of Hagen, 1999.

Process Algebra–Guided Design of Java Mobile Network Applications.

Marco Carbone, Matteo Coccia, Gianluigi Ferrari, Sergio Maffeis

Dipartimento di Informatica
Università di Pisa, Italy

Highly distributed networks have now become a common platform for wide-area applications which use network facilities to access remote resources and services. WEB applications distinguish themselves from traditional distributed applications mainly because they have to deal with *dynamic* and *unpredictable* changes of network environment, e.g. availability of network connectivity, lack of resources, dynamic service creation, network reconfiguration, and so on. Mechanisms to control how WEB applications can be dynamically assembled, extended and reconfigured are therefore the key programming abstractions.

We propose a programming model and a design discipline for developing Java-based (JINI–like) applications which exploit a programmable coordination language amenable to formal verification. Designers are forced to develop applications by clearly separating the computational modules from the coordination ones. The distinguishing feature of our approach is that the coordination language takes the shape of a distributed process calculus which provides mechanisms for mobility of coordinating agents and for explicit distribution of modules and their dynamic assembling over wide-are networks. The separation between computation and coordination makes the calculus amenable to be effectively analyzed with formal techniques. Hence behavioural properties can be stated and possibly established, enforcing correctness for the programmable coordination policies.

Our coordination/scripting language ED_π is based on a variant of Hennessy and Riely's Distributed π-calculus (D_π) [3], and extends it with new mechanisms to manage locations, to cope with IP name spaces, and with a new communication primitive allowing more powerful interactions (e.g. negotiation of services). ED_π is endowed with a formal LTS semantics and a structural congruence relation amenable to formal proof techniques such as bisimulation-based equivalences, control-flow analyses and many others. Related work includes several scripting/architecture-description languages: in particular Piccola [1], but also Darwin [5] and Nomadic Pict [6], and the application of formal methods to Java (here we mention [4, 2]).

Follows an example explaining some of the features of the language and its intended use: the definition of a service-fetching abstract operation $FIND$, meant to retrieve on behalf of some client sitting at location *Client*, a service *Service* starting the search from a given server location *Server*.

$\mathsf{rec}\ \mathrm{FIND}(Client, Server, Service, Result).$
$\mathsf{go}\ Server.lookUp(Client^{\uparrow}, Service^{\updownarrow}, R^{\downarrow})?$
$\quad(\mathsf{go}\ Client.\overline{Result}(R^{\uparrow}).$
$\quad\ lookUp(Client^{\uparrow},\ (\mathtt{addr},\ Service)^{\updownarrow},\ NewAddr^{\downarrow})?$
$\quad\quad(\mathrm{FIND}(Client, NewAddr, Service, Result),$
$\quad\quad\ \mathsf{go}\ Client.\overline{Result}(\mathtt{FAIL}^{\uparrow}).0))$

This process first moves ($\mathsf{go}\ Server.$), to the default server site. Then, by exploiting a conventional public channel ($lookUp$) it checks whether the service is available: $lookUp(Client^{\uparrow}, Service^{\updownarrow}, R^{\downarrow})?$ .Here, communication is a bidirectional synchronization via typed tuples and pattern matching. In the example at hand this feature allows us to disclose the client's identity if and only if the server offers the required service. In this case, the network reference to the service is returned as a result. The arrows specify for each argument of the typed tuple whether it is an input, output or synchronization parameter. If the request for the service succeeds, the process goes back to the original location and communicates the result through the channel $Result$. Otherwise it asks the server for an alternative address for the service and recursively begins the fetch protocol. Notice that typed tuples provide the mean to negotiate service requests by explicitly stating a specific range of values, e.g. the minimum level of service that components are willing to accept and the maximum level that they are able to use.

The Java implementation consists in a package providing classes for each of the process-algebra constructs, so to stress on the programmer's side the connection with the underlying formal model. Work in progress includes a security type system for the language, static analyzers and an extensive revision of the implementation.

References

1. F. Achermann, S. Kneubuehl, O. Nierstrasz. Scripting Coordination Styles. In Proc. Proc. Coordination'01, LNCS 1906, 2001.
2. S. Drossopoulou. Towards an abstract model of Java dynamic linking and verification. In Proc. TIC'2000, CMU Technical Reports, CMU-CS-00-161, 2000.
3. M. Hennessy, J. Riely. Type-safe execution of mobile agents in anonymous networks. In J. Vitek, C. Jensen, Eds. *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, LNCS State-Of-The-Art-Survey, NCS 1603, Springer, 1999.
4. S. N. Freund and J. C. Mitchell. A formal framework for the Java Bytecode Language and Verifier. In *ACM OOPSLA '99*, pp. 147–166.
5. Jeff Magee, Naranker Dulay, Susan Eisenbach, Jeff Kramer Specifying Distributed Software Architectures In. Proc. of 5th European Software Engineering Conference (ESEC '95), LNCS 989, (Springer-Verlag), 1995.
6. Sewell, P., Wojciechowski, P.Nomadic Pict: Language and Infrastructure Design for Mobile Agents. IEEE Concurrency, 2000.

Introducing Meta-Interfaces into Java

Peep Küngas, Vahur Kotkas, and Enn Tyugu

Software Department
Institute of Cybernetics at Tallinn Technical University, Estonia
{peep,vahur,tyugu}@cs.ioc.ee

In the present work we apply a formal program construction method with the aim of increasing the flexibility of Java classes. This work relates to recent results in the dynamic composition of software and component-based program development.

A meta-interface is a specification that introduces a collection of interface variables of a class and defines, which interface variables are computable from others under what conditions.

For instance, having a class $Triangle$ with methods $sinFindSide$, $sinFindAngle$ for computing based on theorem of sine: $\frac{a}{sin(A)} = \frac{b}{sin(B)} = \frac{c}{sin(C)}$, we can introduce interface variables for all angles (A, B, C) and sides (a, b, c) of the triangle and declare a meta-interface that will specify the methods that can be used. Such meta-interface could look as follows:

```
var a, b, c, A, B, C : any
rel a,A,B -> b {sinFindSide}
...
rel b,B,a -> A {sinFindAngle}
```

Here $sinFindSide$ and $sinFindAngle$ are implementations of the sine theorem in Java methods. The meta-interface just declares how these methods can be used.

After introducing an extension of the language that allows one to use equations, we can specify this meta-interface by using a shorter description and do not need the methods implemented by programmers:

```
var a,b,c, A,B,C : any
rel a/sin(A)=b/sin(B)
rel a/sin(A)=c/sin(C)
rel c/sin(C)=b/sin(B)
```

Note that the components specified in a meta-interface as of type **any** have to be actual components of the Java class and their type is determined by the Java declarations.

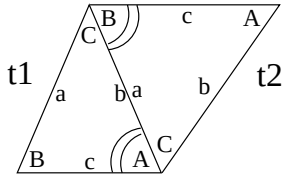
The meta-interface is used as follows: one writes a request for synthesis of a method with input $x_1, \ldots, x_m$, where $m > 0$, and output y, whereas $x_1, \ldots, x_m, y$ are variables specified in the meta-interface, for instance, in the form $x_1 \& \ldots \& x_m \rightarrow y$, and uses a prover to prove that this formula is derivable from the specification of the meta-interface.

The prover returns a sequence of rules applied in the proof, which from the synthesis point of view represents an algorithm we use to generate the program

code to solve the problem. Thus the algorithm is the co-result (or side-effect) of the proving process.

A meta-interface can be written for two different purposes. First, it may specify possible usage of the class, i.e. its derivable methods, like in our example. Second, it can be used as a specification showing how some application software should be composed from components supplied with meta-interfaces. In the latter case, a new class can be built completely from a specification of its meta-interface. The aim of introducing meta-interfaces is to make classes as components more flexible.

To illustrate the usage of meta-interfaces let us specify a class for solving a computational problem on two triangles that have one common side and one common angle (see following figure).



Values of some components of triangles are given, as can be seen from the following class code.

```
class Problem implements metaInterface {
    public static String[] metaInterface = {
        "var t1, t2 : any",
        "rel t1.b==t2.a",
        "rel t1.A==t2.B"};
    Triangle t1 = new Triangle(), t2 = new Triangle();
    public void main(String[] args) {
        Problem p = new Problem();
        p.t1.b = 5;
        p.t1.B = 70;
        p.t2.A = 40;
        String name = SSP.synthesize(p, "t1.a -> t2.b");
        SSP.exec(name, p, new Integer(4));
    }
}
```

As a result of the call $SSP.synthesize$ a new method will be synthesized (see method $xf17634$ below) that realizes the requested computational problem.

The method $SSP.exec$ executes the synthesized method and as a result modifies the object p by assigning proper value to the component $t2.b$.

```
    public void xf17634(Problem p, int i) {
        p.t1.a = i;
        p.t1.A = p.t1.sinFindAngle(p.t1.b, p.t1.B, p.t1.a);
        p.t2.a = p.t1.b;
        p.t2.B = p.t1.A;
        p.t2.b = p.t2.sinFindSide(p.t2.a, p.t2.A, p.t2.B);
    }
```