Formal Techniques for Java-like Programs (FTfJP)

Radboud University Nijmegen, ICIS-R08013

July 8th, 2008
Paphos, Cyprus

Preface

This report contains the papers presented at FTfJP'08: the 10th workshop on Formal Techniques for Java-like Programs held on July 8th, 2008 in Paphos, Cyprus. This workshop is the tenth in a series of workshops aimed at applying formal techniques to improve modern object-oriented languages. Newer languages such as Java and C# provide good platforms to bridge the gap between formal techniques and practical program development, because of their reasonably clear semantics and standardized libraries. It is our hope that this workshop will encourage the further deployment of well-considered formal techniques.

There were 16 submissions. Each submission was reviewed by 3 programme committee members. The committee decided to accept 11 papers. I thank the providers of EasyChair, the organisers of ECOOP 2008 and my fellow workshop organisers. All helped significantly with all practical aspects around the workshop organisation. Thanks also to the program committee members who put in a lot of work, all delivered their reviews in time (despite the tight reviewing schedule), and contributed to a fruitful discussion.

July 2008

Marieke Huisman
INRIA Sophia Antipolis
France

Program Committee

Elvira Albert, Complutense University of Madrid, Spain
Cyrille Artho, RCIS/AIST, Japan
Anindya Banerjee, Kansas State University, USA
Mike Barnett, Microsoft Research, Redmond, USA
Amy Felty, University of Ottawa, Canada
Paola Giannini, University of Eastern Piedmont, Italy
Rene Rydhof Hansen, Aalborg University, Denmark
Marieke Huisman, INRIA Sophia Antipolis, France (chair)
Atsushi Igarashi, Kyoto University, Japan
Bart Jacobs, University of Leuven, Belgium
Gerwin Klein, National ICT Australia, Australia
Neelakantan R. Krishnaswami, Carnegie Mellon University, USA
Matthew Parkinson, University of Cambridge, UK
Arnd Poetzsch-Heffter, University of Kaiserslautern, Germany
Tobias Wrigstad, Purdue University, USA

External Reviewers

Nick Cameron
Dino Distefano
Jean-Marie Gaillourdet
Rafal Kolanski
César Kunz
Patrick Michel
Gustavo Petri
Ina Schaefer
Thomas Sewell
Damiano Zanardino
Elena Zucca

Organisers

Marieke Huisman (chair)
Sophia Drossopoulou, Imperial College London, UK
Susan Eisenbach, Imperial College London, UK
Gary T. Leavens, University of Central Florida, USA
Peter Mller, Microsoft Research, Redmond, USA
Arnd Poetzsch-Heffter, University of Kaiserslautern, Germany
Erik Poll, Radboud University Nijmegen, Netherlands

Table of Contents

Implicit Dynamic Frames

Jan Smans, Bart Jacobs, and Frank Piessens

Katholieke Universiteit Leuven, Belgium

Abstract. The dynamic frames approach has proven to be a powerful formalism for modular specification and verification of object-oriented programs. However, the approach requires writing and checking frame annotations.
In this paper, we propose a variant of the dynamic frames approach that eliminates the need to explicitly write and check frame annotations. Reminiscent of separation logic's frame rule, programmers write accessibility predicates inside pre- and postconditions instead of writing frame annotations. From the precondition one can then infer an upper bound on the set of locations writable or readable by the corresponding method. We implemented our approach in a tool, and used it to automatically verify several challenging examples, including the iterator and observer patterns.

1 Introduction

Modular verification requires that a method's contract specifies upper bounds on the locations read or written by the method. In the dynamic frames approach [1–4], the programmer specifies these upper bounds in terms of expressions denoting sets of locations. To preserve information hiding, these expressions can involve dynamic frames, which are special specification variables that abstract over sets of locations. A disadvantage of this approach is that frame annotations must be provided for each method, and that these annotations need to checked explicitly at verification time.

The contributions of this paper are as follows:

- We propose a variant of the dynamic frames approach inspired by separation logic that eliminates the need to explicitly write and check frame annotations. Instead, framing information is inferred from accessibility predicates within pre- and postconditions. This typically leads to more compact contracts.
- The dynamic frames approach [5, Future Work] imposes global acyclicity on specification variable implementation dependencies to ensure consistency. We replace this non-modular restriction by modular rules.
- We implemented the approach in a tool, and used it to verify several challenging programs.

The remainder of this paper is structured as follows. In Section 2, we propose a solution to the frame problem based on the concept of required access sets. In Section 3, we combine the approach for framing with data abstraction through the use of method calls in specifications. Finally, we discuss our results, compare with related work and conclude in Sections 4, 5, and 6.

2 Framing

To reason modularly about a method invocation, one must not rely on the callee's implementation, but only on its specification. For example, consider the code of Figure 1(b). To prove that the assertion at the end of the code holds in every execution, one should only take into account the contracts of Cell's methods. However, the given contracts are too weak to prove this assertion. Indeed, setX's implementation is allowed to change the program state arbitrarily, as long as it ensures that x equals v on exit. In particular, the contract does not prevent c_2.setX(10) from changing the value of c_1.x.

```
class Cell module Lib {
  int x;

  Cell()
    ensures this.x = 0;
  { this.x := 0; }

  void setX(int v)
    ensures this.x = v;
  { this.x := v; }
}
```

(a)

```
Cell c1 :=new Cell();
c1.setX(5); //A

Cell c2 :=new Cell();
c2.setX(10);

assert c1.x = 5;
```

(b)

Fig. 1. A class Cell and some client code.

To prove the assertion of Figure 1(b), we must strengthen Cell's method contracts. More specifically, the contracts should additionally specify an upper bound on the set of memory locations modifiable by the corresponding methods. This problem is called the frame problem.

Various solutions to the frame problem have been proposed in the literature [6, 1, 7–9, 4] (see Section 5 for a detailed comparison). The solution proposed in this paper is as follows. Writing to or reading[1] from a memory location o.f requires o.f to be *accessible*. Accessibility of o.f is denoted as **acc**(o.f). Method implementations are not allowed to mention **acc**(o.f). In particular, they are not permitted to branch over accessibility of a memory location. As a consequence, a location o.f that was allocated before execution of a method m is only known to be accessible during execution of m if m's precondition requires accessibility of o.f. In other words, a method's precondition provides an upper bound on the set of locations modifiable by the corresponding method: the method can only modify an existing location o.f if that location is required to be accessible by its precondition. As an example, consider the revised version of the class Cell of Figure 2. Cell's constructor does not require accessibility of any location, and

[1] Requiring accessibility for reading a memory location is not strictly necessary in this section. However, this restriction will be needed for data abstraction (see Section 3).

can therefore only assign to fields of the new object. Similarly, setX only requires accessibility of this.x, and can consequently only update the location this.x.

```
class Cell module Lib {
  Cell()
    ensures acc(this.x) ∧ this.x = 0;

  void setX(int v)
    requires acc(this.x);
    ensures acc(this.x) ∧ this.x = v;
}
```

Fig. 2. A revised version of the class Cell. The implementation has been omitted as it is equal to the one of Figure 1(a).

Java implicitly allocates memory on the heap whenever a new object is created: a fresh memory location is allocated for each field of the new object. We assume that at the start of a constructor the fields of the new object are accessible. The Java language however does not provide a mechanism for explicit memory deallocation, and instead relies on garbage collection for freeing unused memory. Since the assertions in our language can only mention allocated objects, it would be safe to assume the set of accessible locations only grows. However, this assumption would rule out interesting specification patterns, where a method "captures" accessibility of a location. Furthermore, this assumption would break in the presence of concurrency where accessibility of locations can be passed on to other threads. Therefore, we use the following rule instead: a method can change accessibility of an existing location only if the location was required to be accessible by the method's precondition.

Given the new method contracts for Cell together with the rules for framing outlined above, we can now prove the assertion at the end of Figure 1(b). Informally, the reasoning is as follows. At program location A, we know the postcondition of setX holds: $\mathsf{c_1.x}$ is accessible and holds the value 5. Since $\mathsf{c_2}$'s constructor does not require anything, it cannot modify the value nor the accessibility of any existing location. In particular, $\mathsf{c_1.x}$ is still accessible and still holds 5. Similarly, the call $\mathsf{c_2.setX(10)}$ only requires $\mathsf{c_2.x}$ to be accessible, and hence the location $\mathsf{c_1.x}$ is not affected. We may conclude that the assertion, $\mathsf{c_1.x} = 5$, holds at the end in any execution.

2.1 Formal Details

Language In this section, we consider the following language.

$$
\begin{array}{lll}
\mathit{program} & ::= & \overline{\mathit{class}}\ \overline{s} \\
\mathit{class} & ::= & \textbf{class}\ \mathsf{C}\ \textbf{module}\ \mathsf{M}\ \{\ \overline{\mathit{field}}\ \overline{\mathit{method}}\ \} \\
\mathit{field} & ::= & t\ \mathsf{f}; \\
\mathit{method} & ::= & \mathit{constr} \mid \mathit{mutator} \\
\mathit{constr} & ::= & \mathsf{C}(\overline{t\ \mathsf{x}})\ \mathit{contract}\ \{\ \overline{s}\ \} \\
\mathit{mutator} & ::= & \textbf{void}\ \mathsf{m}(\overline{t\ \mathsf{x}})\ \mathit{contract}\ \{\ \overline{s}\ \} \\
\mathit{contract} & ::= & \textbf{requires}\ \phi;\ \textbf{ensures}\ \phi;
\end{array}
$$

$$
\begin{array}{lll}
t & ::= & \mathsf{C} \mid \textbf{int} \mid \textbf{bool} \\
s & ::= & e.\mathsf{f} := e;\ \mid e.\mathsf{m}(\overline{e});\ \mid \\
 & & \mathsf{C}\ \mathsf{x} := \textbf{new}\ \mathsf{C}(\overline{e});\ \mid \textbf{assert}\ \phi; \\
e & ::= & \mathsf{x} \mid e.\mathsf{f} \mid \mathsf{c} \\
\phi & ::= & \textbf{acc}(e.\mathsf{f}) \mid \phi \wedge \phi \mid \phi * \phi \mid e = e \mid \\
 & & \textbf{true}
\end{array}
$$

We distinguish two kinds of expressions: ordinary expressions e and specification expressions ϕ. The specification expression $\phi_1 * \phi_2$ is called the separating conjunction, and denotes that both ϕ_1 and ϕ_2 hold, and that the set of locations required to be accessible by ϕ_1 is disjoint from the set of locations required to be accessible by ϕ_2. We do not formalize a type system here, as it is entirely standard.

Verification Our verifier checks the correctness of a program by generating, via translation into the intermediate language BoogiePL [10], a set of verification conditions. The verification conditions are first-order formulas whose validity implies the correctness of the program. The formulas are analyzed automatically by satisfiability-modulo-theory (SMT) solvers. In the remainder of this section, we focus on novel aspects of the translation to BoogiePL.

Notation The heap is modeled in the verification logic as a map from (object reference, field name) pairs to values. For example, $\mathsf{h}[\mathsf{o}, \mathsf{f}]$ represents the value of the field f of the object o in the heap h. Allocatedness of objects is tracked by means of a boolean ghost field alloc. Postconditions in the verification logic can contain old expressions, $\mathbf{old}(\mathsf{e})$, representing the value of e in the method pre-state. $\mathsf{wf}(\mathsf{h})$ denotes whether the heap h is well-formed. Well-formedness implies (among others) that fields of allocated objects never point to unallocated objects, and that unallocated objects are not accessible. $\mathsf{succ}(\mathsf{h}_1, \mathsf{h}_2)$ denotes that heap h_2 is a direct successor (after a field update or mutator invocation) of h_1. $\mathcal{T}[\![e]\!]_h$ (respectively $\mathcal{D}[\![e]\!]_h$) denotes the translation (respectively definedness) of the expression e into the verification logic in a context where h denotes the heap. Figure 4 of appendix A defines $\mathcal{T}$ and $\mathcal{D}$ for each expression.

Accessibility To model accessibility in the verification logic, we add to each object a ghost field acc. The value of the ghost field is a map from field names to booleans, where each entry indicates the accessibility of the corresponding location. For example, accessibility of the location $\mathsf{o.f}$ is denoted as $\mathsf{h}[\mathsf{o}, \mathsf{acc}][\mathsf{f}]$. Our verifier checks before each field access that the corresponding location is accessible.

Required Access Set A property of our methodology is that a method can only modify an existing location if the method's precondition requires the location to be accessible. A naive, literal encoding of this property however does not give rise to good performance with automatic theorem provers. In particular, the combination of the literal encoding and the approach for data abstraction described in Section 3 yields verification conditions that are too hard for those automatic provers. Therefore, we propose a slightly different encoding. More specifically, we syntactically infer from each method precondition a *required access set*, i.e. an expression denoting the set of locations required to be accessible by the precondition. The required access set of a specification expression ϕ in a heap h is denoted as $\mathcal{R}[\![\phi]\!]_h$, and is defined as follows.

$$\begin{array}{ll} \mathcal{R}[\![\mathbf{acc}(\mathsf{e.f})]\!]_h ::= \{\ (\mathcal{T}[\![\mathsf{e}]\!]_h, f)\ \} & \\ \mathcal{R}[\![\phi_1 \wedge \phi_2]\!]_h ::= \mathcal{R}[\![\phi_1]\!]_h \cup \mathcal{R}[\![\phi_2]\!]_h & \mathcal{R}[\![\mathsf{e}_1 = \mathsf{e}_2]\!]_h ::= \emptyset \\ \mathcal{R}[\![\phi_1 * \phi_2]\!]_h ::= \mathcal{R}[\![\phi_1]\!]_h \cup \mathcal{R}[\![\phi_2]\!]_h & \mathcal{R}[\![\mathbf{true}]\!]_h ::= \emptyset \end{array}$$

$\mathcal{R}$ yields expressions of type "set of memory locations". To support such expressions in the verification logic, we introduced a number of functions and axioms about sets. For example, we added a parameterless function $\mathsf{empty}()$ representing the empty set, together with an axiom stating that no location is an element of the empty set. In this paper, we will use the standard mathematical set notations instead of the actual functions used in the verification logic.

We can now encode the property that a method can only modify an existing location if the method's precondition P requires the location to be accessible in terms of the required access set by adding a free postcondition to each method.

$$\begin{array}{l}\textbf{free ensures } (\forall \mathsf{o}, \mathsf{f} \bullet \textbf{old}(\mathsf{H})[\mathsf{o}, \mathsf{acc}][\mathsf{f}] \Rightarrow \\ \quad (\textbf{old}(\mathsf{H})[\mathsf{o}, \mathsf{f}] = \mathsf{H}[\mathsf{o}, \mathsf{f}] \wedge \mathsf{H}[\mathsf{o}, \mathsf{acc}][\mathsf{f}]) \vee (\mathsf{o}, \mathsf{f}) \in \mathcal{R}[\![\mathsf{P}]\!]_{\textbf{old}(\mathsf{H})});\end{array}$$

A free postcondition is a postcondition which can be assumed by callers, but does not have to be proven explicitly when verifying the implementation.

Another property of our methodology is that the difference of the required access set of the postcondition Q and the required access set of the precondition P consists of fresh locations. This property is sometimes called the swinging pivot requirement, and we also encode it by means of a free postcondition.

$$\textbf{free ensures } (\forall \mathsf{o}, \mathsf{f} \bullet (\mathsf{o}, \mathsf{f}) \in \mathcal{R}[\![\mathsf{Q}]\!]_{\mathsf{H}} \Rightarrow \neg\textbf{old}(\mathsf{H})[\mathsf{o}, \mathsf{alloc}] \vee (\mathsf{o}, \mathsf{f}) \in \mathcal{R}[\![\mathsf{P}]\!]_{\textbf{old}(\mathsf{H})});$$

3 Data Abstraction

Data abstraction is crucial in the construction of modular programs, since it ensures that internal changes in one module do not propagate to other modules. In object-oriented programs, classes typically enforce data abstraction by providing access to their internal fields only through methods.

The class Cell of Figure 2 however was not written with data abstraction in mind: (1) client code must directly access the internal field x to query a Cell object's state, and (2) Cell's method contracts are not implementation-independent as they mention the field x. Any change to Cell's internal representation, such as renaming x to y, would break or at least oblige us to reconsider the correctness of client code.

To solve issue (1), we add a getter getX to the class Cell as shown in Figure 3(a). The assertion in the client code of Figure 3(b) can then call getX instead of directly referring to x. To complete the decoupling between Cell's internal representation and client code, we should also solve issue (2) and make Cell's method contracts implementation-independent. In this paper, we use method calls in specifications to achieve this independence. That is, we allow the effect of one method to be specified in terms of other methods. For example, the method setX is specified in terms of the getter getX.

In this paper, methods used within method contracts are called *pure* methods. The body of a pure method must consist of a single return statement, and the method itself should be annotated with either **pure** or **predicate**. Normal pure

methods return an ordinary expression, and can be called both within method bodies and specifications. Predicate pure methods on the other hand return a specification expression, and can therefore only be called within specifications. Furthermore, predicates are not allowed to have preconditions. Finally, the rules for ensuring consistency of the underlying axiomatization differ for normal and predicate pure methods. This will be discussed in Section 3.1. In our running example, both getX and valid are pure methods. The former is a normal pure method, while the latter is a predicate pure method. Predicate pure methods are typically used to represent invariants and to abstract over accessibility of memory locations.

```
class Cell module Lib {
  int x;

  Cell()
    ensures valid() ∧ getX() = 0;
  { this.x := 0; }

  void setX(int v)
    requires valid();
    ensures valid() ∧ getX() = v;
  { this.x := v; }

  pure int getX()
    requires valid();
  { return this.x; }

  predicate bool valid()
  { return acc(this.x); }

  void copy(Cell c)
    requires c ≠ null;
    requires valid() ∗ c.valid();
    ensures valid() ∗ c.valid();
    ensures getX() = old(c.getX());
    ensures c.getX() = old(c.getX());
  { this.x := c.getX(); }
}
```

(a)

```
Cell c1 :=new Cell();
c1.setX(5); //A

Cell c2 :=new Cell();
c2.setX(10);

assert c1.getX() = 5;
```

(b)

Fig. 3. A revised version of the class Cell.

To prove the assertion at the end of Figure 3(b), one must show that c_2's constructor and c_2.setX(10) do not affect the return value of c_1.getX(). In other words, it suffices to show that the set of locations modified by those statements is disjoint from the set of locations c_1.getX() depends on. But how can we determine which locations influence the return value of c_1.getX()? The answer is simple.

We can deduce from the precondition of a normal pure method an upper bound on the set of locations readable by that method: a normal pure method n can only read a location o.f if n's precondition requires o.f to be accessible. Similarly, a predicate pure method q can only read a location o.f if q itself requires o.f to be accessible.

Given this property of pure methods, we can now prove the assertion at the end of Figure 3(b). Informally, the reasoning is as follows. At program location A, we know that the postcondition of c_1.setX(5) holds: c_1.valid() and c_1.getX() equals 5. Since c_2's constructor does not require anything, it can only modify fresh locations. Since the locations required to be accessible by c_1.valid() are non-fresh, c_1.valid() still holds and c_1.getX() still equals 5. In addition, we know that the locations required to be accessible by c_1.valid() are disjoint from the locations required to be accessible by c_2.valid(), since the the latter set only contains fresh locations. Since c_2.setX(10) can only modify locations required to be accessible by c_2.valid() and since this set is disjoint from the locations required to be accessible by c_1.valid(), c_1.getX() is not affected by c_2.setX(10). We may conclude that the assertion, c_1.getX() = 5, holds in any execution.

To illustrate the use of the separating conjunction in our approach, we extended the class Cell of Figure 3(a) with a copy method. copy's precondition requires that the receiver and c are "separately" valid, i.e. that both this.valid() and c.valid() hold and that their required access sets are disjoint. If we would have used a regular conjunction instead of a separating conjunction, we would not be able to prove that c.valid() holds after the assignment to this.x.

3.1 Formal Details

Language We extend the language of Section 2 as follows.

$$
\begin{array}{lll}
\mathit{method} & ::= \ldots \mid \mathit{pure} \mid \mathit{predicate} \\
\mathit{pure} & ::= \textbf{pure}\ t\ \mathsf{n}(\overline{t\ \mathsf{x}})\ \mathit{contract}\ \{\ \textbf{return}\ e;\ \} \\
\mathit{predicate} & ::= \textbf{predicate bool}\ \mathsf{q}(\overline{t\ \mathsf{x}})\ \mathit{contract}\ \{\ \textbf{return}\ \phi;\ \} \\
e & ::= \ldots \mid e.\mathsf{n}(\overline{e}) \\
\phi & ::= \ldots \mid e.\mathsf{q}(\overline{e})
\end{array}
$$

Verification For every pure method m with precondition P, postcondition Q, parameters $t_1\ x_1, \ldots, t_n\ x_n$, and body **return** E; declared in class C of module M, we introduce a function symbol #C.m that that takes the heap, the receiver, and m's formal parameters as its arguments. An invocation of a pure method is modeled by an application of the corresponding function. For example, the assertion c_1.getX() = 5 is represented as $\#\mathsf{Cell.getX}(\mathsf{H}, c_1) = 5$ in the verification logic.

To reason about the function symbol corresponding to the method m, we introduce two axioms: an implementation axiom and a framing axiom. The *implementation axiom* relates the function symbol to m's implementation. That is, applying the function equals evaluating the method body.

$$\textbf{axiom}\ (\forall \mathsf{h}, \mathsf{this}, \mathsf{x}_1, \ldots, \mathsf{x}_n \bullet \mathsf{wf}(\mathsf{h}) \wedge \mathcal{T}[\![\mathsf{P}]\!]_\mathsf{h} \Rightarrow \#\mathsf{C.m}(\mathsf{h}, \mathsf{this}, \mathsf{x}_1, \ldots, \mathsf{x}_n) = \mathcal{T}[\![\mathsf{E}]\!]_\mathsf{h});$$

The implementation axiom for normal pure methods can only be used by mutator methods in the module M, while the axiom for predicate pure methods can additionally be used in normal pure methods in M. The *framing axiom* relates the return value of m in two heaps $\mathsf{h_1}$ and $\mathsf{h_2}$. The framing axiom of normal pure methods differs slightly from the framing axiom of predicate pure methods. More specifically, the framing axiom for normal pure methods states that the return value of the pure method is equal in $\mathsf{h_1}$ and $\mathsf{h_2}$, provided the values of locations in the required access set of m's precondition are equal.

$$\begin{array}{l}\textbf{axiom}\ (\forall \mathsf{h_1}, \mathsf{h_2}, \mathsf{this}, \mathsf{x_1}, \ldots, \mathsf{x_n} \bullet \mathsf{succ}(\mathsf{h_1}, \mathsf{h_2}) \wedge \mathcal{T}[\![\mathsf{P}]\!]_{\mathsf{h_1}} \wedge \mathcal{T}[\![\mathsf{P}]\!]_{\mathsf{h_2}} \wedge \\ \quad (\forall \mathsf{o}, \mathsf{f} \bullet (\mathsf{o}, \mathsf{f}) \in \mathcal{R}[\![\mathsf{P}]\!]_{\mathsf{h_1}} \Rightarrow \mathsf{h_1}[\mathsf{o}, \mathsf{f}] = \mathsf{h_2}[\mathsf{o}, \mathsf{f}]) \Rightarrow \\ \quad\quad \#\mathsf{C.m}(\mathsf{h_1}, \mathsf{this}, \mathsf{x_1}, \ldots, \mathsf{x_n}) = \#\mathsf{C.m}(\mathsf{h_2}, \mathsf{this}, \mathsf{x_1}, \ldots, \mathsf{x_n}));\end{array}$$

The framing axiom for predicate pure methods states that the return value of the predicate is equal in both heaps, provided the values of locations in its own required access set are equal.

$$\begin{array}{l}\textbf{axiom}\ (\forall \mathsf{h_1}, \mathsf{h_2}, \mathsf{this}, \mathsf{x_1}, \ldots, \mathsf{x_n} \bullet \mathsf{succ}(\mathsf{h_1}, \mathsf{h_2}) \wedge \#\mathsf{C.m}(\mathsf{h_1}, \mathsf{this}, \mathsf{x_1}, \ldots, \mathsf{x_n}) \wedge \\ \quad (\forall \mathsf{o}, \mathsf{f} \bullet (\mathsf{o}, \mathsf{f}) \in \mathcal{R}[\![\mathsf{this.m}(\mathsf{x_1}, \ldots, \mathsf{x_n})]\!]_{\mathsf{h_1}} \Rightarrow \mathsf{h_1}[\mathsf{o}, \mathsf{f}] = \mathsf{h_2}[\mathsf{o}, \mathsf{f}]) \Rightarrow \\ \quad\quad \#\mathsf{C.m}(\mathsf{h_1}, \mathsf{this}, \mathsf{x_1}, \ldots, \mathsf{x_n}) = \#\mathsf{C.m}(\mathsf{h_2}, \mathsf{this}, \mathsf{x_1}, \ldots, \mathsf{x_n}));\end{array}$$

Framing axioms can only be used in mutator methods.

Footprint Functions In this section, we introduced a new kind of specification expression, namely predicate method invocation. What is the required access set of such an expression? One obvious solution would be to define the required access set of a predicate method invocation as the required access set of the predicate's method body. However, such a definition would expose implementation details to client code. For example, the required access set of the precondition of setX of Figure 3(a) would be the singleton containing $\mathsf{this.x}$. Yet, this is just a detail of the current implementation, and client code should not rely on it.

To address this issue, we propose introducing an extra layer of indirection. More specifically, for a predicate pure method $\mathsf{C.q}$ with body **return** ϕ; we introduce an additional function symbol $\#\mathsf{C.q_{FP}}$, called the footprint function, which represents the predicate method's required access set. The footprint function is axiomatized as if the source program contained the following method:

$$\textbf{pure set}\ \mathsf{C.q_{FP}}(\mathsf{t_1}\ \mathsf{x_1}, \ldots, \mathsf{t_n}\ \mathsf{x_n})\ \textbf{requires}\ \mathsf{C.q}(\mathsf{x_1}, \ldots, \mathsf{x_n});\ \{\ \textbf{return}\ \mathcal{R}[\![\phi]\!]_{\mathsf{H}};\ \}$$

In addition, a *footprint accessible axiom* is generated which states that all locations contained by the footprint function are accessible, provided the corresponding predicate pure method returns **true**.

Consistency The introduction of new axioms in the verification logic can potentially lead to inconsistencies. We guarantee that these inconsistencies are detected as follows. For normal pure methods, we ensure consistency by requiring they terminate: the return value can then be used as model for the function application. More specifically, we use the size of the required access set as a measure for proving termination. That is, the required access set of the precondition of method n called from the body of a pure method m must be a strict

subset of the required access set of the precondition of m. For predicate pure methods, we ensure consistency by syntactically checking that predicate pure methods are only called in positive positions. The latter check is similar to the one proposed by [11]. Contrary to Kassios' acyclicity restriction [5], our rules for ensuring consistency can modularly handle dynamically bound calls.

4 Experience

To demonstrate the approach described in previous sections is amenable to automatic static verification, we implemented it in a verifier prototype. The prototype was used to verify several (variations of) programs used in related work. The time taken to verify each program and a reference to the paper(s) containing the program is shown in Table 1. The experiments were executed on a regular laptop with a Pentium Core Duo 1.86Ghz CPU and 2 GB of ram. To discharge the generated verification conditions, we used the Z3 theorem prover [12]. To the best of our knowledge, this is the first time the observer pattern (with multiple observers and data abstraction between the subject, its observers and client code) is verified automatically. The verifier and the programs referred to in Table 1 can be downloaded from `http://www.cs.kuleuven.be/~jans/vericool2`.

	# lines	time taken	source
Cell	28	0.3	[13–15]
Interval	52	4.4	
ArrayList and Iterator	75	3.7	[1, 11]
Marriage	32	3.2	[16]
Recell, TCell, DCell	129	2.1	[17]
Observer	123	91.3	[18, 4, 15]

Table 1. Table showing the time taken (in seconds) to verify each program.

5 Related Work

In the dynamic frames approach [5, 2, 3], specification variables similar to our pure methods are used to achieve data abstraction. To solve the frame problem, Kassios proposes using dynamic frames, special specification variables that return a set of memory locations. For each mutator method (specification variable respectively), one must explicitly specify an upper bound on the set of writable (respectively readable) locations in terms of dynamic frames. In our approach, no additional frame annotations are needed as the set of accessible locations is inferred from accessibility predicates in the method precondition. Moreover, one must explicitly verify in the dynamic frames approach that implementations respect the specified upper bound and that the swinging pivot requirement holds, while in our approach this follows from the methodology. To ensure the definitions of specification variables are consistent, Kassios prohibits cycles in the implementation of specification variables. This restriction is non-modular and rules out interesting programming patterns [5, Future Work]. Instead of the

no-cycles restriction, we ensure consistency by checking termination for normal pure methods, and by syntactically checking predicates only occur in positive positions.

Banerjee *et al.* [4] recently proposed a regional logic for local reasoning about global invariants. Similarly to dynamic frames, their approach supports region expressions which are used to explicitly specify read and write effects. These effect annotations need to be checked at verification time. Our approach does not require explicit effect annotations nor verification-time checking of those annotations. Instead, at each field access we check the accessibility of the corresponding location and infer framing information from preconditions.

Our approach was heavily inspired by separation logic [11, 6, 13, 17, 15]. In particular, our accessibility expression $\mathbf{acc}(\mathsf{o.f})$ is similar to the separation logic predicate $\mathsf{o.f} \mapsto \_ * \mathsf{true}$ (denoted as $\mathsf{o}.f \hookrightarrow \_$ in [6]). To the best of our knowledge, this is the first approach based on verification condition generation and automatic, first-order theorem proving which captures separation logic's idea of inferring from the precondition the set of writable and readable locations. Parkinson and Bierman's [13] abstract predicates inspired our pure predicate methods. Our approach additionally supports using normal pure methods in specifications.

In [7], the authors propose using data groups to specify side-effects. To ensure the soundness, their approach imposes two methodological restrictions: the pivot uniqueness and owner exclusion restriction. Our approach requires no such restrictions, and as a consequence it can handle programs that [7] cannot. For example, the former restriction rules out sharing of representation objects, as is the case in the iterator pattern.

In the universe type system [8] and the Boogie methodology [9, 14, 16], abstractions (pure methods, model fields or invariants) can depend on the fields of owned objects and on the fields of peers (i.e. objects with the same owner as the receiver), provided the abstraction is visible to the peer. For example, the method hasNext of an iterator would have to be visible to the class ArrayList. Our approach has no such restriction.

Using method calls in specifications to achieve data abstraction is not new. In particular, our encoding of pure methods is similar in many ways to encodings proposed in other approaches [19, 20, 14, 2]. However, we are not aware of any such approach that infers the set of locations a pure method depends on from the method's precondition.

6 Conclusion

We proposed a variant of the dynamic frames approach that eliminates the need to explicitly write and check frame annotations. Instead, framing information is inferred from accessibility predicates within pre- and postconditions. We replaced the non-modular acyclicity-restriction of [5] by modular rules. Our approach is implemented in a tool, which has been used to verify several challenging programs.

In the future, we plan to extend our approach to concurrent programs.

Acknowledgments

Jan Smans is a research assistant of the Fund for Scientific Research - Flanders (FWO). Bart Jacobs is a postdoctoral fellow of the Fund for Scientific Research - Flanders (FWO).

References

1. Kassios, Y.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Formal Methods. (2006)
2. Jan Smans, Bart Jacobs, F.P., Schulte, W.: An automatic verifier for a java-like language based on dynamic frames. In: Formal Aspects of Software Engineering. (2008)
3. Schoeller, B.: Making classes provable through contracts, models and frames. PhD thesis, ETH Zurich (2007)
4. Anindya Banerjee, D.A.N., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Twenty-second European Conference on Object-oriented Programming. (2008)
5. Kassios, Y.: A Theory of Object-Oriented Refinement. PhD thesis, University of Toronto (2006)
6. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th Annual IEEE Symposium on Logic in Computer Science. (2002)
7. Leino, K.R.M., Poetzsch-Heffter, A., Zhou, Y.: Using data groups to specify and check side effects. In: Programming Language Design and Implementation. (2002)
8. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. PhD thesis, FernUniversität Hagen (2001)
9. Barnett, M., DeLine, R., Fahndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3**(6) (2004)
10. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research Redmond, USA (2005)
11. Parkinson, M.: Local Reasoning for Java. PhD thesis, University of Cambridge (2005)
12. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Conference on Tools and Algorithms for the Construction and Analysis of Systems. (2008)
13. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: 32nd Symposium on Principles of Programming Languages. (2005)
14. Jacobs, B., Piessens, F.: Inspector methods for state abstraction. Journal of Object Technology **6**(5) (2007)
15. Distefano, D., Parkinson, M.: jStar: Towards practical verification for java. In: Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2008)
16. Leino, K.L.M., Müller, P.: Object invariants in dynamic contexts. In: European Conference on Object-Oriented Programming. (2004)
17. Parkinson, M., Bierman, G.: Separation logic, abstraction and inheritance. In: 35nd Symposium on Principles of Programming Languages. (2008)
18. Parkinson, M.: Class invariants: The end of the road? In: International Workshop on Aliasing, Confinement and Ownership. (2007)

19. Darvas, A., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Formal Aspects of Software Engineering. (2007)
20. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: European Symposium on Programming. (2008)

A Translation

$$
\begin{array}{ll}
\mathcal{T}[\![\mathsf{x}]\!]_h & ::= \mathsf{x} \\
\mathcal{T}[\![\mathsf{e.f}]\!]_h & ::= h[\mathcal{T}[\![\mathsf{e}]\!]_h, \mathsf{f}] \\
\mathcal{T}[\![\mathsf{c}]\!]_h & ::= \mathsf{c} \\
\mathcal{T}[\![\mathbf{acc}(\mathsf{e.f})]\!]_h & ::= h[\mathcal{T}[\![\mathsf{e}]\!]_h, \mathsf{acc}][\mathsf{f}] \\
\mathcal{T}[\![\phi_1 \wedge \phi_2]\!]_h & ::= \mathcal{T}[\![\phi_1]\!]_h \wedge \mathcal{T}[\![\phi_2]\!]_h \\
\mathcal{T}[\![\phi_1 * \phi_2]\!]_h & ::= \mathcal{T}[\![\phi_1 \wedge \phi_2]\!]_h \wedge (\mathcal{R}[\![\phi_1]\!]_h \cap \mathcal{R}[\![\phi_2]\!]_h = \emptyset) \\
\mathcal{T}[\![\mathsf{e}_1 = \mathsf{e}_2]\!]_h & ::= \mathcal{T}[\![\mathsf{e}_1]\!]_h = \mathcal{T}[\![\mathsf{e}_2]\!]_h \\
\mathcal{T}[\![\mathsf{e.n}(\mathsf{e}_1, \ldots, \mathsf{e}_i)]\!]_h & ::= \#\mathsf{C.n}(h, \mathcal{T}[\![\mathsf{e}]\!]_h, \mathcal{T}[\![\mathsf{e}_1]\!]_h, \ldots, \mathcal{T}[\![\mathsf{e}_i]\!]_h) \\
 & \quad \text{where } \mathsf{C} \text{ is the class declaring } \mathsf{n}. \\
\mathcal{T}[\![\mathsf{e.q}(\mathsf{e}_1, \ldots, \mathsf{e}_i)]\!]_h & ::= \#\mathsf{C.q}(h, \mathcal{T}[\![\mathsf{e}]\!]_h, \mathcal{T}[\![\mathsf{e}_1]\!]_h, \ldots, \mathcal{T}[\![\mathsf{e}_i]\!]_h) \\
 & \quad \text{where } \mathsf{C} \text{ is the class declaring } \mathsf{q}.
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{D}[\![\mathsf{x}]\!]_h & ::= \mathbf{true} \\
\mathcal{D}[\![\mathsf{e.f}]\!]_h & ::= \mathcal{D}[\![\mathsf{e}]\!]_h \wedge \mathcal{T}[\![\mathsf{e}]\!]_h \neq \mathsf{null} \wedge h[\mathcal{T}[\![\mathsf{e}]\!]_h, \mathsf{acc}][\mathsf{f}] \\
\mathcal{D}[\![\mathsf{c}]\!]_h & ::= \mathbf{true} \\
\mathcal{D}[\![\mathbf{acc}(\mathsf{e.f})]\!]_h & ::= \mathcal{D}[\![\mathsf{e}]\!]_h \wedge \mathcal{T}[\![\mathsf{e}]\!]_h \neq \mathsf{null} \\
\mathcal{D}[\![\phi_1 \wedge \phi_2]\!]_h & ::= \mathcal{D}[\![\mathsf{e}_1]\!]_h \wedge (\mathcal{T}[\![\mathsf{e}_1]\!]_h \Rightarrow \mathcal{D}[\![\mathsf{e}_2]\!]_h) \\
\mathcal{D}[\![\phi_1 * \phi_2]\!]_h & ::= \mathcal{D}[\![\phi_1 \wedge \phi_2]\!]_h \\
\mathcal{D}[\![\mathsf{e}_1 = \mathsf{e}_2]\!]_h & ::= \mathcal{D}[\![\mathsf{e}_1]\!]_h \wedge \mathcal{D}[\![\mathsf{e}_2]\!]_h \\
\mathcal{D}[\![\mathsf{e.n}(\mathsf{e}_1, \ldots, \mathsf{e}_i)]\!]_h & ::= \mathcal{D}[\![\mathsf{e}]\!]_h \wedge \mathcal{D}[\![\mathsf{e}_1]\!]_h \wedge \ldots \wedge \mathcal{D}[\![\mathsf{e}_i]\!]_h \wedge \mathcal{D}[\![\mathsf{e}]\!]_h \neq \mathsf{null} \wedge \\
 & \quad \mathcal{T}[\![\phi[\mathsf{e}/\mathsf{this}, \mathsf{e}_1/\mathsf{x}_1, \ldots, \mathsf{e}_i/\mathsf{x}_i]]\!]_h \\
 & \quad \text{where } \phi \text{ is the precondition of } \mathsf{n}, \\
 & \quad \text{and } \mathsf{x}_1, \ldots, \mathsf{x}_i \text{ are } \mathsf{n}\text{'s parameters.} \\
\mathcal{D}[\![\mathsf{e.q}(\mathsf{e}_1, \ldots, \mathsf{e}_i)]\!]_h & ::= \mathcal{D}[\![\mathsf{e}]\!]_h \wedge \mathcal{D}[\![\mathsf{e}_1]\!]_h \wedge \ldots \wedge \mathcal{D}[\![\mathsf{e}_i]\!]_h \wedge \mathcal{D}[\![\mathsf{e}]\!]_h \neq \mathsf{null}
\end{array}
$$

Fig. 4. Translation and definedness of expressions.

Ownership, Pointer Arithmetic and Memory Separation

Romain Bardou*

ENS Cachan, F-94230
& Univ Paris-Sud, CNRS, Orsay F-91405
& INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893

Abstract. Ownership systems provide a way to reason about data structures in a hierarchical fashion. We propose a small but extensible language featuring an ownership system and data invariants. It is then extended with pointer arithmetic, showing how to specify array invariants. We show how to express the global properties of the ownership system in the logic. This method can be used with a memory model featuring memory separation, and provides a practical way to use invariants in deductive verification of programs. We implemented the proposed system in the Why platform and applied it to C and Java programs.

1 Introduction

One way of proving computer program properties is to annotate programs with *specifications*, and then use a *verification condition generator* (VCG) to produce proof obligations. These obligations, once proven, ensure the adequation of programs with respect to their specification.

The specification language can feature *data invariants*. For instance, a balanced binary search tree has two invariants: it is sorted, and it is balanced. A program breaking the "balanced" invariant will not be as efficient; and a program breaking the "sorted" invariant will not even be sound. Thus, the VCG must produce proof obligations ensuring that data invariants are not broken. For instance, the soundness of a search function might rely on the "sorted" invariant. Handling data invariants in a sound way is not trivial, in particular for object-oriented programs [12].

A key issue is to choose an *invariant policy* saying when an invariant is supposed to hold. A *strong* invariant policy, where invariants hold permanently, is too constraining. For instance, when inserting a new element into a binary search tree, its invariants might temporarily be broken. In the Java Modeling Language (JML) [5], invariants are supposed to hold at method boundaries for all *accessible* objects. This policy is difficult to support, both for static and dynamic verification tools.

Ownership is a particular invariant policy where objects can own other objects. By preventing owned objects to be modified, the language can allow the invariant of an object to depend on its owned objects. Barnett *et al.* [2] use ownership in the Boogie methodology, which is used to prove C# programs using the Spec# language. Dietl and Müller use ownership in JML, using the Universe type system [6]. Boulmé and Potet use ownership to interpret invariant composition in the B method [4].

* This work is partly supported by the ARC INRIA "CeProMi" (`http://www.lri.fr/cepromi/`) and by the "CAT" grant ANR-05-RNTL-00302

Ownership and invariant systems have global properties such as: "every object built by the program verifies its invariants", that are important to prove a program specification. It is tempting to provide these properties as axioms to the user, but the predicate "built by the program" is hard to express in the logic. The VCG cannot just enumerate objects in memory and add their invariants in hypotheses, as the number of these objects is unknown, and would be too big anyway.

In the Why platform [8], functions are specified using pre-conditions and post-conditions, as in Hoare logic [9]. The Why VCG, based on Dijkstra's weakest pre-condition calculus [7], computes proof obligations from these specifications. The Why platform contains an intermediate language called Jessie, whose memory model features pointer arithmetic and memory separation using the "component-as-array" model of Burstall and Bornat [3]. This model causes problems to implement an ownership system:

- because of pointer arithmetic, a field of a structure can represent several ownable objects;
- because of memory separation, the global invariants of the ownership system are harder to express.

Memory separation allows to reason about pointer modification without having to worry about pointer aliases, as memory is syntactically split into several *regions*. In particular, this simplifies modular reasoning about programs. Other works tackle this problem, such as separation logic [14] or systems where the user himself may define region variables as location sets [1]. Dynamic frames [11, 15], in particular, allow these variables to be modified during the execution. These systems allow finer memory separation than in Jessie, at the cost of verbosity.

This paper proposes a small formal language with ownership and invariants, compatible with the memory model of Jessie, *i.e.*, with pointer arithmetic and memory separation. We also show how to use invariants when proving the proof obligations of a program. This proposal generalizes the work of Barnett *et al.* in Spec# and was implemented in Jessie, allowing to test it on C and Java programs.

2 Invariants and Ownership

In this section, we give an intuitive description of our ownership system, which is mostly the same than the one used in Spec# [2].

In this paper, the term "object" does not necessarily refer to an instance of a class. In fact, an object is any reference (or pointer) to any data structure.

Objects Are Boxes If a box $\square$ is inside a box $\triangle$, then $\square$ is *owned* by $\triangle$. If $\triangle$ is itself inside another box $\diamond$, then $\square$ is also owned by $\diamond$, because $\square$ is also inside $\diamond$. This defines the *ownership* relation on objects.

The ownership relation is the transitive closure of the *direct ownership* relation. $\square$ is *directly owned* by $\triangle$ if $\square$ is owned by $\triangle$, and if for all boxes $\diamond$ owning $\square$, $\diamond = \triangle$ or $\diamond$ owns $\triangle$. In other words, there is no box between $\square$ and its direct owner.

The direct owner is unique. It doesn't have to exist, though: a box which is not inside any other box is not owned, nor directly owned.

Boxes Can Be Open or Closed To modify the content of a box, it must be *open*. And to open a box, it must be outside any other box, *i.e.*, it must not be owned. This means that to modify an owned object, one must open its owner first. Of course, this owner might be in another bigger box that we might need to open before.

Before closing a box, we must check that all the boxes it contains are already closed. In other words, an object can only own – directly or not – closed objects, and an open object cannot be owned.

Invariants on Closed Boxes The problem with an invariant is that it can be broken. Let's say that you have an integer x, and an invariant saying that $x \neq 0$. Nothing prevents the programmer from assigning 0 to x.

Except if the structure is closed. If we always check its invariant before closing x, we know that $x \neq 0$ always holds when x is closed, because x cannot have been modified since its invariant was checked. By transitivity, if a box is closed, every object it contains also verifies its invariants.

Invariants on Multiple Objects If an invariant depends on several objects, we have to check it each time one of these objects is closed, and we can only assume it if all the objects are closed. This constraint is too heavy.

In the ownership system of Spec#, invariants are associated to objects. The invariants of $\square$ can only depend on the objects that $\square$ owns. Thus, it is sufficient to check the invariant of $\square$ only when closing $\square$, because the other boxes on which the invariant depends are already closed, as they are owned by $\square$.

3 Core Language

In this section we define a small core language featuring pointers on values. These values are our boxes: they contain an invariant and may be open or closed. We show how to express and prove the soundness of the ownership system.

Due to space limitation, we omitted typing rules and the soundness proof of the ownership system. They can be found in the extended version of this article at `http://romain.bardou.fr/papers/jcownlong.pdf`.

Syntax The syntax of our core language is defined in Fig. 1. Values are expressions which cannot be reduced. They may be constants or pointers. Pointers are not directly used when programming; they are the result of allocation. They are annotated with their types. The language has basic expressions: let-binding, sequence, while loops and if-then-else tests.

A pointer p is allocated using **new** $\langle v; I; \boldsymbol{r} \rangle$. The value v can be accessed by dereferencing using $!p$ or modified using $p := e$. Pointers are the boxes of the ownership system: they have an invariant I which is given at allocation. I may depend on the contents of p and a set of pointers $\boldsymbol{r}$, which are also given at allocation. We assume given a syntax for sets of pointers, such as in `assignable` clauses of JML. These are the *reps* pointers of p (standing for "representation" pointers). Finally, pointers can be closed or

Expressions:

$e ::= v$		Values
	x	Variables
	let $x = e$ **in** e	Binding
	$e; e$	Seq
	while e **do** e	Turing-completion
	if e **then** e **else** e	Test
	new $\langle e; I; \boldsymbol{r} \rangle$	Allocation
	$!e$	Dereferencing
	$e := e$	Assignment
	pack e	Closing boxes
	unpack e	Opening boxes

Values:

$v ::= c$		Constants
	$p : \langle \tau \rangle$	Pointers

Types:

$\tau ::= \text{unit} \mid \text{bool} \mid \cdots$		Base types
	$\langle \tau \rangle$	Pointers

Environments:

$\Delta ::= \epsilon \mid \Delta, x : \tau$	Typing
$\Gamma ::= \epsilon \mid \Gamma, p = \langle e; I; \boldsymbol{r} \rangle^{\square}$	Memories
$\square ::= \circ \mid \times \mid \otimes$	Box state

Fig. 1. Core language syntax

opened using **pack** or **unpack**. As in Spec#, ownership transfer on a pointer p can be achieved to change the ownership hierarchy by unpacking the owner of p and packing another owner which has p as a rep.

We do not specify the logic used to write invariants. We only need to be able to know if an invariant holds, given the state of the memory. We assume that modifying a pointer which is not a rep of a pointer p cannot break the invariant of p.

As an example, the following expression E has one free variable: x which is an integer pointer. It returns a new pointer which has an invariant: it is greater than x. The invariant is a function which takes its future associated pointer as an argument (here p).

$$E = \textbf{let } y = \textbf{new } \langle 0; \lambda p.\ (!p > !x); x \rangle \textbf{ in } y := !x + 1; \textbf{pack } y; y$$

Note that the invariant does not necessarily hold initially, as x might be stricly greater than 0. This is allowed because the pointer is initially open.

This example also shows that in our core language, invariants may be associated with any pointer and may depend on any pointers. This is more general than the ownership system of Spec# where the reps of an object are defined in its type and are restricted to its fields.

Semantics We define a small-step semantics for our core language. $\rightarrow$ is a relation on states, and states are couples of a memory and an expression. The following:

$$\Gamma_1; e_1 \rightarrow \Gamma_2; e_2$$

should be read: "e_1 in memory Γ_1 reduces to e_2 in memory Γ_2".

The $\rightarrow$ relation is the smallest fixpoint of the rules given in Fig. 2, plus some context rules which define the evaluation order (for example, the test cannot be reduced if its condition has not been reduced already). We suppose that the substitution used when reducing let-bindings does not capture variables.

Rules for let-binding (1), sequence (2), loop (3) and test (4, 5) are easy to read. The interesting rules are the ones for allocation (6), pointer dereferencing (7) or assignment (8), packing (9) and unpacking (10), as they can read and write in memory. The

$$\Gamma; \textbf{let } x = v \textbf{ in } e \to \Gamma; e[v/x] \tag{1}$$

$$\Gamma; (\text{unit}; e) \to \Gamma; e \tag{2}$$

$$\Gamma; \textbf{while } e_1 \textbf{ do } e_2 \to \Gamma; \textbf{if } e_1 \textbf{ then } (e_2; \textbf{while } e_1 \textbf{ do } e_2) \textbf{ else } \text{unit} \tag{3}$$

$$\Gamma; \textbf{if } \text{true} \textbf{ then } e_1 \textbf{ else } e_2 \to \Gamma; e_1 \tag{4}$$

$$\Gamma; \textbf{if } \text{false} \textbf{ then } e_1 \textbf{ else } e_2 \to \Gamma; e_2 \tag{5}$$

$$\Gamma; \textbf{new } R \to \Gamma, p = R^\circ; p \text{ (where } p \text{ is fresh in } \Gamma) \tag{6}$$

$$\Gamma, p = \langle v; I; \boldsymbol{r} \rangle^\square; !p \to \Gamma, p = \langle v; I; \boldsymbol{r} \rangle^\square; v \tag{7}$$

$$\Gamma, p = \langle v_1; I; \boldsymbol{r} \rangle^\circ; p := v_2 \to \Gamma, p = \langle v_2; I; \boldsymbol{r} \rangle^\circ; \text{unit} \tag{8}$$

$$\Gamma, \begin{pmatrix} p = \langle v; I; p_1 \cdots p_n \rangle^\circ \\ p_1 = R_1^\times, \cdots, p_n = R_n^\times \end{pmatrix}; \textbf{pack } p \to \Gamma, \begin{pmatrix} p = \langle v; I; p_1 \cdots p_n \rangle^\times \\ p_1 = R_1^\otimes, \cdots, p_n = R_n^\otimes \end{pmatrix}; \text{unit} \quad (\text{If } I(\Gamma) \text{ holds}) \tag{9}$$

$$\Gamma, \begin{pmatrix} p = \langle v; I; p_1 \cdots p_n \rangle^\times \\ p_1 = R_1^\otimes, \cdots, p_n = R_n^\otimes \end{pmatrix}; \textbf{unpack } p \to \Gamma, \begin{pmatrix} p = \langle v; I; p_1 \cdots p_n \rangle^\circ \\ p_1 = R_1^\times, \cdots, p_n = R_n^\times \end{pmatrix}; \text{unit} \tag{10}$$

We omitted rules concerning evaluation order.

Fig. 2. Core language semantics

memory is a set of allocated pointers with their value, their invariant and their reps (the pointers their invariant may depend on). Syntax for memories Γ is given in Fig. 1; they are maps from pointers to their values, invariant and reps. Each pointer can be open ($\circ$), closed ($\times$), or owned ($\otimes$); this is also stored in memory Γ.

To allocate a new pointer, one first needs a fresh pointer p. This pointer is entered in memory with its state R which contains its value, its invariant and the pointers it depends on. Initially, the pointer is open.

Pointer access and modification are just reading or modifying the value associated to the pointer in memory. Access can be done whatever the state of the pointer is, but assignment can only be done on open pointers.

Closing a pointer p with **pack** does not only change the state of p from $\circ$ (open) to $\times$ (closed); it also changes the state of its reps from $\times$ to $\otimes$ (owned). This prevents them from being opened and modified. Opening a pointer with **unpack** is similar, except that the pointer state goes from $\times$ to $\circ$ and the state of its reps goes from $\otimes$ to $\times$. Packing can only be done if the invariant holds.

Our example expression E reduces as follows, in a memory where pointer x has value 42 (omitting some trivial reductions):

$$\begin{array}{rl}
x = \langle 42; \text{true}; \emptyset \rangle^\times & \textbf{let } y = \textbf{new } \langle 0; \lambda p.\ !p >!x; x \rangle \textbf{ in } \cdots \\
x = \langle 42; \text{true}; \emptyset \rangle^\times, p = \langle 0; !p >!x; x \rangle^\circ & \textbf{let } y = p \textbf{ in } y :=!x + 1; \textbf{pack } y; y \\
x = \langle 42; \text{true}; \emptyset \rangle^\times, p = \langle 0; !p >!x; x \rangle^\circ & p :=!x + 1; \textbf{pack } p; p \\
x = \langle 42; \text{true}; \emptyset \rangle^\times, p = \langle 43; !p >!x; x \rangle^\circ & \textbf{pack } p; p \\
x = \langle 42; \text{true}; \emptyset \rangle^\otimes, p = \langle 43; !p >!x; x \rangle^\times & p
\end{array}$$

Safety A memory is consistent if all closed pointers verify their invariants, all reps pointers of all closed pointers are owned, and all owned pointers have a unique owner.

Definition 1 (Memory Consistency and Pointer State). *A memory Γ is* valid*, written $Valid(\Gamma)$, when:*

- *for all $\langle v; I; p_1 \cdots p_n \rangle^{\square}$ in Γ where $\square \in \{\times, \otimes\}$, I holds and for all i, the state of p_i in Γ is $\otimes$;*
- *for all $p = R^{\otimes}$ in Γ, there is a unique $\langle v; I; \boldsymbol{r} \rangle^{\square}$ in Γ such that $\square \in \{\times, \otimes\}$ and $p \in \boldsymbol{r}$.*

The safety of the ownership system is given by the following property: the memory stays consistent when executing the program.

Theorem 1 (Ownership Safety). *If $Valid(\Gamma_1)$ and $\Gamma_1; e_1 \rightarrow^{\star} \Gamma_2; e_2$ then $Valid(\Gamma_2)$*

The proof is basically the same as the one of Barnett *et al.* [2] but adapted to our simpler core language.

4 Pointer Arithmetic

This section shows how to extend our core language with pointer arithmetic. This can also be used to model arrays.

Pointer Shifting and Difference We assume an operation $\oplus$, called *shift*, which takes a pointer and an integer *offset* and returns a pointer. This operation should verify the following properties:

$$p \oplus i = p \iff i = 0$$
$$(p \oplus i) \oplus j = p \oplus (i + j)$$

We also assume an operation $\ominus$ which takes two pointers and return an integer offset. This operation should verify the following property:

$$p' \ominus p = i \iff p' = p \oplus i$$

Note that the axiomatisation of $\oplus$ does not necessarily mean that all pointers are related through an offset shift. This means that $\ominus$ does not have to be defined for all pairs of pointers. For instance, $\ominus$ allows to model pointer difference in C programs. The ANSI semantics of C specifies that the result is undetermined if the two pointers are not in the same block.[1] This property will be useful when defining arrays (see Sect. 4).

The syntax of expressions is extended to handle the $\oplus$ and $\ominus$ operations. The semantics of the syntax constructions $\oplus$ and $\ominus$ is simply to reduce into the value returned by the respective operations, without changing the memory.

[1] In the Why platform, pointer difference generates a proof obligation requesting that the two pointers are in the same block.

Allocation Pointer shifting can be used to build new pointers, but nothing ensures that these pointers have been allocated. If they are not in memory, the access reduction rule cannot be applied.

We extend allocation by adding an expression n of type int in brackets. This integer is the size of the allocated block. The invariant is now parameterized by the offset of the pointer. The semantics of **new** is defined by the following reduction rule:

$$\Gamma; \textbf{new}\ R[n] \rightarrow \Gamma, p = R(0), p \oplus 1 = R(1), \cdots, p \oplus n - 1 = R(n-1); p$$

where R denotes $\lambda o.\ \langle v; I(o); \boldsymbol{r}(o)\rangle$ and p is a fresh pointer such that $p\oplus 1 \cdots p\oplus(n-1)$ are also fresh in Γ.

This extended **new** can be used to allocate several pointers at the same time. These pointers are all accessible by shifting the pointer returned by the allocation, and their invariants can be different.

Arrays The pointer arithmetic extension can be used to build arrays. For example, the following expression allocates a new array of positive integers:

$$\textbf{new}\ \lambda o.\ \langle 0; \lambda p.\ !p \geq 0; \emptyset\rangle[n]$$

However, the invariant of this array is split into each cell, which is handled separately. A better solution would be to have one single invariant for the whole array:

$$\begin{array}{l}\textbf{let}\ p = \textbf{new}\ \langle 0; \text{true}; \emptyset\rangle[n]\ \textbf{in}\\ \textbf{new}\ \langle p; (\lambda p'.\ !p' \geq 0); p \oplus [0..(n-1)]\rangle\end{array}$$

Note that the size of the rep pointer set depends on n. Another solution is to use the set constructor $p[\star]$ meaning: "all valid shifts of p".

This supposes that the cells of an array are exactly the valid shifts of its first cell. This is possible in our extension thanks to the unconstrained operator $\oplus$ which doesn't have to link all pointers together; otherwise there could be only one array in the whole program.

For pratical purposes, it is also handy to add packing and unpacking operations on blocks $p[\star]$ of pointers. Without them, the user has to write a loop everytime an array is opened or closed.

5 Using Invariants in Proofs

We assume that we can statically determine the invariant of a pointer. This strong restriction can be obtained by typing: the user defines a finite set of invariants and the invariant of a pointer is added to its type. In Spec# or in Jessie, all objects of the same class or structure have the same invariant.

Problem Let's assume that example E in section Sect. 3 has $!p > !x$ as a post-condition P. The generated proof obligation looks like (universal quantifications are omitted):

$$\begin{array}{rcl} \Gamma_1^x = (x = \langle 42; \text{true}; \emptyset \rangle) & \Rightarrow & S_1^x = (x = \times) \Rightarrow \\ \Gamma_1^p = (p = \langle 0; !p > !x; x \rangle) & \Rightarrow & S_1^p = (p = \circ) \Rightarrow \\ \Gamma_2^p = store(\Gamma_1^p, p, 43) & \Rightarrow & S_2^x = store(S_1^x, x, \otimes) \Rightarrow \\ S_2^p = store(S_1^p, p, \times) & \Rightarrow & select(\Gamma_2^p, p) > select(\Gamma_1^x, x) \end{array}$$

where $store$ and $select$ are logic functions to, respectively, change and read the value of a pointer in a memory map. For example, $!p$ becomes $select(\Gamma, p)$ in the logic, where Γ is the current memory. Note how memory separation such as the "component-as-array" model [3] or even finer separation using regions [10] allow to split Γ into several maps. Here, we separated pointers x and p. We also separated values and states of pointers using two maps: Γ and S respectively.

One way of proving P is to read the values of x and p using the hypotheses about Γ_1^x and Γ_2^p. In this example, this is trivial; but usually it is not so simple. A much easier solution would be to use the last hypothesis (p is closed) to apply the invariant of p. To do so we need to apply $Valid$. But one cannot just add an axiom such as:

$$\forall \Gamma,\ Valid(\Gamma) \tag{11}$$

This axiom is inconsistent; the quantification on Γ should be restricted to memories that are actually produced by the program. In Spec# this is done using a predicate called $IsHeap$. The VCG adds instances of this predicate in the proof-obligation hypotheses and (11) becomes:

$$\forall \Gamma,\ IsHeap(\Gamma) \Rightarrow Valid(\Gamma) \tag{12}$$

With memory separation, we cannot instanciate $IsHeap$ on all memory parts, otherwise one could prove inconsistent instances of $Valid$ such as $Valid(\Gamma_1^x, \Gamma_1^p, S_2^x, S_2^p)$ which implies $0 > 42$.

Solution In Jessie, we choose to bypass the use of $IsHeap$: the $Valid$ predicate is instantiated everytime the user might need it, and this instantiation is added as an assumption in the hypotheses of the obligations the user has to prove. We add the following hypotheses to the proof obligation for example E:

$$\begin{array}{ll} Valid(\Gamma_1^x, S_1^x) & Valid(\Gamma_1^x, S_1^x, \Gamma_1^p, S_1^p) \\ Valid(\Gamma_1^x, S_1^x, \Gamma_2^p, S_1^p) & Valid(\Gamma_1^x, S_2^x, \Gamma_2^p, S_2^p) \end{array}$$

In theory, the user might need the $Valid$ predicate to be instantiated everytime the memory is modified, but this would pollute proof obligations with too many hypotheses. In practice, we only instantiate $Valid$ at the beginning of function bodies and loops, and when the memory is modified. It is instanciated on the needed memory parts only.

Another possibility is to instantiate $Valid$ only at the beginning of functions; the user can then deduce the other instances of $Valid$. However, proving $Valid$ can sometimes be quite difficult. Another drawback would be that we would lose some separation properties. Thanks to memory separation, memory can be split into (Γ, S) where

Γ contains pointer values, and S contains pointer states ($\circ$, $\times$ or $\otimes$). For example:

$$\textbf{let } x = \textbf{new } \langle 1; \lambda p.\ !p > 0; \emptyset \rangle \textbf{ in } e$$

A theorem prover can easily deduce that $!x > 0$ if it knows that e returns x packed, but only if *Valid* has been added as an assumption after e.

Well-foundedness We are using *Valid* to prove some proof obligations, but *Valid* only holds if the proof obligations have been proven. In this section, we show that this is well-founded.

The proof obligations ensure, among other things, that an expression e_1 which is not a value reduces without errors:

$$\text{If } \mathit{Valid}(\Gamma_1) \text{ then there exist } \Gamma_2, e_2 \text{ such that } \Gamma_1; e_1 \rightarrow \Gamma_2; e_2 \tag{13}$$

In particular, this means that invariants hold before packing, assigned pointers are open, and so on.

We apply Theorem 1 (which does not depend on proof obligations) to extend (13):

$$\text{If } \mathit{Valid}(\Gamma_1) \text{ then there exist } \Gamma_2, e_2 \text{ such that } \Gamma_1; e_1 \rightarrow \Gamma_2; e_2 \text{ and } \mathit{Valid}(\Gamma_2) \tag{14}$$

By applying (14) inductively, we show our final theorem:

Theorem 2. *All instances of Valid introduced as assumptions are correct.*

6 Example

This example is inspired by some Java code due to Müller [13]. We suppose an integer array t of size $n+1$. The following Java code counts the number of positive integers in the array, and then copies them into a new array u:

```
int i, j, count = 0;
for (i=0; i < t.length; i++)
  if (t[i] > 0) count++;
int u[] = new int[count];
for (i=0, j=0; i < t.length; i++)
  if (t[i] > 0) u[j++] = t[i];
```

We can encode this in our core language:

let $i = \textbf{new } \langle 0; \text{true}; \emptyset \rangle$ **in**
let $j = \textbf{new } \langle 0; \text{true}; \emptyset \rangle$ **in**
let $\mathit{count} = \textbf{new } \langle 0; (\lambda p.\ !p = \mathit{Card}\{k \mid 0 \le k \le n \ \wedge\ !(t \oplus k) > 0\}); t \oplus [0..n] \rangle$ **in**
while $!i \le n$ **do**
  (**if** $!(t \oplus !i) > 0$ **then** $\mathit{count} := !\mathit{count} + 1$;
  $i := !i + 1$);
pack count;
let $u = \textbf{new } \langle 0; \text{true}; \emptyset \rangle[!\mathit{count}]$ **in**
$i := 0$;
while $!i \le n$ **do**
  (**if** $!(t \oplus !i) > 0$ **then** $(u \oplus !j := !(t \oplus !i);\ \ j := !j + 1)$;
  $i := !i + 1$)

Pointer $count$ has an invariant saying that its content is the number of strictly positive integers in t. We prove it when packing $count$ using a loop invariant on the first loop.

This illustrates several features of our core language: invariants on any pointer and not just objects, pointer arithmetic, and memory separation support. Memory separation is key to prove the safety of the accesses to u in the second loop. Indeed, $j < count$ is a loop invariant; but to show it we need to know that t is not modified by updates to u. Memory separation gives this for free by separating t and u [10].

To show that $j < count$ we also need the invariant of $count$, given by instanciating the following predicate:

$$\begin{array}{l} \mathit{Valid}(\Gamma^t, \Gamma^{count}, S^t, S^{count}) \equiv \\ \quad \forall count,\ \mathit{select}(S^{count}, count) \in \{\times, \otimes\}\ \Rightarrow \\ \quad \mathit{select}(\Gamma^{count}, count) = \mathit{Card}\{k \mid 0 \leq k \leq n\ \wedge\ \mathit{select}(\Gamma^t, (t \oplus k)) > 0\} \end{array}$$

This predicate is instanciated using the memory parts corresponding to the last time $count$ was modified, *i.e.*, when it was packed. We only show the part of the predicate needed to deduce the invariant of $count$.

7 Conclusion

We introduced a small language with pointers, an ownership system and invariants. It was shown to be safe, and then extended with pointer arithmetic. As far as we know, this work is the first attempt to formalize an ownership system with pointer arithmetic, and thus applicable to the C language, although the VCC tool [16] implements a solution. This extension offers multiple ways to specify array invariants: they can be split in each cell, or a pointer on the array can be allocated with a global invariant for the array.

Our core language could be extended with several other features. In particular, pointers could contain extensible records, which would model objects. As for pointer arithmetic, this is mostly independent from the ownership system itself. Our language is an attempt to generalize the Spec# methodology: invariants are not limited to object fields and may depend on any pointer. This simplifies formal reasoning, and can be used in languages without objects, or with invariants which are not necessarily defined for a whole class.

We also showed how the safety properties of the ownership system can be instantiated to be used when proving the proof obligations of the program. This was already done with simple memory models, although to the best of our knowledge, formalizing this method and proving its soundness is new. Moreover, our solution can be used when the memory model features memory separation, as in the Jessie language in which is was implemented.

Another lead for research could be the separation properties of the ownership system shown in Sect 5. Invariant properties on the whole memory can be deduced from a small part of the memory.

Acknowledgements Thanks to Claude Marché, Yannick Moy, to all the members of the ProVal team, and to all anonymous reviewers for their invaluable remarks and advice.

References

1. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *22nd European Conference on Object-Oriented Programming (ECOOP'08)*, Paphos, Cyprus, July 2008.
2. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.
3. R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.
4. S. Boulmé and M.-L. Potet. Interpreting invariant composition in the B method using the Spec# ownership relation: a way to explain and relax B restrictions. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2007.
5. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2004.
6. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
7. E. W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.
8. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Berlin, Germany, July 2007. Springer.
9. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580 and 583, Oct. 1969.
10. T. Hubert and C. Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, Braga, Portugal, Mar. 2007. `http://www.lri.fr/~marche/hubert07hav.pdf`.
11. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283, Hamilton, Canada, 2006.
12. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007.
13. P. Müller. Specification and verification challenges. Exploratory Workshop: Challenges in Java Program Verification, Nijmegen, The Netherlands, Sept. 2006. `http://www.cs.ru.nl/~woj/esfws06/slides/Peter.pdf`.
14. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.
15. J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for java-like programs based on dynamic frames. In *Fundamental Approaches to Software Engineering (FASE'08)*, Budapest, Hungary, Apr. 2008.
16. J. S. Wolfram Schulte, Songtao Xia and F. Piessens. A glimpse of a verifying c compiler.

Lock Inference Proven Correct

Dave Cunningham[1], Sophia Drossopoulou[1], and Susan Eisenbach[1]

Imperial College London {dc04,sd,sue}@doc.ic.ac.uk

Abstract. With the introduction of multi-core CPUs, multi-threaded programming is becoming significantly more popular. Unfortunately, it is difficult for programmers to ensure their code is correct because current languages are too low-level.
Atomic sections are a recent language primitive that expose a higher level interface to programmers. Thus they make concurrent programming more straightforward. Atomic sections can be compiled using transactional memory or lock inference, but ensuring correctness and good performance is a challenge. Transactional memory has problems with IO and contention, whereas lock inference algorithms are often too imprecise which translates to a loss of parallelism at runtime.
We define a lock inference algorithm that has good precision. We give the operational semantics of a model OO language, and define a notion of correctness for our algorithm. We then prove correctness using Isabelle/HOL.

1 Introduction

Programmers increasingly need to write multi-threaded programs to make full use of the available hardware. When writing multi-threaded code, programmers typically use the same data-structures and algorithms as sequential code. However, many simple *sequential assumptions* that allow local reasoning about program behaviour no longer hold in a concurrent setting. For instance, one can no-longer assume that the value stored in a variable is the value last written by the local thread. Writing code that is robust enough to remain correct, despite these weak semantics, is a mammoth task compared to writing sequential code where the assumptions always hold.

In order to stay productive, programmers often try to make operations *atomic* [15], using locks. If a block of code is atomic, a programmer can once again make sequential assumptions and reason locally about the behaviour of his code. Unfortunately, locks are extremely unforgiving. Small errors can cause the silent loss of atomicity (an *atomicity violation*). Even if the locking code is sufficient for atomicity, there are extra constraints that must be met to avoid *deadlock*. Finally, even if programmers understand these details, they can have great difficulty when writing large programs that have many complicated thread interactions.

Programmers create large programs using encapsulation. This allows them to concentrate on small parts of the program without worrying about the rest. However, in a concurrent scenario, it is not possible to lock successfully without having intimate knowledge of the behaviour of any called functions. If the

internal behaviour of a function changes, it may be necessary to update locking code in distant parts of the program, to reflect these changes. Thus programmers lose encapsulation and consequently have to reason about much more than just the local code and state. The loss of encapsulation results in the perception of concurrent programming as prohibitively difficult.

The difficulty of using locks has led to the introduction of a new language primitive - the *atomic section*. Using this primitive, programmers need only designate an arbitrary block as an atomic section and the implementation does any global reasoning and instrumentation required so that sequential assumptions hold. Thus, programmers can make these assumptions, without having to write complex locking code, and without losing the benefits of encapsulation.

The problem moves from the application programmer to the language implementation, which must output code without atomicity violations, and without sacrificing parallel performance. Whatever mechanisms are used by the implementation, they must be used carefully. Any emergent behaviours such as deadlocks must be prevented or otherwise not exposed to the programmer, who should be able to use atomic sections free from implementation-specific constraints.

Current implementations of atomic sections use either transactional memory [1, 17] or lock inference [11, 9, 5, 3]. Transactional memory relies on being able to rollback blocks of code whose atomicity has been violated, but in general this means it cannot allow I/O or system calls in atomic sections. This restriction cannot in general be hidden from the programmer. Lock inference does not have this problem, but relies on precise static approximation of program behaviour. The more precise the inference, the more threads are allowed to execute in parallel. Programmers should not have to design their code so that the lock inference can easily understand it.

Our lock inference algorithm [5] is designed to give good precision. Whereas previous work relies on pointer analysis to statically model program behaviours, we use a more direct approach that has more in common with how programmers infer locks manually. However since our approach does not make such extensive use of proven technology, its correctness is brought into question. The contribution of this paper is a notion of correctness for our analysis, and an account of our experience proving it in the Isabelle proof assistant. In Section 2 we give examples showing how our analysis works. In Section 3 we formalise a model Object-Oriented language, our lock inference algorithm, the notion of correctness that binds them together, and describe our experience with Isabelle. In Section 4 we compare to related work and we conclude with Section 5.

2 Approach

We use a two phase locking protocol [8]. We derive a set of locks that we acquire before the atomic section and release afterwards. Our inference therefore takes an atomic section as input, which we call a *program*. We henceforth assume that atomic sections have already been converted into Control Flow Graphs (CFGs) and are therefore ready for program analysis. For simplicity, we assume

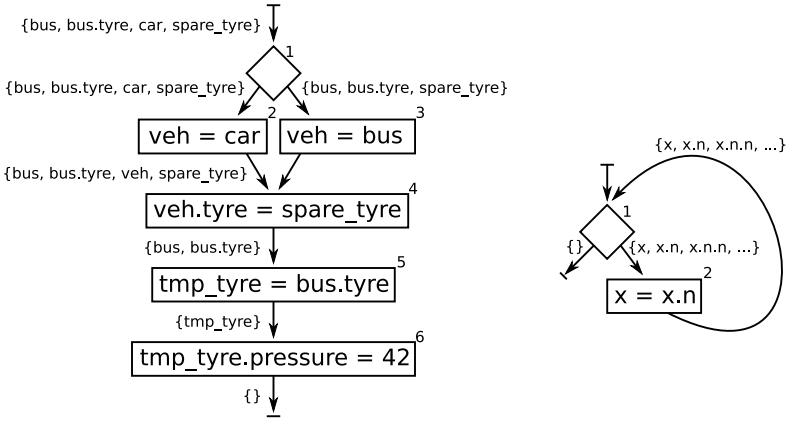


Fig. 1. Two example atomic sections

that atomic blocks are never nested[1]. We use a runtime mechanism that detects when a thread's lock acquisition would cause a deadlock [14] and rolls back all the lock acquisitions of that thread. Since lock acquisitions are together at the beginning of the atomic section, no transaction log is required to facilitate the roll back. We also assume everything accessed from an atomic section is shared between threads, and everything shared between threads is only accessed from atomic sections. We will discuss sharing more in section 5.

Consider the atomic section in Fig. 1. We use a backwards 'may' analysis to infer a set of locks at each edge. We assume every object has a lock that protects it, as in Java. Starting at node 6, we first infer a lock to protect the access of the object `tmp_tyre`. This propagates towards the beginning of the atomic section, where lock acquisition code is inserted. However, we have to translate locks as they are propogated to account for the statements they pass through. E.g. at node 5, `bus.tyre` is assigned to `tmp_tyre`, so acquiring the lock on `tmp_tyre` before node 5 does not help us ensure atomicity since it is not the object accessed at node 6. However, the correct object is held in `bus.tyre` so we can lock that instead. Also at node 5, a lock is inferred to protect the access of `bus`.

At node 4, we infer a lock to protect the `veh` access, but we also need to lock `spare_tyre`. This is because `veh` and `bus` may be aliases, and thus it may have been `spare_tyre` that was pressurised. In general we do not know, so we approximate. In this case we include both the `bus.tyre` lock (not aliased case) and the `spare_tyre` lock (aliased case).

In both branches of the conditional, there is a copy statement. At node 3, we translate `veh` to `bus` but since the set already contains `bus`, we effectively lose `veh`. Node 2 is similar. Since we do not know which branch will be chosen at node 1, we take the union of the two branches to form the final result. It is this set of locks that we hold for the duration of the atomic section.

[1] At runtime, one can set a flag that disables the inferred locks of inner atomic sections.

The right hand side of Fig. 1 is an atomic section that iterates over objects. Here, the algorithm as described above would not terminate. To force the analysis to terminate, we use a nondeterministic finite automaton (NFA) [12] at each edge, instead of the set of locks. When they contain a cycle, NFAs can finitely represent the unbounded set of objects accessed by such iterations. We call such NFAs *path graphs*.

There are many subtle mechanisms at work in our approach – e.g. the translation of locks across assignments, the handling of aliasing, and the use of NFAs to force termination. One could be forgiven for not immediately having confidence in the correctness of our algorithm. Fortunately, we can prove its correctness, as we will shortly demonstrate.

3 The Formal System

We will now give a syntax and semantics for a small Java-like language, the program analysis transition functions over this language, and then prove that the given transition functions infer correct locking information. **Notation:** $A \rightharpoonup B$ is the type of a partial map, $[a \mapsto b, c \mapsto d]$ is a partial map that maps a to b and c to d. We use _ to indicate an anonymous variable. We denote the empty sequence with ε and use . to prepend values onto sequences. Sometimes, for readability, we use commas instead of logical conjunctions ($\wedge$).

3.1 Syntax and Semantics

We analyse atomic sections independently, which we refer to as *Programs*. We assume programs have already been converted to a control flow graph (CFG) representation, where function calls are handled using bounded callstrings to approximate recursion at a fixed depth [18].

We let x, y, z range over local stack variables, f, g range over fields. Every CFG node has a unique id n chosen from some countable set $Node$. Thus our program P is defined in Fig. 2. In order, the statements are *copy assignment*, *object construction*, *heap load*, *heap store*, and *condition*. Every statement has a given successor n which is where execution proceeds after that statement, except the condition $\langle n; n' \rangle$ which non-deterministically chooses to continue execution from either n or n'. If a node has the successor n where $P(n)$ is undefined then the atomic section terminates. The right-hand program in Fig. 1 is therefore $P = [1 \mapsto \langle 3; 2 \rangle, 2 \mapsto [x = x.n; 1]]$, note that $P(3)$ is undefined.

We now give a model of the accesses incurred by an execution of a program P (we are not interested in the resulting heap or stack). This model is abstract, but not static. We have a judgement $P \vdash h, \sigma, n \rightsquigarrow A$. The intuition is that the sequence of actions A are performed by an execution of P, from the initial heap and stack h, σ and from the initial CFG node n. To represent non-terminating executions, we allow the execution to cease at any point. The sequence A may thus be shorter than a completed execution. However, our correctness theorem generalises over A, so it covers complete as well as incomplete executions. Note

$$P \in Program = Node \rightharpoonup Statement$$
$$st \in Statement ::= [x = y; n] \mid [x = \texttt{new}; n] \mid [x = y.f; n] \mid [x.f = y; n] \mid \langle n; n' \rangle$$

$$a \in Addr = \mathbb{N} \qquad f, g \in Field$$
$$v \in Value ::= a \mid \texttt{null} \qquad \sigma \in Stack = Var \rightharpoonup Value$$
$$h \in Heap = Addr \rightharpoonup Object \qquad x, y, z \in Var$$
$$Object = Field \to Value \qquad \alpha \in Action = a \mid \tau$$

$$\frac{}{P \vdash h, \sigma, n \rightsquigarrow \varepsilon} \text{(Stop)}$$

$$\frac{P(n) = [x = y; n'] \qquad P \vdash h, \sigma[x \mapsto \sigma(y)], n' \rightsquigarrow A}{P \vdash h, \sigma, n \rightsquigarrow \tau.A} \text{(Copy)}$$

$$\frac{P(n) = \langle n'; n'' \rangle \qquad P \vdash h, \sigma, n' \rightsquigarrow A \quad \vee \quad P \vdash h, \sigma, n'' \rightsquigarrow A}{P \vdash h, \sigma, n \rightsquigarrow \tau.A} \text{(Cond)}$$

$$\frac{P(n) = [x.f = y; n'] \qquad a = \sigma(x) \qquad P \vdash h[(a, f) \mapsto \sigma(y)], \sigma, n' \rightsquigarrow A}{P \vdash h, \sigma, n \rightsquigarrow a.A} \text{(Store)}$$

$$\frac{P(n) = [x = \texttt{new}; n'] \qquad a \notin dom(h) \qquad P \vdash h[a \mapsto \lambda f.\texttt{null}], \sigma[x \mapsto a], n' \rightsquigarrow A}{P \vdash h, \sigma, n \rightsquigarrow \tau.(A[a \mapsto \tau])} \text{(New)}$$

$$\frac{P(n) = [x = y.f; n'] \qquad a = \sigma(y) \qquad P \vdash h, \sigma[x \mapsto h(a, f)], n' \rightsquigarrow A}{P \vdash h, \sigma, n \rightsquigarrow a.A} \text{(Load)}$$

Fig. 2. Syntax and Semantics of Execution Model

that while the language does not allow assignment of `null`, the runtime uses `null` as a default field value, and allows `null` to be stored on the stack. Assignment of `null` can thus be encoded by reading an uninitialised field.

We can consider the execution of the above example P in the heap $h = [1 \mapsto (n \mapsto 2), 2 \mapsto (n \mapsto 3), 3 \mapsto (n \mapsto 3)]$ and the stack $\sigma = [x \mapsto 1]$. The heap is undefined at addresses other than $1, 2, 3$, and by abuse of notation, fields other than n are `null`. The execution would normally not terminate because the "list" contains a cycle. However, the judgement $P \vdash h, \sigma, 1 \rightsquigarrow 1.2.3.\varepsilon$ holds regardless. It is also true that $P \vdash h, \sigma, 1 \rightsquigarrow 1.2.\varepsilon$ and in fact $\forall P, h, \sigma, n : P \vdash h, \sigma, n \rightsquigarrow \varepsilon$.

Note that we do not record accesses of objects that are constructed by P, due to the substitution in (New). This is because the locks that we infer will ensure that the new object remains thread-local until the end of the atomic section, so we do not have to infer a lock for constructed objects.

3.2 Analysis Transition Functions

To infer locks that make P execute atomically, we use a backwards 'may' analysis to infer a static approximation of the set of objects, the *path graph*, accessed by P. This is in contrast to the complete set of possible A such that $P \vdash h, \sigma, n \rightsquigarrow A$, which cannot be known statically.

Our representation of P is a control flow graph (CFG). At each CFG edge we accumulate a path graph, which is a special kind of nondeterministic finite automaton where every state is an exit state. A path graph is a finite representation of a potentially infinite set of locks, e.g. for the iteration example in

Fig. 1, we do not infer the infinite set of locks $\{x, x.n, x.n.n, \ldots\}$, rather the finite path graph $\{x \rightarrow 2, 2 \rightarrow^n 2\}$. First we will give a formal definition of path graphs, then we give the formal transition functions that show how path graphs are pushed around the CFG as the analysis reaches its fixed point. The definitions are in Fig. 3. We lock a path graph using multi-granularity locks. We first remove nodes involved in cycles, locking all objects that have the same type as those nodes. We then take the set of paths through the path graph and lock them in prefix order, ignoring any objects whose types are already locked.

$$
\begin{aligned}
Edge &::= x \rightarrow n \mid n \rightarrow^f n' \\
G \in PathGraph &= \mathbb{P}(Edge) \\
X \in AnalysisState &= Node \rightarrow PathGraph
\end{aligned}
$$

$$
\begin{aligned}
acc &: Node \rightarrow Statement \rightarrow PathGraph \\
tr &: Node \rightarrow Statement \rightarrow PathGraph \rightarrow PathGraph
\end{aligned}
$$

$$
\begin{aligned}
acc(n)[x = y; \_] &= \emptyset & acc(n)[x = y.f; \_] &= \{y \rightarrow n\} \\
acc(n)[x = \texttt{new}; \_] &= \emptyset & acc(n)[x.f = y; \_] &= \{x \rightarrow n\}
\end{aligned}
$$

$$
\begin{aligned}
tr(n)[x = y; \_](G) &= G \setminus \{x \rightarrow n' | x \rightarrow n' \in G\} \cup \{y \rightarrow n' | x \rightarrow n' \in G\} \\
tr(n)[x = \texttt{new}; \_](G) &= G \setminus \{x \rightarrow n' | x \rightarrow n' \in G\} \\
tr(n)[x = y.f; \_](G) &= G \setminus \{x \rightarrow n' | x \rightarrow n' \in G\} \cup \{n \rightarrow^f n' | x \rightarrow n' \in G\} \\
tr(n)[x.f = y; \_](G) &= G \setminus \{n' \rightarrow^f \_ | x \rightarrow n' \in G, \\
&\qquad (\nexists z \neq x : z \rightarrow n' \in G), \\
&\qquad (\nexists n''' : n''' \rightarrow\!\!- n' \in G)\} \\
&\quad \cup \{y \rightarrow n' | \_ \rightarrow^f n' \in G\}
\end{aligned}
$$

Fig. 3. The analysis

The state of the analysis, X, stores a path graph at each CFG node, which represents the path graph at the edges pointing into that node. For conditional nodes $P(n) = \langle n'; n'' \rangle$, we simply have $X(n) = X(n') \cup X(n'')$, as is standard with backwards 'may' analyses. For all other nodes n, where $P(n) = [st; n']$, we calculate $X(n)$ as follows: $X(n) = acc(n)(st) \cup tr(n)(st)(X(n'))$. The access function acc provides the locks required to protect accesses performed by the local node n. The translation function tr translates path graphs from below n so that their meaning is preserved in spite of the changes to the heap and stack caused by n.

The access function adds locks to protect load and store statements, and otherwise adds nothing. The translation functions we will explain one at a time. Copy statements are handled simply by replacing $x \rightarrow n'$ with $y \rightarrow n'$ (for any n'). Construction is similar except it only removes edges. Accesses are 'lost' when they propagate through construction because the analysis realises that the object accessed is actually thread-local and therefore does not need to be locked. Loads are similar to copies, except that the $x \rightarrow n'$ edge gets replaced by

a $n \rightarrow^f n'$ edge. This only makes sense if we can guarantee that an edge $y \rightarrow n$ exists in the new path graph. This is easily shown, however, since the access function adds precisely this edge. The case for store is (as one would expect) the most complicated. First, we can see that it adds an edge from y to any node in the path graph that might have been affected by the assignment to the f field. This is because we conservatively assume everything can be an alias of everything else. However, we know syntactically that x is an alias of x, so we can remove any $x.f$ accesses from the path graph. At node 4 of Fig. 1, we have `veh.tyre = spare_tyre`, and below we have $X(5) = \{bus \rightarrow 5, 5 \rightarrow^{tyre} 6\}$. We therefore add the edge $\{veh \rightarrow 4\}$ due to the access function *acc*. We also add $\{spare\_tyre \rightarrow 6\}$ due to the last part of the translation function tr. There is no $veh \rightarrow 5$ in $X(5)$, but even if there was, we would still not subtract $5 \rightarrow^{tyre} 6$ from $X(5)$ because $bus \rightarrow 5$ is present.

When the analysis reaches a fixed point, we know that the path graph $X(n)$ at every node n satisfies the constraints in Fig. 3. We denote this with $P \vdash X$.

3.3 Soundness

We want our inferred path graph at the initial edge $X(n)$ to represent at least the addresses accessed by the program as it executes. For this we need a *concretisation* function γ that interprets $X(n)$ in a given stack and heap to reveal which addresses it statically represents. We overload this function to also extract the addresses from a sequence of actions A (i.e. ignoring duplicate addresses and τ actions). We can state the theorem we want to prove:

Theorem 1. *Soundness:*

$$\left.\begin{array}{l} P \vdash h, \sigma, n \rightsquigarrow A \\ P \vdash X \end{array}\right\} \Longrightarrow \gamma(A) \subseteq \gamma(h, \sigma, X(n))$$

Proof: Induction over length of A.

This intuitively says that whatever may be accessed by an execution beginning from n, these accesses will be represented by the path graph at that node in the fixed point of the analysis. It remains to see how to define γ in the case of path graphs.

3.4 Assigning meaning to path graphs

We now consider an arbitrary path graph G and an *assignment* φ which maps each node in this path graph to a set of addresses. We will define γ by flattening an appropriate φ, i.e. just keeping the set of addresses mapped by φ and forgetting at which node they occur.

Definition 1. *Valid assignments:*

$$\begin{array}{l} h, \sigma \vdash G : \varphi \Longleftrightarrow (\forall x \mapsto n \in G : \sigma(x) \in \varphi(n)) \wedge \\ \qquad\qquad\qquad (\forall n \rightarrow^f n' \in G : \{h(a, f) | a \in \varphi(n)\} \subseteq \varphi(n')) \end{array}$$

The intuition is that if the path graph contains the edge $x \rightarrow n$ then we want $a_x = \sigma(x)$ to be present in φ at n. However, if the stack is undefined at that variable, or if it is `null` then we ignore it. If the stack contains a valid address a_x for x, and G also contains $n \rightarrow^f n'$ then we want $h(a_x, f)$ to be present at n', unless that address is not defined on the heap[2] or the field contains `null`. We want addresses to flow around the path graph, initially with stack lookups, and then using the heap to follow field edges and find more addresses. Even if the path graph contains a cycle, such as with our linked list example, then the set of addresses involved can remain finite since the heap is finite. This is a purely theoretical mechanism to allow us to realise a path graph in a given stack and heap. At run-time we will use multi-granularity locks to effectively lock many more addresses than φ. To formally represent the flowing around the path graph, we give a judgement $h, \sigma \vdash G : \varphi$, and we let $\gamma(h, \sigma, G)$ be the flattened minimal φ that satisfies $h, \sigma \vdash G : \varphi$. We say that an assignment is *valid* in the context of some h, σ, G if it satisfies this judgement.

Note that there will likely be many valid φ for a given h, σ, G. In particular, $\forall h, \sigma, G : h, \sigma \vdash G : \varphi_{max}$ where $\varphi_{max} = \lambda n.Addr$. There will, however, be one minimal φ for a particular h, σ, G. We can define a partial ordering over assignments by lifting $\subseteq$ point-wise: $\varphi_1 \sqsubseteq \varphi_2 \iff \forall n.\varphi_1(n) \subseteq \varphi_2(n)$. We also let $\varphi_1 \sqcap \varphi_2 = \lambda n.\varphi_1 \cap \varphi_2$, i.e. the point-wise intersection of the two assignments.

Theorem 2. *Valid assignments join to make valid assignments:*

$$\left.\begin{array}{l} h, \sigma \vdash G : \varphi_1 \\ h, \sigma \vdash G : \varphi_2 \end{array}\right\} \Longrightarrow h, \sigma \vdash G : (\varphi_1 \sqcap \varphi_2)$$

Proof: Follows from the definitions.

If we define the minimal assignment:

$$\Phi(h, \sigma, G) = \bigsqcap \{\varphi | h, \sigma \vdash G : \varphi\}$$

Using the above theorem, we know that $h, \sigma \vdash G : \Phi(h, \sigma, G)$. Clearly, there cannot be any other valid $\varphi \sqsubset \Phi(h, \sigma, G)$. Now we can finish our notion of correctness by defining $\gamma(h, \sigma, G) = flatten(\Phi(h, \sigma, G))$.

3.5 Proof

We have proved correctness in Isabelle/HOL using Proofgeneral [2]. The file is 800 lines long, takes 30 seconds to process on a 3GHz P4, and is available online [6]. Aside from basic notation, explicit quantifiers, and explicit handling of the cases where partial maps do not contain a mapping from a particular value, the Isabelle/HOL formalism is identical to the one considered here.

Theorem 2 and many auxiliary lemmas were proved automatically. Theorem 1 was a long proof but often the final stages of each case were automatic. In

[2] Although this cannot happen in a language like Java, for simplicity our formalism permits initial stacks/heaps to contain undefined addresses.

particular, the extra details that are required in a proof assistant (but usually omitted in a hand-written proof) can usually be handled automatically at the beginning or end of the proof. We originally proved correctness with a slightly different formalism that had an extra parameter in the execution judgement to accumulate the constructed objects and needed only primitive recursion on A in the (New) rule. We later converted this to the form given in the paper. The conversion required us to manually intervene in the proof, but in all cases except (New) this was very easy, needing only the removal of any references to the extra parameter. Our overall experience with Isabelle was positive, and we enjoy having greater confidence in the correctness of our proof.

4 Related Work

As our work is an implementation of atomic sections we compare it first with the more popular atomic section implementation technique of transactional memory and then with other lock inference techniques.

Transactional memory [10, 1, 17] has trouble with I/O, which presents no problem with lock inference approaches. Conversely, transactions (being dynamic) can handle reflection easily, whereas lock inference has to be conservative. Particularly, plugin systems can cause problems for lock inference, but we expect that JIT techniques would be a solution. Transactional memory needs compiler support to instrument code with logging mechanisms, but lock inference requires much more compiler support in the form of complex analyses. Conversely, transactions use a lot of runtime mechanisms to detect conflict and rollback, whereas lock inference only needs locks at runtime. In terms of performance, transactions can waste cycles rolling back, but have perfect granularity. Lock inference must use conservative approximation and thus will always have worse granularity than transactional memory. However, the granularity of lock inference is still reasonable, and can be very good if ownership types are available [4].

Lock inference algorithms have become more precise over time. Initially they used only points-to sets to statically characterise objects with very coarse granularity [19, 11], or if they did have better precision, they restricted assignment and required annotations [16]. More recently, custom alias analyses have been used to allow instance locks without annotations, falling back to static locks if aliasing is uncertain [7, 9], but the choice of instance/static locks was still on a per-object basis. Only recently [5, 3] have multi-granularity locks been used to allow the instance/static distinction on a per-atomic-section basis. Also, only recently have analyses begun to use translation techniques to handle assignment [5, 3] without coarsening the lock granularity. One key difference between our approach and [3] is that they force termination with a simple static bound, whereas we represent cycles accurately within an NFA. The program analysis used by Khedker et al. [13] was very similar to ours, but there it is used to infer object liveness for the purpose of accelerating garbage collection. We believe our approach is the only one that prevents deadlock with a dynamic mechanism. The other approaches

attempt to statically order lock acquisitions, falling back to static locks if this is not possible.

One contribution of Cherem et al. [3] is a framework for specifying and combining lock inference approaches. Although they distinguished between dereferencing and object offset, whereas we just use fields, we believe our NFA approach can be represented in this framework. However, it is less useful because our approach is monolithic, supporting all the features we need without needing to be combined with other analyses. Another contribution of the above is a notion of correctness that is intuitively similar to ours, but specialised for their approach.

5 Conclusion

We have proved that our lock inference infers at least the objects accessed by the body of an atomic section. Since we use a two phase locking discipline, we therefore know that the atomic section executes atomically. It may not be clear why we used a single-threaded semantics; e.g., how can we be sure that other threads won't change the state so that the locked object was not the one accessed? We get this property for free from the locking discipline, since we lock any shared memory accessed.

We assume that all shared memory accesses occur in atomic blocks, a property that all the more efficient atomic section implementations require. In practice, both lock inference and transactional memory would greatly benefit from a type system to distinguish between thread-local and shared memory (e.g. [17, 1]), since thread-local state need not be protected by locks or logged by a transaction. Some may argue that such a type system would restrict programmers, but choosing between shared and thread local memory is already an important design decision that programmers often document with comments. The type system would enforce these comments by ensuring that local memory is never shared, and shared memory is always accessed within atomic sections.

Our analysis supports read/write locks for better precision, but we omitted this detail for simplicity. We could extend the proof to additionally require that the path graph node where an address is represented is the same as the CFG node where the access occurs during execution. Knowing the statement is sufficient to know what kind of access occurred there. The path graphs already store this information, but in the proof presented here we "flattened" this detail.

Our previous paper gave an implementation that released locks as early as possible, by calculating the difference between the CFG edges either side of each node. It would be good to prove that this is correct and we hope to do that soon. We are working on an implementation for Java, using Soot, that correctly handles features like arrays, functions, exceptions, constructors, finalizers, static initializers, and static members. Extending the proof to cover these constructs is also left as further work.

We believe that using ownership types would give us much better granularity when converting path graphs to actual locking code [4]. For instance, we could lock only the nodes that are iterated through, in a list, rather than every node

in the system. We have avoided using ownership types until now because of the type annotations required. Inferring ownership type annotations would therefore be a useful subject for further work. We believe that with ownership types and knowledge of thread locality, we can infer locks with granularity equivalent or better than manual locking, and additionally with guaranteed correctness.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

References

1. Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–74, New York, NY, USA, 2008. ACM.
2. David Aspinall. Proof general: A generic tool for proof development. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 38–42, London, UK, 2000. Springer-Verlag.
3. Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *Proceedings of the ACM Conference on Program Language Design and Implementation (PLDI 2008), to appear.*, Tucson, Arizona, June 2008.
4. Dave Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, September 2007. `http://pubs.doc.ic.ac.uk/universes-races/`.
5. Dave Cunningham, Khilan Gudka, and Susan Eisenbach. Keep Off The Grass: Locking the Right Path for Atomicity. In *Compiler Construction 2008*, volume 4959 of *Lecture Notes in Computer Science*, pages 276–290, April 2008.
6. Dave Cunningham. Isabelle/HOL file containing correctness proof of lock inference, 2008. Available online at `http://www.doc.ic.ac.uk/~dc04/ftfjp08.thy`.
7. Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–296, New York, NY, USA, 2007. ACM Press.
8. K.P. Eswaran, J. Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, 1976.
9. Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, September 2007. To appear.
10. T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25, 2006.
11. Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

12. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
13. Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.*, 30(1):1, 2007.
14. E. Koskinen and M. Herlihy. Dreadlocks: Efficient Deadlock Detection. In *Proceedings from the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '08), Special track on Multicore*, June 2008. To appear.
15. David Lomet. Process structuring, synchronization, and recovery using atomic actions. *ACM SIGOPS Operating Systems Review*, 11(2):128–137, 1977.
16. B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Notices*, 41(1):346–358, 2006.
17. Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 51–62, New York, NY, USA, 2008. ACM.
18. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.
19. Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. *SIGPLAN Not.*, 41(1):334–345, 2006.

An Operational Semantics including "Volatile" for Safe Concurrency*

John Boyland

University of Wisconsin–Milwaukee, USA
boyland@cs.uwm.edu

Abstract. In this paper, we define a novel "write-key" operational semantics for a kernel language with fork-join parallelism, synchronization and "volatile" fields. We prove that programs that never suffer write-key errors are exactly those that are "data race free" and also those that are "correctly synchronized" in the Java memory model. This 3-way equivalence is proved using Twelf.

1 Introduction

This work is motivated by the desire to define a type system for a Java-like language to prevents data races. Data races are intrinsically a multi-threaded issue. However a scalable type system or program analysis analyzes each thread, indeed every method body separately, using invariants and annotations to ensure that interactions follow desired patterns. It is well known that deadlock can be prevented by requiring that mutexes be acquired in strictly increasing order. Here we show how we can characterize programs without data races in a similar way, that is without explicitly needing to refer to multiple threads.

There are different understandings of what a data race is. At an intuitive level, a data race occurs when an execution of a multi-threaded program leads to the point where two conflicting accesses in two different threads occur "at the same time." Two accesses are *conflicting* if they are to the same object's field and one of them is a write. Somewhat more precisely, the current Java memory model (JMM) [15] defines a "happens before" partial order; a program is *correctly synchronized* if in all sequentially consistent executions, two conflicting accesses are always ordered by "happens before." Reading and writing of "volatile" fields affect the "happens before" order and thus whether a program is correctly synchronized. Both of these techniques explicitly involve reasoning about multiple threads at once.

This paper addresses this situation with the following contributions:

– It defines a simple imperative language with Java-style (re-entrant) monitors, volatile fields and fork-join parallelism. A novel aspect of the operational

* Work supported in part by the National Science Foundation (CCF-0702635). The opinions expressed here are not necessarily those of the National Science Foundation or the US Government.

semantics is that the system constructs and passes "write keys" to simulate the "happens before" relation.
- The paper defines that a program is data-race free if no execution has a "write-key error" in which a thread attempts to access a (non-volatile) field for which it does not possess the write key.
- It is proved that this characterization is equivalent *both* to the intuitive concept of lack of "simultaneous" conflicting accesses, *and* to the JMM-inspired definition of "happens before"-ordered accesses. The proof is mechanically checked in Twelf's metalogic [17].

A corollary to the last contribution is that the two earlier conceptions of data-race freedom are equivalent, a result that I have not seen previously.

The advantage of the "write-key error" conception for data races is that write-key errors are detected (and can be prevented) thread locally. In other words, if a type system can ensure for each thread that it always possesses the write keys for the (non-volatile) fields that it accesses and that it has exclusive access to fields it writes, then the entire program is thread safe. Moreover, since a write-key error causes a thread to get stuck in our semantics, and since (full) deadlocks always means the program as a whole is stuck, then if a type system enjoys "progress" and "preservation" over the operational semantics, then *per force* the type system will also prevent race conditions.

2 Background on the Current Java Memory Model

This section briefly describes multi-threading primitives in Java and the "happens before" relation of the current Java Memory Model.

In Java, a new thread can be started which executes a given `run` method; we call this a fork action. At the other end, one may wait for a thread t to complete execution by executing t`.join()`; this is a join action. Thread mutual exclusion is effected by "synchronizing" on an object o: `synchronized (`o`) {` *body* `}`. The runtime system ensures that two separate threads that both synchronize on the same object (known by its role as a *mutex*) mutually exclude each other's "body" instructions. When a synchronized block is executed, it first attempts to acquire the mutex, blocking if some other thread is currently executing a synchronized block on the same object. Once acquisition is successful, the body is executed after which the mutex is released. Synchronized statements in Java are "re-entrant" in that if a synchronization block is nested dynamically within another synchronization block on the same object, the inner synchronization succeeds immediately.

Fields in Java may be declared as "volatile." This designation may be seen as a declaration that these fields will be read and written without mutual exclusion. More importantly, accesses to volatile fields (denoted readv and writev) constrain the memory model.

A memory model is a contract between the programmer on the one hand and the compiler and the runtime system on the other hand. The most informative model for programmers is a "sequentially consistent" model that indicates that

execution will always be consistent with a system in which the thread interleaving of instructions ensures that each instruction fully executes before the next starts. Sequential consistency however is very limiting for a compiler. Consider the following situation where two threads are executing in parallel, `x` and `y` represent shared mutable locations and `r`n represent thread local "registers.":

Initially `x = y = 0`	
`x = 1`	`r2 = y`
`y = 2`	`r1 = x`

In a sequentially consistent execution, no interleaving of the threads could result in `r1 = 0, r2 = 2`. Thus a compiler would not be permitted to reorder the two write actions even though they have no data dependencies.

For such reasons, most memory models only guarantee sequential consistency for fields declared "volatile." For other fields, threads must use mutual exclusion techniques. The (intuitive) guarantee for normal (non-volatile) fields is that if a program is *data-race free*, that is, if no sequentially consistent execution ever exhibits a "race condition" for a normal field, then that program will enjoy sequentially consistent semantics. A *race condition* is when one thread is ready to write an object's field when another thread is ready to read or write the same object's field. If a program *could* exhibit a race condition under a sequentially consistent semantics, then most memory models usually do not guarantee a sequentially consistent semantics. In the small example given above, there is a race condition. Thus a compiler is justified if it wishes to reorder the statements, even though this reordering violates sequential consistency.

In the current Java memory model (JMM) [15], the guarantee is expressed in a different way. First, a "synchronizes with" relation is defined:

1. A release action synchronizes with an acquire action on the same object;
2. A writev action synchronizes with a readv action on the same object's field;
3. A fork action synchronizes with the first action in the spawned thread;
4. The last action in a thread synchronizes with a join action on that thread.

Additionally the default initialization of a field (with `null` for reference types) synchronizes with all actions in all threads.

Then the "happens before" partial order is defined as the transitive closure of (1) the intra-thread execution order and (2) the "synchronizes with" relation over actions already ordered by the execution.

Specifically of interest to the present paper, the JMM defines what it means to be "correctly synchronized":

> A program is *correctly synchronized* if and only if in all sequentially consistent executions, all conflicting accesses to non-volatile variables are ordered by "happens-before" edges.

The JMM (and generalizations [18]) guarantees that a correctly synchronized program will observe sequentially consistent semantics. This guarantee appears rather different than that expressed concerning "data race free" but as proved in Sect. 5, the definitions are equivalent for our small concurrent language.

```
class Node {
  Node next;
  Node(Node n) { next = n; }

  Node getNext() { next; }

  int count() {
    if this == null then 0
    else 1 + next.count();
  }

  Node copy() {
    if this == null then null
    else new Node(next.copy());
  }

  Node nap(Node n) {
    if this == null then n
    else (next = next.nap(n);
          this);
  }

  void add1() {
    this.nap(new Node(null));
  }
}
```

Fig. 1. A simple node class.

```
class Race {
  Node nodes;

  Race() { }

  int get() {
    nodes.count();
  }

  void inc() {
    nodes = nodes.add1();
  }
}


class Main {

  void main() {
    let t = new Race() in
      ( fork { t.inc(); t.get(); }
        t.inc(); t.get() );
  }

}
```

Fig. 2. A class with an unprotected field; and a test harness.

3 Example

Figure 1 declares a node class. The surface syntax resembles Java, but method bodies contain expressions, not statements. For instance, `getNext()` returns the next field. The `count` method shows another difference: since the language omits dynamic dispatch for simplicity, one can call methods on null references. In the body, one may test for null. In this way, we can model so-called "static" methods. The `copy` method performs a deep copy; `nap` does a destructive append; `add1` extends the list by one node. The `Node` class has mutable state and thus cannot be safely used in a concurrent program without additional restrictions.

We now define several different classes wrapping a node list with the same interface: an `inc` method that adds to the list and a `get` method that counts the size of the current list. The first implementation, `Race` (Fig. 2), does nothing to protect the list. The main program forks off a thread that calls `inc` and `get` and then proceeds to do the same calls in its own thread. Lacking synchronization, the call `t.inc()` in one thread conflicts with `t.inc()` or `t.get()` in the other.

The traditional technique ("standard practice") for protecting mutable state is to designate a *protecting* object for each piece of mutable state (one object may protect many others) and ensure that all accesses to the state occur dynamically

```
class Traditional {
  Node nodes;

  int get() {
    synch (this) do
      nodes.count();
  }

  void inc() {
    synch (this) do
      nodes = nodes.add1();
  }
}
```

Fig. 3. Traditional approach.

```
class UsingVolatile {
  volatile Node nodes;

  int get() {
    nodes.count();
  }

  void inc() {
    synch (this) do
      nodes = nodes.copy().add1();
  }
}
```

Fig. 4. Using volatility.

only within a synchronization on the protecting object. For example, see class `Traditional` in Fig. 3; the bodies of the methods `get()` and `inc()` include synchronizations around the access of the mutable state.

If `get()` calls are frequent and updates very infrequent, one can do better with a less-known pattern using volatile variables. Figure 4 shows how a volatile field can substitute for synchronization. The reading method can simply access the nodes directly using a volatile field read, and then traverse the list without synchronization. The incrementing method copies the structure before modifying it, to avoid interfering with `get` calls. Furthermore, `inc` is synchronized to ensure that two increments are not carried out in parallel (to preserve "atomicity" [9], an important concept beyond the scope of this paper). We permit interleaving of `get()` and `inc()` calls since the `inc()` method never updates state the `get()` method can see, except for the volatile field.

4 Operational Semantics

This section defines the syntax and dynamic semantics of the paper's kernel concurrent language. The set of all fields is F. A subset $F_V \subseteq F$ are "volatile" and one (Lock $\in F_V$) holds the state of the mutex associated with each object.

4.1 Syntax

Figure 5 gives the syntax. For simplicity, we omit primitive types and arithmetic operators. Expressions include literal object references (natural numbers) and uses of local variables. A new object can be allocated with the given set of fields. Fields of objects can be read or written. The "`let`," "`if`" and "`while`" constructs are conventional. Procedure calls are included, but not dynamic dispatch because the details would obscure the emphasis of this work.

The concurrency-related terms are fork-join terms (`fork` creates a new thread and starts it; and `join` waits for it to terminate) and synchronization (`synch`

$e ::=$		*expression term:*
	o	*literal reference*
	x	*program variable*
	`new` $(\overline{f})$	*allocation*
	e`.`f	*field read*
	e`.`f`:=`e	*field write*
	`let` x`=`e `in` e	*local*
	`if` c `then` e `else` e	*conditional*
	`while` c `do` e	*loop*
	$m(\overline{e})$	*procedure call*
	`fork` e	*fork a thread*
	`join` e	*get thread result*
	`synch` e `do` e	*synchronization*
	`hold` o `do` e	*... in execution*
	e_1`;`e_2 $\triangleq$	`let _=`e_1 `in` e_2

$c ::=$		*conditional term:*
	`true`	*true*
	`not` c	*negation*
	c `and` c	*conjunction*
	e `==` e	*equality*
	`false` $\triangleq$	`not true`
	c `or` c' $\triangleq$	`not(not` c `and not` c'`)`

$t ::= e \mid c$		*term*
$d ::= m(\overline{x})$ `=` e		*procedure definition*
$g ::= d$`;...;`d		*program*

Fig. 5. Syntax.

and `hold`). A `hold` expression is used to indicate that this thread is currently executing a `synch` statement.

The examples in the previous section use a surface syntax with classes, methods and types. A simple translation (not shown) can strip out these features.

4.2 Semantics

This section defines the small-step operational semantics. Novel here is the use of "write keys." Write keys allow us to separate the notion of "happens before" from considering the execution of multiple threads and instead look at a single thread at a time. A (possibly new) *write key* (a natural number) is generated whenever a normal field is written. This field is added to the *knowledge* of the thread that performed the write. Knowledge is monotonically non-decreasing. Write keys are passed from one thread to another during synchronization actions *indirectly* through memory. In this way, if a write in one thread happens before the read in the other thread, the read is guaranteed to have the necessary key.

The main evaluation relation $(\mu; \theta; \kappa) \underset{g}{\rightarrow} (\mu'; \theta'; \kappa')$ relates triples:

- μ maps a location $(o.f)$ to a pair (W, o') of a set of write keys and a value. For a normal field, $W = \{w\}$, where w is the key from the most recent write; for a volatile field, W is the set of keys from all threads having written it.
- θ maps a thread identifier (natural number) to the expression that the thread is currently executing.
- κ maps a thread identifier to its set of known write keys.
- g lists the procedure definitions.

The following notation is used for map update:

$$f[x \mapsto v](x') = \begin{cases} v & \text{if } x = x' \\ f(x) & \text{otherwise} \end{cases} \qquad f[x \overset{\bigcirc}{\mapsto} v] = f[x \mapsto f(x) \bigcirc v]$$

$$\mathrm{T}[\bullet] ::= \bullet \mid \mathrm{T}[\bullet].f \mid \mathrm{T}[\bullet].f \texttt{ := } e \mid o.f \texttt{ := } \mathrm{T}[\bullet] \mid m(\overline{o},\mathrm{T}[\bullet],\overline{e})$$
$$\mid \texttt{let } x\texttt{=}\mathrm{T}[\bullet] \texttt{ in } e \mid \texttt{if } \mathrm{T}[\bullet] \texttt{ then } e \texttt{ else } e \mid \texttt{synch } \mathrm{T}[\bullet] \texttt{ do } e$$
$$\mid \texttt{hold } o \texttt{ do } \mathrm{T}[\bullet] \mid \mathrm{T}[\bullet] \texttt{ and } c \mid \texttt{not } \mathrm{T}[\bullet] \mid \mathrm{T}[\bullet] \texttt{ == } e \mid o \texttt{ == } \mathrm{T}[\bullet]$$

EVAL
$$\frac{\theta p = \mathrm{T}[t] \qquad (\mu;\theta;\kappa;t) \xrightarrow[g]{p} (\mu';\theta';\kappa';t')}{(\mu;\theta;\kappa) \underset{g}{\rightarrow} (\mu';\theta'[p \mapsto \mathrm{T}[t']];\kappa')}$$

E-CALL
$$\frac{g = \ldots;\ m(\overline{x})\texttt{ = }e;\ \ldots \qquad |\overline{x}| = |\overline{o}|}{(\mu;\theta;\kappa;m(\overline{o})) \xrightarrow[g]{p} (\mu;\theta;\kappa;[\overline{x \mapsto o}]e)}$$

E-LET
$$(\mu;\theta;\kappa;\texttt{let } x \texttt{ = } o_1 \texttt{ in } e_2) \xrightarrow[g]{p} (\mu;\theta;\kappa;[x \mapsto o_1]e_2)$$

E-IF
$$(\mu;\theta;\kappa;\texttt{if } c \texttt{ then } e_{\texttt{true}} \texttt{ else } e_{\texttt{false}}) \xrightarrow[g]{p} (\mu;\theta;\kappa;e_c)$$

E-WHILE
$$(\mu;\theta;\kappa;\texttt{while } c \texttt{ do } e) \xrightarrow[g]{p} (\mu;\theta;\kappa;\texttt{if } c \texttt{ then } e\texttt{; while } c \texttt{ do } e \texttt{ else } 0)$$

E-NOTNOTTRUE
$$(\mu;\theta;\kappa;\texttt{not false}) \xrightarrow[g]{p} (\mu;\theta;\kappa;\texttt{true})$$

E-ANDTRUE
$$(\mu;\theta;\kappa;\texttt{true and } c) \xrightarrow[g]{p} (\mu;\theta;\kappa;c)$$

E-ANDFALSE
$$(\mu;\theta;\kappa;\texttt{false and } c) \xrightarrow[g]{p} (\mu;\theta;\kappa;\texttt{false})$$

E-EQUALTRUE
$$\frac{o = o'}{(\mu;\theta;\kappa;o\texttt{==}o') \xrightarrow[g]{p} (\mu;\theta;\kappa;\texttt{true})}$$

E-EQUALFALSE
$$\frac{o \neq o'}{(\mu;\theta;\kappa;o\texttt{==}o') \xrightarrow[g]{p} (\mu;\theta;\kappa;\texttt{false})}$$

Fig. 6. Non-concurrency-related evaluation rules.

Evaluation proceeds using EVAL: a thread is chosen non-deterministically and evaluates one step. Here $(\mu;\theta;\kappa;t) \xrightarrow[g]{p} (\mu';\theta';\kappa';t')$ states that thread p in program g makes progress by converting t into t' while side-effecting μ, θ and κ.

For explanatory reasons, the evaluation rules are presented in two groups. The first group of evaluation rules (Fig. 6) are those that have no side-effects. A procedure call uses rule E-CALL once all the parameters are evaluated: we find a procedure in the program with the correct number of arguments and replace the call with the procedure body, substituting the parameters. A `let`-bound variable is substituted in the body once its value is ready. An `if` with a constant boolean is evaluated by choosing the appropriate branch. A `while` loop is converted immediately into an `if`. Conditions use short-circuit evaluation (E-ANDFALSE).

Figure 7 includes the remaining evaluation rules. As mentioned previously, normal fields are associated with a write key that indicates what knowledge is needed to read the field. When a `new` expression is encountered, all of the fields are initialized with null using a write key (0) that all threads know. (This

E-New

$$\frac{f_0 = \text{Lock} \qquad (o, \text{Lock}) \notin \text{Dom}(\mu) \qquad f_i \text{ distinct}}{(\mu; \theta; \kappa; \texttt{new (}f_1\texttt{,...,}f_n\texttt{)}) \xrightarrow[g]{p} (\mu[(o, f_i) \mapsto (\{0\}, 0) \mid 0 \le i \le n]; \theta; \kappa; o)}$$

E-Read

$$\frac{\mu(o.f) = (\{w\}, o') \qquad w \in \kappa(p) \qquad f \notin F_V}{(\mu; \theta; \kappa; o\texttt{.}f) \xrightarrow[g]{p} (\mu; \theta; \kappa; o')}$$

E-Write

$$\frac{\mu(o.f) = (\{w\}, \_) \qquad w \in \kappa(p) \qquad f \notin F_V \qquad w' \text{ arbitrary} \qquad \mu' = \mu[o.f \mapsto (\{w'\}, o')] \qquad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]}{(\mu; \theta; \kappa; o\texttt{.}f\texttt{:=}o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

E-ReadV

$$\frac{\mu(o.f) = (W, o') \qquad \text{Lock} \neq f \in F_V \qquad \kappa' = \kappa[p \overset{\cup}{\mapsto} W]}{(\mu; \theta; \kappa; o\texttt{.}f) \xrightarrow[g]{p} (\mu; \theta; \kappa'; o')}$$

E-WriteV

$$\frac{\mu(o.f) = (W, \_) \qquad \text{Lock} \neq f \in F_V \qquad \mu' = \mu[o.f \mapsto (W \cup \kappa(p), o')]}{(\mu; \theta; \kappa; o\texttt{.}f\texttt{:=}o') \xrightarrow[g]{p} (\mu'; \theta; \kappa; o')}$$

E-Fork

$$\frac{(p, \text{Lock}) \notin \text{Dom}(\mu)}{(\mu; \theta; \kappa; \texttt{fork } e) \xrightarrow[g]{p} (\mu[(p', \text{Lock}) \mapsto (\{0\}, 0)]; \theta[p' \mapsto e]; \kappa[p' \mapsto \kappa(p)]; p')}$$

E-Join

$$\frac{\theta(p') = o}{(\mu; \theta; \kappa; \texttt{join } p') \xrightarrow[g]{p} (\mu; \theta; \kappa[p \overset{\cup}{\mapsto} \kappa(p')]; o)}$$

E-Re-Enter

$$\frac{\mu(o.\text{Lock}) = (\emptyset, p)}{(\mu; \theta; \kappa; \texttt{synch } o \texttt{ do } e) \xrightarrow[g]{p} (\mu; \theta; \kappa; e)}$$

E-Acquire

$$\frac{\mu(o.\text{Lock}) = (W, 0) \qquad W \neq \emptyset \qquad \mu' = \mu[(o.\text{Lock}) \mapsto (\emptyset, p)] \qquad \kappa' = \kappa[p \overset{\cup}{\mapsto} W]}{(\mu; \theta; \kappa; \texttt{synch } o \texttt{ do } e) \xrightarrow[g]{p} (\mu'; \theta; \kappa'; \texttt{hold } o \texttt{ do } e)}$$

E-Release

$$\frac{\mu(o.\text{Lock}) = (\emptyset, p) \qquad \mu' = \mu[(o.\text{Lock}) \mapsto (\kappa(p), 0)]}{(\mu; \theta; \kappa; \texttt{hold } o \texttt{ do } o') \xrightarrow[g]{p} (\mu'; \theta; \kappa; o')}$$

Fig. 7. Remaining evaluation rules.

follows the JMM—default initialization synchronizes with the first action in every thread.) Every object is allocated with a mutex (special field Lock).

Field reads and writes of non-volatile fields (E-Read, E-Write) require that the thread has knowledge of the write that produced the value: $w \in \kappa(p)$. For a write, an arbitrary write key w' is used to label the new write. In general, this may be one that no thread is aware of. Using such a key would cause the `Race` program in earlier Fig. 2 to get stuck when the second increment executes.

One way in which knowledge of writes can be transmitted is through volatile fields (E-ReadV, E-WriteV). Writing a volatile field adds the thread's knowledge κp to the memory with the written value. When the volatile field is read, the reading thread picks up this knowledge. This follows the JMM rule that says that writing a volatile field synchronizes with all following reads.

For E-Fork, the new thread gets the knowledge of the "forker." This corresponds to the JMM rule that a fork synchronizes with the first action in the new thread. An object is allocated to represent the thread. In E-Join, this thread

can only progress if the other thread has finished execution (down to a value). It gets a copy of all the thread's knowledge. This follows from the JMM's rule that the final action in a thread synchronizes with a thread that "joins" it.

During synchronization, the lock's value is replaced with the number of the acquiring thread, and the knowledge is replaced by the empty set. In E-Re-Enter, if we synchronize on a lock that this thread already has acquired, the body is simply executed without any effect on the lock. This last rule corresponds to Java's re-entrant monitors; here, we avoid the need to count multiple entrances because the evaluation rule drops the release action as well as the acquire action.

If the lock is not held by any thread (E-Acquire), the lock field is assigned the number of this thread, and we get the keys from the lock. The synchronization block is then converted into a hold block. When the body has finished evaluation (E-Release), the lock is given the knowledge of the current thread. This knowledge is thus made available for the next thread which acquires the lock. These rules again follow from the JMM.

The semantics defined here is sequentially consistent, but if a thread lacks the necessary write key, it gets stuck. Thus if the program has race conditions, it *may* get stuck (but may not, for instance if an old key is chosen by E-Write). A type system for this language that enjoys progress and preservation for *all* executions will prevent this (and the other problems not mentioned). We have designed a type system [19] based on fractional permissions [5, 6] that we believe will achieve this goal, but space precludes including it here.

5 Equivalence

Programs that execute in our operational semantics without ever blocking because of missing write keys are "correctly synchronized" according to (our variant) of the Java Memory Model *and* to the traditional definition of "race free." In other words, we show a three-way equivalence.

In order to prove equivalence, we need to formally define the aspects we are showing equivalent. To start with, we restrict programs so that they do not include arbitrary object reference constants or partially executed synchronizations:

Definition 1. *A program g is* valid *if every declaration $m(\bar{x}) = e$ in g has no instance of* `hold` *nor any literal object reference except the null reference* 0.

Execution starts by calling the `main` procedure in thread 0, which starts with no knowledge except write key 0.

Definition 2. *The* initial state *I is the state*

$$I = ([(0, \mathit{Lock}) \mapsto (\{0\}, 0)][0 \mapsto \texttt{main()}][0 \mapsto \{0\}]) \quad .$$

We formalize what it means for there to be a *write key error* in a program:

Definition 3. *A program $g = \overline{d}$ has a write key error if for some execution $I \xrightarrow[g]{*} (\mu, \theta, \kappa)$ in which a read or write access on a non-volatile field $o.f$ is ready to execute in thread p ($\theta p = T[o\,.\,f := o']$ or $\theta p = T[o\,.\,f]$, where $f \notin F_V$), and the thread does not have the required write key: ($\mu(o.f) = (\{w\}, \_)$ and $w \notin \kappa p$).*

Definition 4. *Two terms t_1 and t_2 are* conflicting accesses *of a non-volatile field $o.f$ ($f \notin F_v$) if one of them is a write to this field ($t_i = o\,.\,f := o'$) and the other is a write ($t_{3-i} = o\,.\,f := o''$) or a read ($t_{3-i} = o\,.\,f$) of the same field.*

Next, we formalize what it means to have a "race condition": a write access to a field happens at the "same time" as a read access to the same field, and that field is not volatile:

Definition 5. *A program $g = \overline{d}; e_0$ exhibits a* race condition *if there is some execution $I \xrightarrow[g]{*} (\mu, \theta, \kappa)$ such that for two threads $p_1 \neq p_2$ we have $\theta(p_i) = T(t_i)$ and t_1, t_2 are conflicting accesses.*

Before we can define what it means to be "correctly synchronized," we must define an "action" and the "happens-before" relation for actions:

Definition 6. *An* action *λ is an evaluation $(\mu; \theta; \kappa; t) \xrightarrow[g]{p} (\mu'; \theta'; \kappa'; t')$. An evaluation sequence $I \xrightarrow[g]{*} (\mu, \theta, \kappa)$ induces the actions above the line for each instance of* Eval*: $\lambda_1, \lambda_2, \ldots, \lambda_n$.*

Definition 7. *Given an execution $\lambda_1, \ldots, \lambda_n$, we define a* happens-before *(written $i \sqsubset j$) relation on the subset of natural numbers $\{1, \ldots, n\}$. It is smallest transitive relation that includes the following pairs:*

1. *$i \sqsubset j$ if $i < j$ and λ_i is an instance of* E-Release *and λ_j is an instance of* E-Acquire *on the same object.*
2. *$i \sqsubset j$ if $i < j$ and λ_i is an instance of* E-WriteV *and λ_j is an instance of* E-ReadV *on the same field.*
3. *$i \sqsubset j$ if $i < j$ and $\lambda_i = \_ \xrightarrow[g]{p} \_$ and $\lambda_j = \_ \xrightarrow[g]{p} \_$.*
4. *$i \sqsubset j$ if $i < j$ and $\lambda_i = (\mu; \theta; \kappa; \texttt{fork}\ t) \xrightarrow[g]{p} (\mu'; \theta'; \kappa'; q)$ and $\lambda_j = \_ \xrightarrow[g]{q} \_$.*
5. *$i \sqsubset j$ if $i < j$ and $\lambda_i = \_ \xrightarrow[g]{q} \_$ and $\lambda_j = (\mu; \theta; \kappa; \texttt{join}\ q) \xrightarrow[g]{p} (\mu'; \theta'; \kappa'; t)$.*

It can be easily shown that $\sqsubset$ is a partial order compatible with $<$.

Our final definition is for *correctly synchronized* in the style of the JMM:

Definition 8. *A program $g = \overline{d}; e$ is correctly synchronized if for any execution of g: $\lambda_1, \ldots, \lambda_n$ and any i for which λ_i is an instance of* E-Write *and any j for which λ_j is an instance of* E-Read *or* E-Write *for the same field, then either $i \sqsubset j$ or $j \sqsubset i$.*

It might seem that because our operational semantics detects race conditions, the conflicting access would never execute and thus could not demonstrate an incorrect synchronization, but because write keys are arbitrary, the write could use 0 and thus enable execution. The semantics does not ensure that *all* executions of a program with race conditions will get stuck, just that there will be *some* execution that does.

We now show the three-way equivalence between the three conceptions of race-freedom:

Theorem 1. *The following statements about a valid program g are equivalent:*

1. *g exhibits a race condition;*
2. *g has a write key error;*
3. *g is incorrectly synchronized.*

Proof. (Sketch)

(1) ⇒ (2): Suppose we have a program with a race condition. Starting with the execution state that exhibits the race condition, we choose to evaluate the write first. If this write cannot execute because of a missing write key, we are done. Otherwise we choose a new write key not known by the other thread, and we now have a write key error.

(2) ⇒ (3): We prove the contrapositive: if the program is correctly synchronized, there will be no write-key error. This is because if there is a happens-before connection between two actions, the thread knowledge of the second will include that produced by the first, and thus the second access will succeed. The connection between write key knowledge and happens-before follows from the fact that the knowledge never decreases (the first case for happens-before) and the other cases for happens-before involve the reader/acquirer getting all the write keys left by the writer.

(3) ⇒ (1): Suppose we have an incorrectly synchronized program, in which the code of the first action λ_i is a write executed in thread p and the second action λ_j is an access executed in thread q. (The case that λ_i is a read and λ_j is a write is analogous.)

If the actions are already consecutive, we have the required race condition in the state just before the first executed. Otherwise, we consider how evaluation actions can be reordered (between different threads, never within a thread) to get the accesses adjacent. We partition the intervening actions into those that happen before j and those which do not. The second must include action i, from the definition of incorrect synchronization. We find the last action λ_* in the second group. It cannot be "happens before" any in the first group, or a transitive happens-before relation would exist putting it *in* the first group. Now we reorder it step-by-step with all later actions until it is after λ_j. If λ_* was λ_i, then the last reordering would have resulted in the required race condition. Otherwise, now that it is after λ_j we have reduced the number of intervening instructions. This process must terminate at some point.

6 Extensions

Extending the simple language here to full Java is almost entirely just a matter of complex but uninteresting details. Static fields and static synchronization can be modeled using instance fields and instance synchronization of singleton objects. Types, primitive values and dynamic dispatch have no effect on concurrency. Java 5 adds a new library of synchronization primitives, but it has a reference implementation in core Java. Timeout and timing issues can be modeled by claiming that each step of execution takes 1 nanosecond.

The `Thread` class includes a number of deprecated methods that permit one thread to suspend or terminate another. These we can omit from the formalism. Other methods such as `holdsLock` (because one can only use it to query the current thread) can be implemented without affecting the proof substantially.

Java's `wait`/`notify` system would require substantive changes to the formalism. When a thread calls `wait`, it first releases the object's lock, then it waits to be "notified" and then it waits to re-acquire the lock. The lock release and acquisition lead to the corresponding standard happens-before relations. Another missing piece is thread interruption (and the corresponding interrupted exception). My guess is that the proof could be modified to handle `wait` and interruption.

7 Related Work

The current Java memory model is much more complex than what is modeled here. In particular it gives semantics for programs that are *not* properly synchronized. Since its publication, it has been generalized [18], Apsinall and Ševčík [1] formally prove the main guarantee—that correctly synchronized programs will have a sequentially consistent semantics (whereas the work described here *assumes* sequential consistency). The initialization of reference fields causes some concern which we avoid by using a universally known write key for initialization.

Cenciarelli, Knapp and Sibilio [8] give a vastly different semantics of the Java Memory Model based on configuration structures. As with the papers just reviewed, it handles all Java programs, not just properly synchronized ones, and does not assume sequential consistency. The present author must confess that he was unable to understand the details.

Type systems have been proposed that prevent race conditions and sometimes deadlocks in concurrent programming languages. Flanagan and Abadi [10, 11] define two separate type systems for avoiding races, both of which are accompanied by operational semantics. One is based on Gordon and Hankin's concurrent object calculus [13] in which mutable objects are represented in the syntax as concurrent processes. The other uses a conventional store. Neither semantics directly detects race conditions, nor includes "volatile." In either case, a race condition is defined as the (global) possibility that a write could occur at the same time as a read of the same field. (In one system [11], two "simultaneous" reads are also considered a race.) The type system maintains certain invariants that are shown to prevent data races.

Later work (such as Flanagan and Freund [12], Greenhouse [14] and Boyapati and Rinard [4, 3]) omit formal specification of operational semantics, implicitly following the same approach just outlined. Volatile fields, if they are handled at all, are simply regarded as loopholes in the type system.

Permandla and Boyapati [16] define a small-step semantics for a subset of Java virtual machine language (JVML) including synchronization (but not volatile fields) and show that well-typed programs are free of concurrency errors. The semantics however enforces an ownership model and uses method an-

notations that indicate required locking state. The operational semantics is not independent of the type system.

Guava [2] uses a type system to prevent races in a dialect of Java. Guava permits reader/reader parallelism, but omits volatiles. Guava is defined by (informally described) compilation to Java byte-code. Guava is intended as a practical programming language rather than as a minimal concurrent language.

Brookes [7] gives the semantics of a concurrent program by defining its set of "action traces." Roughly this means that all possible interleavings are considered. A race condition in which a write to mutable state is directly interleaved with another access to the same state is "catastrophic," in that this particular trace immediately aborts. The semantics omits "volatile."

8 Conclusions

This paper defines an operational semantics of volatile fields that enables a type system to reason compositionally about them. It uses write keys to detect threading violations. It shows that write-key errors occur if and only if the program may exhibit a race condition, if and only if it is not correctly synchronized.

Acknowledgments

I thank Aaron Greenhouse and Jonathan Aldrich for providing helpful feedback on early drafts. I thank Yang Zhao, Bill Retert and Mohamed ElBendary for frequent conversations on the topic. I thank the many anonymous reviewers for their comments.

SDG

References

1. David Aspinall and Jaroslav Ševčík. Formalising Java's data race free guarantee. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*. Springer, 1007.
2. David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *OOPSLA'00 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, *ACM SIGPLAN Notices*, 35(10):382–400, October 2000.
3. Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. Ph.D., Massachusetts Institute of Technology, February 2004.
4. Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA'01 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, *ACM SIGPLAN Notices*, 36(11):56–69, November 2001.
5. John Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

6. John Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 283–295. 2005.
7. Stephen Brookes. A semantics for concurrent separation logic. In *CONCUR 2004—15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2004.
8. Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP'07 — Programming Languages and Systems, 16th European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*. Springer, 2007.
9. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, *ACM SIGPLAN Notices*, 38:338–349, May 2003.
10. Cormac Flanagan and Martín Abadi. Object types against races. In *CONCUR '99—10th International Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 1999.
11. Cormac Flanagan and Martín Abadi. Types for safe locking. In *ESOP'99 — Programming Languages and Systems, 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer, 1999.
12. Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, *ACM SIGPLAN Notices*, 35(5):219–232, May 2000.
13. Andrew D. Gordon and Paul D. Hankin. A concurrent object calculas: Reduction and typing. In *HCLC '98, 3rd International Workshop on High-Level Concurrent Languages*. Elsevier, September 1998. Published as *Electronic Notes in Theoretical Computer Science* 16 No. 3 (1998), `http://www.elsevier.nl/local/entcs/volume16.html`.
14. Aaron Greenhouse. *A Programmer-Oriented Approach to Safe Concurrency*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2003.
15. Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 378–391. ACM Press, 2005.
16. Pratibha Permandla and Chandrasekhar Boyapati. A type system for preventing data races and deadlocks in the Java virtual machine language. In *ACM SIGPLAN/SIGBED 2007 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, pages 1–10. ACM Press, June 2007.
17. Frank Pfenning and Carsten Schürmann. Twelf user's guide, vesion 1.4. Available at `http://www.cs.cm.edu/~twelf`, 2002.
18. Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *PPoPP'07: ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 161–172. March 2007.
19. Yang Zhao. *Concurrency Analysis Based on Fractional Permissions*. PhD thesis, University of Wisconsin-Milwaukee, 2007.

Auxiliary Materials

See `http://www.cs.uwm.edu/~boyland/papers/simple-concur.html` for how to download the Twelf proof.

Securing Java with Local Policies

Massimo Bartoletti[1], Gabriele Costa[2], Pierpaolo Degano[1],
Fabio Martinelli[2], and Roberto Zunino[3]

[1] Dipartimento di Informatica, Università di Pisa, Italy
[2] Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche, Italy
[3] Dipartimento di Informatica e Telecomunicazioni, Università di Trento, Italy

Abstract. We propose an extension to the security model of Java. It allows for specifying, analysing and enforcing history-based policies. Policies are defined by finite state automata recognizing the permitted execution histories. Programmers can sandbox an untrusted piece of code with a policy, which is enforced at run-time through its *local* scope. A static analysis allows for optimizing the execution monitor, that will only check the program points where some security violation may actually occur.

1 Introduction

A fundamental concern of security is to ensure that resources are used correctly. Devising expressive, flexible and efficient mechanisms to control resource usages is therefore a major issue in the design and implementation of security-aware programming languages. The problem is made even more crucial by the current programming trends, which allow for reusing code, and exploiting services and components, offered by (possibly untrusted) third parties. It is common practice to pick from the Web some scripts, or plugins, or packages, and assemble them into a bigger program, with little or no control about the security of the whole.

Stack inspection, the mechanism adopted by Java [23] and the .NET CLR [31], offers a pragmatic setting for access control. Roughly, each frame in the call stack represents a method; methods are associated with "protection domains", that reflect their provenance; a global security policy grants each protection domain a set of permissions. Code includes local checks that guard access to critical resources. At run-time, an access authorization is granted when *all* the frames on the call stack have the required permission (a special case is that of privileged calls, that trust the methods below them in the call stack). Being strongly biased towards implementation, this mechanism suffers from some major shortcomings. First, local checks must be explicitly inserted into code by the programmer. Since forgetting even a single check might compromise the safety of the whole application, programmers have to carefully inspect their code. This may be cumbersome even for small programs, and it may lead to unnecessary checking. Second, many security policies are not enforceable by stack inspection, because a method removed from the call stack no longer affects security. This may be harmful, e.g. when trusted code depends on the results supplied by untrusted code [22].

History-based access control has been receiving major attention as an alternative to stack inspection. Differently from stack inspection, the run-time security

state depends on (a suitable abstraction of) the *whole* execution. History-based policies and mechanisms have been studied at both levels of foundations [3, 21, 35] and of language design and implementation [1, 18]. A common drawback of all these approaches is that the security policy is a *global* invariant, that must hold at any point of the execution. This may involve guarding each resource access, and ad-hoc optimizations are then in order to recover efficiency, e.g. compiling the global policy to local checks [16, 28]. Furthermore, a large monolithic policy may be hard to understand, and not very flexible either.

Local policies [6], formalise and enhance the concept of *sandbox* [23], while being more flexible than global policies and local checks spread over program code. In the spirit of history-based security [1], local policies can inspect the whole trace of security-relevant events generated by a running program. Local policies smoothly allow for safe composition of programs with their own security requirements, and they can drive call-by-contract composition of services [9]. In mobile code scenarios, local policies can be exploited e.g. to model the interplay among clients, untrusted applets and policy providers: before running an untrusted applet, the client asks the trusted provider for a suitable policy, which will be locally enforced by the client throughout the applet execution.

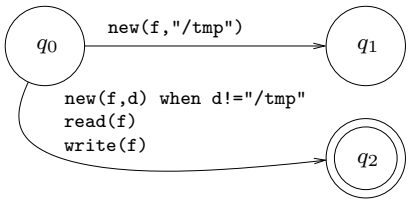
In this paper, we outline the design of an extension to the Java language, so to enhance its security mechanism with local policies. In the spirit of JML [27], policies are orthogonal to Java code and they are specified as comments. Our policies are defined through a special kind of finite state automata (FSA), where the input alphabet comprises the security-relevant events, parametrized over resources. So, policies can express any regular property on execution histories.

The first contribution of this paper is the design of a run-time mechanism for enforcing local policies in Java. Apart from the specification of policies and sandboxes, this requires no intervention by the programmer in the source code.

The second contribution is an optimization of the run-time enforcement mechanism. This is based on a static analysis that detects the policies violated by a program in some of its executions [8]. The analysis is performed in two phases. The first phase over-approximates the patterns of resource usages in a program. The second phase consists in model-checking the approximation of a program against the policies on demand. We have implemented this phase in [10], as a polynomial-time algorithm on the size of the approximation and of the policies. Summing up, we optimise the run-time security mechanism, by discarding the policies guaranteed to never fail, and by checking just the events that may lead to a violation of the other policies.

An example. Consider a trusted component `NaiveBackup` that offers static methods for backing up and recovering files. Assume that the file resource can be accessed through the following interface:

```
public File(String name, String dir);
public String read();
public void write(String text);
public String getName();
public String getDir();
```



where `new(f,d)` and `read(f)` stand for:
`f = new File(n,d)`
`f.read()`

Fig. 1. File confinement policy `file-confine(f,d)`.

The constructor takes as parameters the name of the file and the directory where is is located. A new file is created when no file with the given `name` exists in the given `dir`. The meaning of the other methods is as expected.

In the class `NaiveBackup`, the method `backup(src)` copies the file `src` into a file with the same name, located in the directory `/bkp`. The method `recover(dst)` copies the backed up data to the file `dst`. As a naïve attempt to optimise the access to backup files, the last backed up file is kept open.

```
class NaiveBackup {
  static File last;
  public static backup(File src) {
    if(src.getName() != last.getName())
      last = new File(src.getName(), "/bkp");
    last.write(src.read());
  }
  public static recover(File dst) {
    if(dst.getName() != last.getName())
      last = new File(dst.getName(), "/bkp");
    dst.write(last.read());
  }
}
```

Consider now a malicious `Plugin` class, trying to spoof `NaiveBackup` so to obtain a copy of a secret passwords file. The method `m()` of `Plugin` first creates a file called `passwd` in the directory `"/tmp"`, and then uses `NaiveBackup` to recover the content of the backed up password file (i.e. `/bkp/passwd`).

```
class Plugin {
  public void m() {
    File g = new File("passwd","/tmp");
    NaiveBackup.recover(g);
  }
}
```

To prevent from this kind of attacks, the `Plugin` is run inside a sandbox, that enforces the following policy. The sandboxed code can only read/write files it has created; moreover, it can only create files in the directory `"/tmp"`. This policy is modelled by the automaton `file-confine(f,d)` in Fig. 1. The edge from q_0 to q_1 represents creating a file `f` in the directory `"/tmp"` (the name of the file is immaterial). The edge from q_0 to q_2 labelled `read(f)` prohibits reading the

file `f` if no `new(f,d)` has occurred beforehand (the double circle means q_2 is *offending*). Similarly for the edge labelled `write(f)`. The edge from q_0 to q_2 labelled `new(f,d) when d!="/tmp"` prohibits creating a file in any directory `d` different from `"/tmp"`.

```
class Main {
  public static void main() {
    File f = new File("passwd","/etc");
    NaiveBackup.backup(f);
    PolicyPool.sandbox("file-confine", new Runnable() {
      public void run() {
        new Plugin().m();
      }});
  }
}
```

The class `Main` first backs up the passwords file through the `NaiveBackup`. Then, it runs the untrusted `Plugin` inside a sandbox enforcing the policy `file-confine`. The `Plugin` will be authorized to create the file `"/tmp/passwd"`, yet the sandbox will block it while attempting to open the file `"/bkp/passwd"` through the method `NaiveBackup.recover()`. Indeed, `Plugin` is attempting to read a file it has not created, which is prohibited by `file-confine`. Note also that any attemp to directly open/overwrite the password file will fail, because the policy only allows for opening files in the directory `"/tmp"`. Note that our sandboxing mechanism enables us to enforce security policies on the untrusted `Plugin`, without intervening in its code.

2 Local policies: specification and enforcement

We start by introducing resources, events and policies. The specification of sandboxes and of their run-time enforcement mechanism follows.

2.1 Resources and Events

We model resources $\mathtt{R}_1, \mathtt{R}_2, \ldots$ as objects, and security-relevant events as method calls. For notational convenience, we use *aliases* for events. An alias `ev` for a method signature $(\mathtt{y} : \mathtt{C}).\mathtt{m}(\mathtt{y}_1 : \mathtt{C}_1, ..., \mathtt{y}_n : \mathtt{C}_n)$ is defined as:

$$\mathtt{alias\ ev}(\mathtt{x}_1, ..., \mathtt{x}_k) = (\mathtt{y} : \mathtt{C}).\mathtt{m}(\mathtt{y}_1 : \mathtt{C}_1, ..., \mathtt{y}_n : \mathtt{C}_n)$$

where $\forall j \in 1..k : \mathtt{x}_j \in \{\mathtt{y}, \mathtt{y}_1, \ldots, \mathtt{y}_n\}$. For instance:

```
alias new(f,d) = (f:File).File(string name, string d)
alias read(f)  = (f:File).File.read()
alias write(f) = (f:File).File.write(String t)
```

means `new(f,d)` is an alias for the constructor `File(string name,string d)` of the class `File`, while `read(f)` (resp. `write(f)`) is an alias for the method `read()` (resp. `write(String t)`) of the same class. The parameter `f` is the target resource, in this case an object of type `File`. The method parameters not involved in the definition of a policy can be omitted, e.g. we simply write `alias ev = C.m(x1,...,xk)` when all the parameters are immaterial.

2.2 Policies

Usage policies (Def. 1) constrain the usage of resources to obey a regular property on the program *trace*, i.e. the sequence of method calls occurred at run-time. E.g., a file usage policy `file-usage(x)` might require that before reading or writing a file `x`, that file must have been opened, and not yet closed. A usage policy gives rise to an finite state automaton (FSA) when the formal parameters are instantiated to actual resources (see [8] for further details). These automata will be exploited in Sec. 2.4 to implement the execution monitor for usage policies.

Definition 1. Usage policies

Let $\mathtt{Ev}$ *be a set of aliases, and let* $\mathtt{Res}$ *be the set of resources. A* usage policy $\mathtt{p}(\mathtt{x}_1, \ldots, \mathtt{x}_k)$ *is a 5-tuple* $\langle S, Q, q_0, F, E \rangle$*, where:*

- S *is the input alphabet, defined as follows:*

$$S = \{\, \mathtt{ev}(\mathtt{R}_1, \ldots, \mathtt{R}_k) \mid \mathtt{ev}(\mathtt{x}_1, \ldots, \mathtt{x}_k) \in \mathtt{Ev} \text{ and } \mathtt{R}_1, \ldots, \mathtt{R}_k \in \mathtt{Res} \,\}$$

- Q *is a finite set of states,*
- $q_0 \in Q \setminus F$ *is the start state,*
- $F \subseteq Q \setminus \{q_0\}$ *is the set of final "offending" states,*
- $E \subseteq Q \times Z \times Q$ *is a finite set of labelled edges, where* Z *is defined as follows:*

$$Z = \{\, \mathtt{ev}(\mathtt{Z}_1, \ldots, \mathtt{Z}_k) \texttt{ when <cond>} \mid \mathtt{ev}(\mathtt{x}_1, \ldots, \mathtt{x}_k) \in \mathtt{Ev} \wedge \mathtt{Z}_i \in \mathtt{Res} \cup \{\mathtt{x}_i\} \,\}$$

where the condition `<cond>` *is defined with the following syntax:*

$$\texttt{<cond> ::= true | } \mathtt{Z}_i \texttt{ != Z | <cond> and <cond>} \quad (\mathtt{Z} \in \mathtt{Res} \vee \mathtt{Z} = \mathtt{x}_j)$$

Usage policies resemble non-deterministic FSA, from which they differ in two points. First, the input alphabet is infinite; second, it does not coincide with the set of labels in the transition relation. Indeed, the parameters $\mathtt{Z}_i$ in the edges of a usage policy can be of two kinds: $\mathtt{Z}_i = \mathtt{R}$ for a static resource $\mathtt{R}$, or $\mathtt{Z}_i = \mathtt{x}_i$. By binding the formal parameters $\mathtt{x}_1, \ldots, \mathtt{x}_k$ to actual resources $\mathtt{R}_1, \ldots, \mathtt{R}_k$ we obtain a FSA $\mathtt{p}(\mathtt{R}_1, \ldots, \mathtt{R}_k)$, to be used in recognizing those traces respecting the policy. Roughly, the transformation into a FSA amounts to: (i) instantiating $\mathtt{x}_i$ to $\mathtt{R}_i$, while respecting the conditions in the `when` clauses, (ii) maintaining $\mathtt{Z}_i = \mathtt{R}$ for $\mathtt{R}$ static, and (iii) adding self-loops for all the events not explicitly mentioned in the policy (see [8] for details).

A trace η *respects* a policy $\mathtt{p}(\mathtt{x}_1, \ldots, \mathtt{x}_k)$ when, for all the relevant instantiations of the formal parameters $\mathtt{x}_1, \ldots, \mathtt{x}_k$ to actual resources $\mathtt{R}_1, \ldots, \mathtt{R}_k$ in η, we

have that η is not in the language of the FSA $\mathtt{p}(\mathtt{R}_1, \ldots, \mathtt{R}_k)$ – i.e. it is not possible to reach an offending state on η. For instance, consider the following traces:

$$\begin{aligned} \eta_0 &= \texttt{new(f0,"/tmp") read(f1)} \\ \eta_1 &= \texttt{new(f0,"/tmp") read(f0)} \\ \eta_2 &= \texttt{new(f0,"/tmp") read(f0) new(f1,"/etc")} \end{aligned}$$

The trace η_0 violates the policy `file-confine`, because it drives the instantiation `file-confine(f1,"/tmp")` to the offending state q_2. The trace η_1 respects the policy, because the `read` event is performed on a newly created file `f` in the directory `"/tmp"` (recall that instantiations have a self-loop labelled `read(f)` on q_1). Instead, η_2 violates the policy, because it drives the instantiation `file-confine(f1,"/etc")` to the state q_2. Indeed, instantiating the `when` clause results in an edge labelled `new(f1,"/etc")` from q_0 to q_2.

We advocate an extension of JML [27, 15] as an instrument for the formal specification of usage policies. The following comment specifies the file confinement policy of Fig. 1. The first part introduces the needed aliases. The usage policy follows, where `states`, `start`, `final`, and `trans` stand respectively for the sets Q, q_0, F and E of Def. 1.

```
\*@ alias new(f,d) = (f:File).File(String name, string d)
  @ alias read(f)  = (f:File).File.read()
  @ alias write(f) = (f:File).File.write(String t)
  @ name:   file-confine
  @ states: q0 q1 q2
  @ start:  q0
  @ final:  q2
  @ trans:  q0 -- new(f,"/tmp") --> q1
  @         q0 -- new(f,d) --> q2 when d != "/tmp"
  @         q0 -- read(f) --> q2
  @         q0 -- write(f) --> q2
  @*/
```

Note that policies can only control methods known at static time. In the case of dynamically loaded code, where methods are only discovered at run-time, it is still possible to specify and enforce policies on statically-known methods. For instance, system resources – which are accessed through the JVM libraries only – can always be protected by policies.

2.3 Sandboxes

The programmer defines the scope of a local policy through the method `sandbox()` of the static class `PolicyPool`. This signature of `sandbox()` is:

```
public static void sandbox(String pol,Runnable c) throws SecurityException
```

The string `pol` is the name of the policy to be enforced through the execution of the code `c`. For instance:

```
PolicyPool.sandbox("file-confine", new Runnable() {
        public void run() {
            // sandboxed code
            ...
        }
    });
```

The set of policies to be checked at run-time is passed as an option to the `java` command, with the following syntax (where $p_1, \ldots, p_k$ are policy names):

`java -Dcheck=<value> class` where `value` ::= `NONE` | `ALL` | $p_1; \cdots ; p_k$

When `value = NONE`, no policy is checked at run-time; when `value = ALL`, all the policies mentioned in the program are checked; in the other case, only the policies $p_1, \ldots, p_k$ are checked. Typically, the set of policies that need to be checked at run-time will be provided by the static analysis in Sec. 3.

2.4 Run-time enforcement

The implementation of the execution monitor for local policies goes through the following steps:

- as a preprocessing step, the specification of the policies to be enforced is extracted from the source code and translated into Java code.
- a custom class loader is set up, to act as a proxy for method invocations.
- when starting the execution of a method `sandbox(p,c)`, the policy `p` is activated.
- the proxy dispatches a monitored method call to the actual class, only if the call respects all the active policies.
- when leaving a `sandbox(p,c)`, the policy `p` is deactivated.

The first step is straightforward. For the second step, we use a statically generated proxy for wrapping method calls, similarly to JavaCloak [34]. Before dispatching the call to the actual class, the proxy updates the state of the policy automata. If an active policy is violated, then the proxy throws an exception.

```
public class SecurityProxy implements InvocationHandler {
    private Object obj;
    ...
    public Object invoke(Object proxy, Method meth, Object[] args)
        throws Throwable {
        Object result;
        if(PolicyPool.check(obj, meth, args))    // monitor call
             result = meth.invoke(obj, args);    // method call
        else throw new SecurityException(meth.toString());
    }
}
```

The method `PolicyPool.check()` is the core of the enforcement mechanism. For each active usage policy, it tracks the states of all the needed instantiations. States are modelled as sets of pairs $((\mathtt{R}_1, \ldots, \mathtt{R}_k), \mathtt{q})$, where $(\mathtt{R}_1, \ldots, \mathtt{R}_k)$ is a tuple of weak references [4] to the resources upon which the policy is instantiated, and `q` is the current state of the automaton. The result of `check()` is true if and only if no policy automaton reaches an offending state. If so, the method call is authorized and forwarded to the actual class; otherwise, a `SecurityException` is thrown. For instance, consider the policy `file-confine` of Fig. 1. Assume that the policy is active when the proxy traps a call to the constructor `File("passwd","/tmp")`. The `check()` method looks up the aliases table and finds the event `new(f0,"/tmp")` associated with the constructor. Firing this event updates the states of the policy `file-confine` to:

$$\{((\mathtt{f0}, \mathtt{"/tmp"}), \mathtt{q}_1),\ ((\mathtt{f1}, \mathtt{"/tmp"}), \mathtt{q}_0)\}$$

Assume now the method `read()` is invoked on the file `f1`. The state becomes:

$$\{((\mathtt{f0}, \mathtt{"/tmp"}), \mathtt{q}_1),\ ((\mathtt{f1}, \mathtt{"/tmp"}), \mathtt{q}_2)\}$$

Since the offending state q_2 has been reached, the call to `read()` is not dispatched by the proxy, which instead throws a `SecurityException`.

3 Static analysis and optimizations

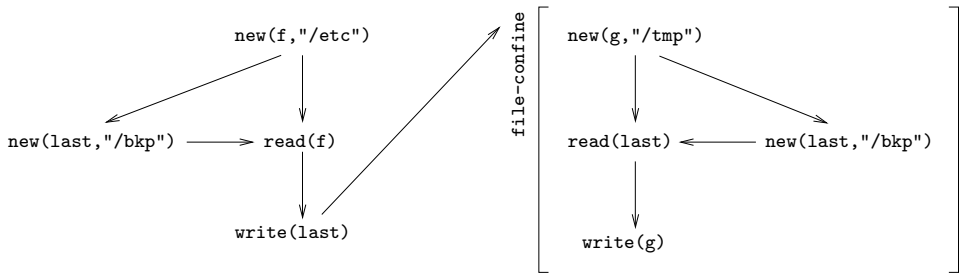
We statically analyse programs to detect those policies that are always respected in all possible executions, so to avoid checking them at run-time. For those policies that may fail, our static analysis finds the method calls that may lead to violations. This allows for optimizing the execution monitor, that will only check the program points where some security violation may actually occur.

The static analysis consists in two phases, briefly described below.

- first, we extract the program *control flow graph* (CFG), and we transform it into a *history expression*.
- then, we model-check the history expression against the usage policies enforced by the sandboxes used in the program.

The CFG of a program is a static-time data structure that represents all the possible run-time control flows. In particular, we are interested in constructing a CFG the paths of which describe the possible sequences of method calls. This construction is the basis of many interprocedural analyses, and a large amount of algorithms, with different tradeoffs between complexity and precision, have been

[4] Weak references [17] are used to avoid interference with the garbage collector. Using standard references would indeed prevent the garbage collector from disposing resources referenced by the `PolicyPool` only, so potentially leading to memory exhaustion. An object referenced only by weak references is considered unreachable, and so it may be disposed by the garbage collector.



νf.new(f,"/etc") $\cdot$ νlast.$(\varepsilon +$ new(last,"/bkp"))$\cdot$ read(f) $\cdot$ write(last) $\cdot$
file-confine$[\nu$g.new(g,"/tmp") $\cdot$ νlast.$(\varepsilon +$ new(last,"/bkp"))$\cdot$ read(last) $\cdot$ write(g)$]$

Fig. 2. CFG and history expression of the method `main()`. Sequential composition is modelled by the operator $\cdot$, while $+$ stands for non-deterministic choice. The scope of a dynamically created resource n is defined by the binder νn.

developed [24, 32]. CFGs hide most of the data flow, so approximating the actual behaviour. This approximation is *safe*, in the sense that each actual execution flow is represented by a path in the CFG. Yet, some paths may exist which do not correspond to any actual execution. A typical source of approximation is dynamic dispatching. When a program invokes a method on an object O, the run-time environment chooses among the various implementations of that method. The decision is not based on the declared type of O, but on the actual class O belongs to, which is unpredictable at static time. To be safe, CFGs over-approximate the set of methods that can be invoked at each program point.

Once a context-sensitive CFG has been extracted, it is transformed into a *history expression* [6], a sort of context-free grammar enriched with special constructs for dealing with policies and resources. To do that, we suitably adapt the classical state-elimination algorithm for FSA [14]. E.g., the CFG and the history expression associated with the `main()` of Sec. 1 are depicted in Fig. 2.

The second phase consists in model-checking history expressions against usage policies. As a first step, history expressions are transformed into Basic Process Algebras (BPAs, [13]), so to enable us to exploit standard model-checking techniques [20]. Roughly, one checks the emptiness of the pushdown automaton resulting from the conjunction of the BPA obtained in the previous step, and the negation of the policy. The transformation into BPA preserves the validity of the approximation, i.e. the traces of the BPA respect the same policies as those of the history expression. The two main issues are dealing with dynamic creation of resources (not featured by BPAs), and with redundant sandboxes, i.e. nested occurrences of the same sandbox. For the first item, we devised a sort of Skolemization of history expressions, which uses a finite number of witness resources in place of the ν-binders. For the second item, we transformed history expressions to remove the redundant sandboxes therein. Full details about our technique and our model-checking tool can be found in [8, 10].

4 Conclusions

We have presented a proof-of-concept for an extension of the security mechanism of Java. This is based on history-based local policies. These policies are naturally expressed through a sort of finite state automata, the edges of which are parametric over resources. The use of these automata is new in the context of Java. It required extending the formal model of [7] with polyadic events, that model method invocations. We have proposed a programming construct for specifying sandboxes, and designed an execution monitor for enforcing them. We have devised a static analysis that optimizes the run-time enforcement of policies. The analysis exploits call-graph construction and model-checking to predict the policies that will always be obeyed, and to single out the program points where run-time checks are needed. An implementation of our framework is currently under development; only the model-checking tool is already available [10]. It runs in polynomial time in the size of the history expression extracted from the analysed program.

Extensions. A significant improvement to our model consists in extending the language of policies by allowing for more logical operators in conditions. The expressive power can be increased by including the usage of JML boolean expressions, like e.g. the evaluation of pure methods without side effects. This would allow to directly specify policies that depend on implicit counters (e.g. no more than N kilobytes of data can be transmitted). The impact of such a refinement on the static analysis requires further investigation.

Related work. Many authors [16, 19, 28, 36] mix static and dynamic techniques to transform programs and make them obey a given policy. Our model allows for local, polyadic policies and events parameterized over dynamically created resources, while the above-mentioned papers only consider global policies and no parameterized events. Polymer [12] is a language for specifying, composing and enforcing (global) security policies, based on *edit automata* [11]. Run-time monitoring is necessary to enforce policies, while our model-checking tecnique may avoid this overhead. A typed λ-calculus with primitives for creating and accessing resources, and for defining their permitted usages, is presented in [25]. A type system guarantees that well-typed programs are resource-safe, yet no effective algorithm is given to check compliance of the inferred usages with the permitted ones. The policies of [25] can only speak about the usage of *single* resources, while ours can span over many resources, e.g. a policy requiring that no socket connections can be opened after a local file has been read. Wang, Takata and Seki [37] propose a model for history-based access control. They use control-flow graphs enriched with permissions and a primitive to check them, similarly to [5]. The run-time permissions are the intersection of the static permissions of all the nodes visited in the past. The model-checking technique can decide if all the permitted traces of the graph respect a given regular property on its nodes. Unlike our local policies, that can enforce any regular policy on traces, the technique of [37] is less general, because there is no way to enforce

a policy unless it is encoded as a suitable assignment of permissions to nodes. Pandey and Hashii [33] enhance the access control model of Java, by specifying fine-grained constraints on the execution of mobile code. A method invocation is denied when a certain condition on the dynamic state of the system is false. Since this condition may be the result of calling an arbitrary method, this mechanism is quite general, yet is has some drawbacks. First, the process of deciding if an action must be denied might not terminate. Second, the dynamic conditions in the policy might prevent from static optimizations. Our local policies and static analysis can be smoothly adapted to the case of mobile code. This extension requires analysing bytecode instead of source code when extracting usage policies and when constructing the CFG. Martinelli et al. [2, 29, 30] model security policies as process algebras. They implement a custom JVM, with an execution monitor that traps system calls and fires them concurrently to the policy. When a trapped system call is not permitted by the policy, the execution monitor tries to force a corrective event – if possible – otherwise it aborts the system call. Being interested in efficient run-time enforcement, this framework neglects static optimizations, which however might be unfeasible because of dynamic conditions in the policies. In [26] a customization of the JVM/KVM is proposed for extending the Java run-time enforcement to a wider class of security policies, mainly designed for devices with reduced computational capabilities. As before, the presented framework does not feature any static analysis. JACK [4] is a tool for the validation of Java applications, both at the levels of bytecode and of source code. Programmers specify application properties through JML annotations, which are equi-expressive with first-order logics. These annotations give rise to proof obligations, to be statically verified by a theorem prover. The verification process might require the intervention of the developer to resolve the proof obligations, while in our framework the verification is fully automated. JACK can specify history-based policies by using ghost variables spread over JML annotations to mimick the evolution of a finite-state automaton defining the policy. Our formalism allows for expressing history-based policies in a more direct and compact way, although our syntax is not pure JML. The problem of wrapping method calls has been widely studied, and several frameworks have been proposed in the last few years. Some approaches, e.g. the Kava system [38], use bytecode rewriting to obtain behavioural run-time reflection. This amounts to modifying the structure of the bytecode, by inserting additional instructions before and after a method invocation. A less invasive solution, adopted e.g. by JavaCloak [34], consists in exploiting the Java core package `java.lang.reflect`. This approach seems particularly appropriate in our framework. Indeed, we can use a custom class loader to substitute a dynamic proxy for the classes involved in the enforcement of security policies. Notably, this solution does not require custom JVMs.

Acknowledgments. This research has been partially supported by EU-FETPI Global Computing Project IST-2005-16004 Sensoria (Software Engineering for Service-Oriented Overlay Computers) and by EU-funded project IST-033817 GridTrust – Trust and Security for Next Generation Grids.

References

1. M. Abadi and C. Fournet. Access control based on execution history. In *Proc. 10th Annual Network and Distributed System Security Symposium*, 2003.
2. F. Baiardi, F. Martinelli, P. Mori, and A. Vaccarelli. Improving grid services security with fine grain policies. In *OTM Workshops*, 2004.
3. A. Banerjee and D. A. Naumann. History-based access control and secure information flow. In *Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards (CASSIS)*, 2004.
4. G. Barthe et al. JACK - a tool for validation of security and behaviour of Java applications. In *Formal Methods for Components and Objects*, 2007.
5. M. Bartoletti, P. Degano, and G. L. Ferrari. Static analysis for stack inspection. In *Proc. International Workshop on Concurrency and Coordination*, 2001.
6. M. Bartoletti, P. Degano, and G. L. Ferrari. History based access control with local policies. In *Proc. Fossacs*, 2005.
7. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In *Proc. Fossacs*, 2007.
8. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Model checking usage policies. Technical report, Dip. Informatica, Univ. Pisa, 2008.
9. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Semantics-based design for secure web services. *IEEE Transactions on Software Engineering*, 34(1), 2008.
10. M. Bartoletti and R. Zunino. LocUsT: a tool for checking usage policies. Technical Report TR-08-07, Dip. Informatica, Univ. Pisa, 2008.
11. L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security (FCS)*, 2002.
12. L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. PLDI*, 2005.
13. J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.
14. J. Brzosowski and J. E. McCluskey. Signal flow graph techniques for sequential circuit state diagrams. *IEEE Trans. on Electronic Computers*, 1963.
15. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3), 2005.
16. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000.
17. K. Donnelly, J. J. Hallett, and A. Kfoury. Formal semantics of weak references. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, 2006.
18. G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, 1999.
19. Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. 7th New Security Paradigms Workshop*, 1999.
20. J. Esparza. On the decidability of model checking for several μ-calculi and Petri nets. In *Proc. 19th Int. Colloquium on Trees in Algebra and Programming*, 1994.
21. P. W. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy*, 2004.

22. C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.
23. L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, 1999.
24. D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6), 2001.
25. A. Igarashi and N. Kobayashi. Resource usage analysis. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.
26. I. Ion, B. Dragovic, and B. Crispo. Extending the java virtual machine to enforce fine-grained security policies in mobile devices. In *ACSAC*, 2007.
27. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
28. K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proc. First Asian Programming Languages Symposium*, 2003.
29. F. Martinelli and P. Mori. Enhancing Java security with history based access control. In *FOSAD*, 2007.
30. F. Martinelli, P. Mori, and A. Vaccarelli. Towards continuous usage control on grid computational services. In *ICAS/ICNS*, 2005.
31. Microsoft Corp. *.NET Framework Developer's Guide: Securing Applications*.
32. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
33. R. Pandey and B. Hashii. Providing fine-grained access control for Java programs. In *Proc. ECOOP*, 1999.
34. K. V. Renaud. Experience with statically-generated proxies for facilitating Java runtime specialisation. *IEEE Proc. Software*, 149(6), Dec 2002.
35. C. Skalka and S. Smith. History effects and verification. In *Asian Programming Languages Symposium*, 2004.
36. P. Thiemann. Enforcing safety properties using type specialization. In *Proc. ESOP*, 2001.
37. J. Wang, Y. Takata, and H. Seki. HBAC: A model for history-based access control and its model checking. In *Proc. ESORICS*, 2006.
38. I. Welch and R. J. Stroud. Kava - using byte code rewriting to add behavioural reflection to Java. In *USENIX Conference on Object-Oriented Technology*, 2001.

Flexible Object Adaptation for Java-like Languages

Tetsuo Kamina and Tetsuo Tamai

The University of Tokyo
{kamina,tamai}@acm.org

Abstract. To realize object adaptability with a clear conceptual framework, a role model *Epsilon* was proposed. The novelty of Epsilon model is its ability to change object's behavior dynamically. However, such kind of flexibility also easily brings type-unsafety and other unreliabilities. This paper proposes a small core language ϵ that formalizes some key concepts of object adaptation, which is informally described in the Epsilon model. In ϵ, three kinds of objects, context instances, role instances, and class instances exist at run-time. A role instance is a member of a context instance where how role instances are collaborating with each other is encapsulated. A class instance can dynamically assume a role by binding itself with a role instance, and can also throw the role away. Their relationship can change during computation, and ϵ's type system assures that the computation does not go wrong, even though some exceptional cases concerning downcasting exist. This formalization clarifies the essential features of the object adaptation incorporated in Epsilon model and provides a solid base for program analysis and language processor implementation.

1 Introduction

Considerable research efforts have been devoted to make objects in object-oriented systems more flexible and adaptable. The recent interest in self-managed (or autonomic, self-healing, adaptive) systems/computing indicates renewed attention on this target.

Objects in the conventional object-oriented languages are created from fixed templates defined as classes and once they are created, it is hard to change their structures and behaviors dynamically. One way of enabling dynamic changes to objects is to fully employ the mechanism of meta-programming or reflection and allow free transformation of objects at run-time. Some languages such as Ruby[27] provide dynamic object structure change capability as their innate feature. The obvious problem with such a feature and meta-programming in general is performance decline. But even if performance is somehow ensured at a certain level, taking advantage of sophisticated optimizing techniques, there will still remain the problem of programming difficulties and error-proneness.

In [25], Tamai et al. proposed a role model *Epsilon* and a language based on the model *EpsilonJ* (the revised version is also available in [26]). The aim of

this model was to realize object adaptability with a clear conceptual framework. A collaboration field called environment or context can be defined by a set of roles that interact with each other. An object can dynamically participate in a collaboration field and assume one of its roles so that it acquires functions of the role and capability of collaborating with other roles in the field. An object may assume multiple roles of different collaboration fields at a time so that it grows into a complex object with rich functions but its behavior can be clearly comprehensible from the base behavioral properties of roles.

However, such kind of flexibility also easily brings type-unsafety and other unreliabilities. To avoid such unreliabilities, constructs of Epsilon should formally be defined. In [25] or [26], however, only a brief description (described by examples) on the semantics of the language EpsilonJ was provided. A full language specification is released on the Web[24] but its description is informal. In fact, even though there are many formal studies on collaboration-based design, relatively few efforts have been made on formalizing object adaptation.

In this paper, we propose a small core calculus ϵ that formalizes some key concepts of object adaptation, described in [25]. In ϵ, three kinds of objects, context instances that represent collaboration fields, role instances, and class instances exist at run-time. A role instance is a member of a context instance where collaboration between role instances is encapsulated. A class instance can dynamically assume a role by binding itself with a role instance, and can also throw the role away. Their relationship can be changed during computation, and ϵ's type system assures that the computation does not go wrong, even though some exceptional cases concerning downcasting exist.

Even though ϵ is quite similar to EpsilonJ, it is not designed to be a small subset of EpsilonJ. Instead, the aim of this work is to understand the essences of object adaptation; therefore, there are some differences between EpsilonJ and ϵ. In short, ϵ puts more emphasis on stating clear language semantics, while EpsilonJ provides much liberal ways for writing programs. Nevertheless, this formalization clarifies the essential features of the object adaptation mechanism incorporated in EpsilonJ and provides a solid base for program analysis and language processor implementation.

2 An Overview of Object Adaptation

To make this paper self-contained, we briefly summarize the main features of object adaptation that are formerly described in [25] as Epsilon model. In this section, we informally describe these features by using Java-like syntax.

In the Epsilon model, three kinds of objects, context instances, role instances, and class instances, exist at run-time. A context instance is a collaboration field where role instances interact with each other. A role instance is a member of a context instance, and there may be multiple role instances associated with a context instance. A set (or a sequence) of role instances associated with a context instance is called a *role group*. Behind the scenes, contexts are augmented by an internal field representing the role group. A class instance is the same as in

```
context Company {
  role Employer {
    void pay() {
      Employee.getPaid();} }
  role Employee {
    int save, salary;
    Employee(int salary) {
      this.salary = salary;}
    void getPaid() {
      save += salary; } } }
```

```
class Person {
  int money; }

Person tanaka = new Person();
Person komiyama = new Person();
Company todai = new Company();
todai.Employer.newBind(komiyama);
todai.Employee.newBind(tanaka,1000);
((Company.Employer)komiyama).pay();
```

Fig. 1. Declaration of the context `Company` and object adaptation.

conventional Java-like languages, except that it can dynamically participate in a collaboration field (represented by a context instance) by assuming one of its role instances so that it acquires functions of the role instance and capability of collaborating with other role instances. Behind the scenes, classes are also augmented by an internal field representing the set of assuming roles.

How a context is declared is demonstrated in Fig.1. A context is declared using context declaration that begins with the keyword `context` followed by the name of context (that is `Company` in Fig.1). A Role is declared as a member of the context using role declaration that begins with the keyword `role`. In Fig.1, two roles, `Employer` and `Employee`, are declared. Each `context` and `role` is declared with fields, methods and constructors just like classes.

A role group is associated with the enclosing context instance and referred by the role name. We can access each instance of role group by using an iterator that iterates over the role group. In Fig.1, however, we use a more convenient syntactic sugar; we can apply a method declared in the role to the whole role group, and the method is invoked for all the role instances. Thus, the method call `Employee.getPaid()` is interpreted as calling the method `getPaid` of all the `Employee`'s instances.

In Epsilon model, contexts and roles are the first class constructs at runtime as well as at model description time. A context is instantiated by the `new` expression; the expression `new Company()` creates an instance of `Company`. A role instance is created by a special operation `newBind` that performs two things; (1) it creates a role instance that is a member of the receiver context; and (2) it binds the role instance with the class instance provided as an argument of `newBind`. In Fig.1, the instances of `Person`, `komiyama` and `tanaka`, assume the roles `todai.Employer` and `todai.Employee`, respectively. Note that the second and the following arguments, if any, of `newBind` call are arguments for constructor call of roles. After the binding, the object bound to the role acquires an access to the role instance and thus can use the role methods. Furthermore, it acquires type of the role, and the role methods are invoked through type-casting. In fact, the type of expression `(Company.Employer)komiyama` is a mixin composition[17] `Company.Employer::Person`, where `::` is a composition operator, thus we can safely access the `pay()` method declared in `Company.Employer`. Whether

`komiyama` can be cast to `Company.Employer` or not (i.e. whether `komiyama` assumes a role instance of `Company.Employer` or not) is checked at run-time.

The binding of a class instance and a role instance may be dissolved at run-time; for this purpose, any role implicitly declares `unbind` method. For example, the following piece of code

```
((Company.Employee)tanaka).unbind();
```

firstly casts `tanaka` to its role `Company.Employee`, then calls `unbind`. After calling `unbind`, the connection between `tanaka` and `Company.Employee` is dissolved, and the dissolved role instance becomes garbage. Instead, another class instance may assume the unbound role instance if we use the `swap` method, which is also implicitly declared in a role. For example, the following code

```
Person sato = new Person();
((Company.Employee)tanaka).swap(sato);
```

results in that `sato` takes over `tanaka`'s `Company.Employee` role.

There should be some interaction between a class instance and a role instance that are bound together so that the state and the behavior of the class instance should be affected by the binding. For this purpose, there is a way of defining an interface to a role and this is used at the time of binding with a class instance, requiring the class instance to supply the interface. This interface is declared with the `requires` phrase as follows.

```
role Employee requires { void deposit(int); } {
  void sendSalary(int salary) { deposit(salary); } }
```

When a required interface is declared to a role, methods can be imported to the role from the class instance. For example, suppose the class `Person` has a method `deposit`:

```
class Person { ... int money;
  void deposit(int s) { money+=s; } }
```

After the binding of `todai.Employee.newBind(tanaka)` in the previous program piece, the method `deposit(int)` of `tanaka` is imported to the `Employee` role instance through the interface. When an interface method is overridden by the corresponding role method, the replacing method of the binding object becomes hidden. If there is a need for invoking the hidden method in the context, either in the body of the overriding method or in other role (or context) methods, it is possible to invoke it by attaching the qualifier `super` to the method name.

Since the role instance requires the class instance to supply the `requires` interface, the class has to implement it. Note that the `requires` interface may be anonymous, just as shown in the above program. In other words, the class has to be a *structural subtype* of the `requires` interface. A similar mechanism is also found in [17].

$$
\begin{array}{rl}
A & ::= C \mid X \\
T & ::= A \mid X.R \mid X.R :: C \\
T_S & ::= T \mid \{\ \bar{M}_I\ \} \\
L_C & ::= \texttt{class}\ C\ \lhd\ D\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \} \\
L_R & ::= \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M}_I\ \}\{\bar{T}\ \bar{f};\ \bar{M}\} \\
L_X & ::= \texttt{context}\ X\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{L}_R\ \} \\
M & ::= T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \\
M_I & ::= T\ m(\bar{T}\ \bar{x}); \\
r & ::= e_0.R(\bar{e}) \\
e & ::= x \mid e.f \mid e.m(\bar{e}) \mid (\texttt{new}\ C(\bar{e}), \bar{r}) \mid \texttt{new}\ X(\bar{e}) \mid e_0.R.\texttt{newBind}(e, \bar{d}) \mid \\
 & \quad e_0.\texttt{unbind}() \mid e_0.\texttt{swap}(e) \mid (X.R)e \\
v & ::= (\texttt{new}\ C(\bar{v}), \bar{r}) \mid \texttt{new}\ X(\bar{v}) \mid (X.R)(\texttt{new}\ C(\bar{v}), \bar{r})
\end{array}
$$

Fig. 2. Abstract syntax

Finally, we note that a class instance may assume multiple roles; for example, a person can be a customer, a patient, and an employee depending on the context. Current context of the person may change through downcasting. Furthermore, a class instance may change roles even within a context; for example, a person can change its role from an employee to an employer, which is possible because a class instance may discard a role and assume another role dynamically. By using the `swap` operation, the state of the old employer (e.g., unfinished tasks, responsibilities, and so on) is taken over by the new employer.

3 ϵ: the core calculus of Epsilon

This section provides a small core calculus ϵ of Epsilon model. This formalization is based on FJ[16], a minimum core language of Java, but includes some additional features found in the full Java language such as `super` calls that are needed to model important features of object adaptation.

Syntax. The abstract syntax of ϵ is shown in Fig.2. In this paper, the metavariable A ranges over class or context names; S, T, and U range over named types; T_S ranges over types (including `requires` interface); C, D and E range over class names; R ranges over role names; X ranges over context names; L_C ranges over class declarations; L_R ranges over role declarations; L_X ranges over context declarations; f and g range over field names; M and N range over method declarations; M_I ranges over interface method declarations; m ranges over method names; b, c, d and e range over expressions; x ranges over variables; r and s range over role instances; v and w range over values.

We put an over-line for a possibly empty sequence. Furthermore, we abbreviate pairs of sequences in a similar way, writing "$\bar{T}\ \bar{f}$" as a shorthand for "$T_1\ f_1, \cdots, T_n\ f_n$", where n is the length of $\bar{T}$ and $\bar{f}$, and so on (the same way introduced in [16]).

Object adaptation can be realized with some imperative features. However, imperative features will introduce complexity in the type system. We can take another approach to concentrate on the new features incorporated in Epsilon model. We design ϵ as a purely functional calculus; i.e. the state of context instance never changes after the constructor invocation (in this paper, constructor declarations are abbreviated and constructor invocations become implicit). To model the dynamic semantics of object adaptation, we regard each class instance as a pair of a constructor invocation and a sequence of roles that the class instance is bound with. Therefore, ϵ is considered as a runtime expression language, since the programmer does not write $(\texttt{new}\ C(\bar{e}), \bar{r})$ but only $\texttt{new}\ C(\bar{e})$, which is identical to $(\texttt{new}\ C(\bar{e}), \cdot)$. The role instances $\bar{r}$ are generated during the evaluation and needed in the rules to check and maintain the roles of the class instance.

In ϵ, there are two kinds of types: named types and interface types. A named type is represented by a class name, a context name, a role name prefixed by a context name, or a mixin composition in the form of $X.R :: C$. These types may appear in field declarations, formal parameter types and return types. On the other hand, interface types, denoted by $\{\ \bar{M}_I\ \}$, may appear only in the `requires` clause.

As in FJ, we assume that the set of variables includes the special variable `this`, which is considered to be implicitly bound in every method declaration. Furthermore, we also assume that the set of variables includes the special variable `super`, which is considered to be implicitly bound in every role method declaration.

For the reduction and typing rules, we need a few auxiliary definitions, given in Fig.3. We write $m \notin \bar{M}$ to mean that the method definition of the name m is not included in $\bar{M}$. The fields of type T, written $\mathit{fields}(T)$, is a sequence $\bar{T}\ \bar{f}$ pairing the type of each fields with its name. The type of method m in type T_S, written $\mathit{mtype}(m, T_S)$, is a pair, written $\bar{T} \to T_0$, of a sequence of formal parameter types $\bar{T}$ and its return type T_0. If T is a role type $X.R$ and m is not found in $X.R$, its `requires` interface is searched. If T is a mixin composition, the left operand of `::` is searched first. Similarly, the body of method m in type T, written $\mathit{mbody}(m, T)$, is a pair, written $(\bar{x}, e)$, of a sequence of formal parameters $\bar{x}$ and an expression e.

We also present a rule that checks whether a role can be bound to a class instance or not. The following predicate *bindable* is used for this checking. An instance of role $X.R$ can be bound to an instance of class C if C is a subtype of $X.R$'s required interface $\{\ \bar{M}_I\ \}$.

Subtyping rules of ϵ are shown in Fig.4. Subtyping is a reflexive and transitive closure induced by the subclassing relation. Furthermore, a class is a subtype of an interface if the class implements all the methods declared in the interface. This subtyping rule is used in checking whether a role can be bound to a class or not. There also exists some straightforward subtyping rules regarding mixin composition.

Field lookup:

$$\frac{\texttt{class}\ C\ \lhd\ D\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}\qquad \mathit{fields}(D) = \bar{S}\ \bar{g}}{\mathit{fields}(C) = \bar{S}\ \bar{g}, \bar{T}\ \bar{f}}$$

$$\frac{\texttt{context}\ X\ \{\ \bar{T}\ \bar{f};\ \bar{N}\ \bar{L_R}\}\qquad \texttt{role}\ R\ \texttt{requires}\ \{\ \cdots\ \}\{\ \bar{S}\ \bar{g};\ \cdots\} \in \bar{L_R}}{\mathit{fields}(X.R) = \bar{S}\ \bar{g}}$$

$$\frac{\texttt{context}\ X\ \{\ \bar{T}\ \bar{f};\ \bar{N}\ \bar{L_R}\ \}}{\mathit{fields}(X) = \bar{T}\ \bar{f}}$$

$$\frac{\mathit{fields}(X.R) = \bar{S}\ \bar{g}\qquad \mathit{fields}(C) = \bar{T}\ \bar{f}}{\mathit{fields}(X.R :: C) = \bar{T}\ \bar{f}; \bar{S}\ \bar{g}}$$

$$\frac{\mathit{fields}(T) = \bar{T}\ \bar{f}}{\mathit{ftype}(f_i, T) = T_i}$$

Method body lookup:

$$\frac{\texttt{class}\ C\ \lhd\ D\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}\qquad T\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mbody}(m, C) = (\bar{x}, e)}$$

$$\frac{\texttt{class}\ C\ \lhd\ D\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}\qquad m \notin \bar{M}}{\mathit{mbody}(m, C) = \mathit{mbody}(m, D)}$$

$$\frac{\texttt{context}\ X\ \{\ \cdots \bar{M}\ \bar{L_R}\ \}\qquad T\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mbody}(m, X) = (\bar{x}, e)}$$

$$\frac{\texttt{context}\ X\ \{\ \cdots \bar{L_R}\ \}\qquad \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M_I}\ \}\{\ \cdots\ \bar{M}\ \} \in \bar{L_R}\qquad T\ m(\bar{S}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mbody}(m, X.R :: C) = (\bar{x}, e)}$$

$$\frac{\texttt{context}\ X\ \{\ \cdots \bar{L_R}\ \}\qquad m \notin \bar{M}\qquad \texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M_I}\ \}\{\ \cdots\ \bar{M}\ \} \in \bar{L_R}}{\mathit{mbody}(m, X.R :: C) = \mathit{mbody}(m, C)}$$

Method type lookup:

$$\frac{\texttt{class}\ C\ \lhd\ D\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}\qquad T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mtype}(m, C) = \bar{T} \to T}$$

$$\frac{\texttt{class}\ C \lhd D\ \{\bar{T}\ \bar{f};\ \bar{M}\}\qquad m \notin \bar{M}}{\mathit{mtype}(m, C) = \mathit{mtype}(m, D)}$$

$$\frac{\texttt{context}\ X\ \{\ \bar{T}\ \bar{f};\ \bar{N}\ \bar{L_R}\ \}\qquad T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}\qquad \texttt{role}\ R\ \texttt{requires}\ \{\bar{M_I}\}\{\cdots\ \bar{M}\} \in \bar{L_R}}{\mathit{mtype}(m, X.R) = \bar{T} \to T}$$

$$\frac{\texttt{context}\ X\ \{\bar{T}\ \bar{f}; \bar{N}\ \bar{L_R}\}\qquad m \notin \bar{M}\qquad \texttt{role}\ R\ \texttt{requires}\ \{\bar{M_I}\}\{\cdots\ \bar{M}\} \in \bar{L_R}}{\mathit{mtype}(m, X.R) = \mathit{mtype}(m, \{\bar{M_I}\})}$$

$$\frac{T\ m(\bar{T}\ \bar{x});\in \bar{M_I}}{\mathit{mtype}(m, \{\bar{M_I}\}) = \bar{T} \to T}$$

$$\frac{\mathit{mtype}(m, X.R) = \bar{T} \to T}{\mathit{mtype}(m, X.R :: C) = \bar{T} \to T}$$

$$\frac{\mathit{mtype}(m, X.R) \text{ is undefined}}{\mathit{mtype}(m, X.R :: C) = \mathit{mtype}(m, C)}$$

$$\frac{\texttt{context}\ X\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \bar{L_R}\ \}\qquad T\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e;\ \} \in \bar{M}}{\mathit{mtype}(m, X) = \bar{T} \to T}$$

Binding check:

$$\frac{C\ \texttt{<:}\ \{\bar{M_I}\}\qquad \texttt{context}\ X\ \{\cdots\ \bar{L_R}\}\qquad \texttt{role}\ R\ \texttt{requires}\ \{\bar{M_I}\}\{\ \cdots\ \} \in \bar{L_R}}{\mathit{bindable}(X.R, C)}$$

Fig. 3. Auxiliary definitions

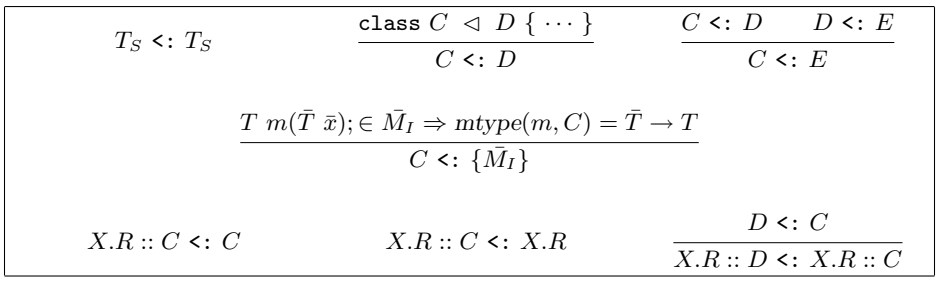


Fig. 4. Subtyping rules

An ϵ program is a pair (CT, e) of a *class table* CT and an expression e. A class table is a map from class names and context names to class declarations and context declarations, respectively. The expression e may be considered as the `main` method of the "real" program. The class table is assumed to satisfy the following conditions: (1) $CT(C) =$ `class` $C\ \cdots$ for every $C \in dom(CT)$; (2) $CT(X) =$ `context` $X\ \cdots$ for every $X \in dom(CT)$; (3) all roles R in $CT(X)$ are uniquely named; (4) $T \in dom(CT)$ for every class name, context name, and role name appearing in $range(CT)$.

Dynamic semantics. The reduction rules of ϵ are shown in Fig.5. The reduction relation is of the form $e \longrightarrow e'$, read "expression e reduces to expression e' in one step." We write $\longrightarrow^*$ for the reflective and transitive closure of $\longrightarrow$.

There are two rules for field access (as in FJ, we assume that all the field names are distinct); one is field access to a class instance or context instance (the rule R-FIELD)[1], and the other is field access to a role instance through type casting (the rule R-RFIELD). Note that R-RFIELD shows that a field access to a role instance reduces to the corresponding actual argument for the role constructor. During the computation, a class instance has to retain the *states* of the role instances which the class instance is bound with, which is why we formulate a class instance as a pair of a class constructor invocation and a sequence of role instances.

Similarly, there are two rules for method invocation. The method invocation reduces to the expression of the method body, substituting all the parameters $\bar{x}$ with the argument expressions $\bar{e}$ and the special variable `this` with the receiver of method invocation. The rule R-RINVK shows the case of role method invocation, where the variable `super` is also substituted with the receiver of method invocation (removing type casting). The rule R-BIND shows that a `newBind` expression takes a class instance as an argument, creates a role instance, and binds it with the argument class instance. The rule R-UNBIND shows that an `unbind` expression reduces to the receiver class instance of `unbind`, removing the designated role (by $\bar{r} - r$, we mean the role r is removed from the sequence $\bar{r}$). The rule R-SWAP shows that a `swap` expression takes a class instance as an argument

[1] We use `new` $X(\bar{e})$ and (`new` $X(\bar{e}), \cdot$) interchangeably.

$$\frac{\mathit{fields}(A) = \bar{T}\ \bar{f}}{(\texttt{new}\ A(\bar{e}), \bar{r}).f_i \longrightarrow e_i} \quad \text{(R-FIELD)}$$

$$\frac{\mathit{fields}(X.R :: C) = \bar{T}\ \bar{f} \qquad (\texttt{new}\ X(\bar{d})).R(\bar{e}) \in \bar{r} \qquad \bar{b}, \bar{e} = \bar{c}}{((X.R)(\texttt{new}\ C(\bar{b}), \bar{r})).f_i \longrightarrow c_i} \quad \text{(R-RFIELD)}$$

$$\frac{\mathit{mbody}(m, A) = (\bar{x}, e_0)}{(\texttt{new}\ A(\bar{e}), \bar{r}).m(\bar{d}) \longrightarrow [\bar{d}/\bar{x}, (\texttt{new}\ A(\bar{e}), \bar{r})/\texttt{this}]e_0} \quad \text{(R-INVK)}$$

$$\frac{\mathit{mbody}(m, X.R :: C) = (\bar{x}, e_0) \qquad (\texttt{new}\ X(\bar{d})).R(\bar{c}) \in \bar{r}}{\begin{array}{c}((X.R)(\texttt{new}\ C(\bar{e}), \bar{r})).m(\bar{d}) \longrightarrow \\ [\bar{d}/\bar{x}, (X.R)(\texttt{new}\ C(\bar{e}), \bar{r})/\texttt{this}, (\texttt{new}\ C(\bar{e}), \bar{r})/\texttt{super}]e_0\end{array}} \quad \text{(R-RINVK)}$$

$$\frac{(\texttt{new}\ X(\bar{b})).R(\bar{d}) \notin \bar{r}}{\begin{array}{c}(\texttt{new}\ X(\bar{b})).R.\texttt{newBind}((\texttt{new}\ C(\bar{e}), \bar{r}), \bar{d}) \longrightarrow \\ (\texttt{new}\ C(\bar{e}), \bar{r}(\texttt{new}\ X(\bar{b})).R(\bar{d}))\end{array}} \quad \text{(R-BIND)}$$

$$\frac{(\texttt{new}\ X(\bar{e})).R(\bar{d}) \in \bar{r}}{\begin{array}{c}((X.R)(\texttt{new}\ C(\bar{c}), \bar{r})).\texttt{unbind}() \longrightarrow \\ (\texttt{new}\ C(\bar{c}), \bar{r} - (\texttt{new}\ X(\bar{e})).R(\bar{d}))\end{array}} \quad \text{(R-UNBIND)}$$

$$\frac{(\texttt{new}\ X(\bar{e})).R(\bar{e'}) \in \bar{r}}{\begin{array}{c}((X.R)(\texttt{new}\ C(\bar{c}), \bar{r})).\texttt{swap}((\texttt{new}\ D(\bar{d}), \bar{s})) \\ \longrightarrow (\texttt{new}\ D(\bar{d}), \bar{s}(\texttt{new}\ X(\bar{e})).R(\bar{e'}))\end{array}} \quad \text{(R-SWAP)}$$

Fig. 5. Reduction rules

and binds it with the designated role, removing the class instance that is the receiver of `swap` from the context instance where the designated role resides.

Reduction rules may be applied to any subexpressions of an expression, so we also need the obvious congruence rules, which are omitted in this paper.

Typing. The typing rules for ϵ expressions are shown in Fig.6. An environment Γ is a finite mapping from variables to types, written $\bar{x} : \bar{T}$. The typing judgment for expressions has the form $\Gamma \vdash e : T$, read "in the environment Γ, expression e has type T."

The rules are syntax directed, with one rule for each form of expressions. The typing rules for method invocations and constructors check that each actual parameter has a type of the corresponding formal parameter. The rule T-INVK checks that the type of receiver of method invocation may be an interface type, thus method call to `super` is allowed. The rule T-NEW checks that all the role instances with which the class instance binds are also well-typed; i.e., the context

Expression typing:

$$\frac{\Gamma \vdash e_0 : S \qquad \mathit{ftype}(f, S) = T}{\Gamma \vdash e_0.f : T} \text{ (T-FIELD)}$$

$$\Gamma \vdash x : \Gamma(x) \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_0 : T_S \qquad \Gamma \vdash \bar{e} : \bar{S} \qquad \mathit{mtype}(m, T_S) = \bar{T} \to T \qquad \bar{S} <: \bar{T}}{\Gamma \vdash e_0.m(\bar{e}) : T} \text{ (T-INVK)}$$

$$\frac{\Gamma \vdash e : X.R :: C}{\Gamma \vdash e.\texttt{unbind}() : C} \text{ (T-UNBIND)}$$

$$\frac{\begin{array}{c}\mathit{fields}(C) = \bar{T}\ \bar{f} \\ \Gamma \vdash \bar{e} : \bar{S} \qquad \bar{S} <: \bar{T} \\ \text{for } r_j \in \bar{r}\ r_j = (\texttt{new}\ X_j(\bar{c}_j)).R_j(\bar{d}_j) \\ \Gamma \vdash \texttt{new}\ X_j(\bar{c}_j) : X_j \\ \mathit{fields}(X.R) = \bar{T}_j\ \bar{g}_j \\ \Gamma \vdash \bar{d}_j : \bar{S}_j \qquad \bar{S}_j <: \bar{T}_j\end{array}}{\Gamma \vdash (\texttt{new}\ C(\bar{e}), \bar{r}) : C} \text{ (T-NEW)}$$

$$\frac{\Gamma \vdash e : X.R :: C \qquad \Gamma \vdash d : D \qquad \mathit{bindable}(X.R, D)}{\Gamma \vdash e.\texttt{swap}(d) : D} \text{ (T-SWAP)}$$

$$\frac{\mathit{fields}(X) = \bar{T}\ \bar{f} \qquad \Gamma \vdash \bar{e} : \bar{S} \qquad \bar{S} <: \bar{T}}{\Gamma \vdash \texttt{new}\ X(\bar{e}) : X} \text{ (T-CNEW)}$$

$$\frac{\Gamma \vdash e : C \qquad \mathit{bindable}(X.R, C) \qquad \Gamma \vdash e_0 : X \qquad \bar{S} <: \bar{T} \qquad \mathit{fields}(X.R) = \bar{T}\ \bar{f} \qquad \Gamma \vdash \bar{d} : \bar{S}}{\Gamma \vdash e_0.R.\texttt{newBind}(e, \bar{d}) : C} \text{ (T-BIND)}$$

$$\frac{\Gamma \vdash e : C}{\Gamma \vdash (X.R)e : X.R :: C} \text{ (T-CAST)}$$

Wellformed definitions:

$$\frac{\bar{x} : \bar{T}, \texttt{this} : C \vdash e_0 : T_0 \qquad \texttt{class}\ C \lhd D\{ \cdots \}}{T_0\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e_0;\ \}\ \texttt{OK IN}\ C} \text{ (T-METHOD)}$$

$$\frac{\bar{M}\ \texttt{OK IN}\ X.R}{\texttt{role}\ R\ \texttt{requires}\ \{\ \bar{M}_I\ \}\{\ \bar{T}\ \bar{f};\ \bar{M}\ \}\ \texttt{OK IN}\ X} \text{ (T-ROLE)}$$

$$\frac{\bar{M}\ \texttt{OK IN}\ C}{\texttt{class}\ C \lhd D\{\ \bar{T}\ \bar{f};\ \bar{M}\}\ \texttt{OK}} \text{ (T-CLASS)}$$

$$\frac{\bar{x} : \bar{T}, \texttt{this} : X \vdash e_0 : T_0 \qquad \texttt{context}\ X\ \{ \cdots \}}{T_0\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e_0;\ \}\ \texttt{OK IN}\ X} \text{ (T-XMETHOD)}$$

$$\frac{\bar{x} : \bar{T}, \texttt{this} : X.R, \texttt{super} : \{\ \bar{M}_I\ \} \vdash e_0 : T_0 \qquad \texttt{context}\ X\ \{ \cdots\ \bar{L_R}\} \qquad \texttt{role}\ R\ \{\ \bar{M}_I\ \}\{ \cdots \} \in \bar{L_R}}{T_0\ m(\bar{T}\ \bar{x})\{\ \texttt{return}\ e_0;\ \}\ \texttt{OK IN}\ X.R} \text{ (T-RMETHOD)}$$

$$\frac{\bar{M}\ \texttt{OK IN}\ X \qquad \bar{L_R}\ \texttt{OK IN}\ X}{\texttt{context}\ X\ \{\ \bar{T}\ \bar{f}; \bar{M}\ \bar{L_R}\}\ \texttt{OK}} \text{ (T-CONTEXT)}$$

Fig. 6. Typing rules

instance of each role is well-typed, and each actual parameters of each role constructor has a type of the corresponding formal parameter. The rules T-BIND and T-SWAP check that the receiver role and argument class instance of `newBind` and `swap`, respectively, are compatible.

The type system assures that the receiver of `newBind` is a context, and the type of receiver of `unbind` and `swap` is a mixin composition of a role type and a class type, i.e., only (role) type casting expressions can be a receiver of `unbind` and `swap` operations.

Finally, we show the typing rules for method declarations, class declarations, context declarations, and role declarations. The rules for wellformed definitions are also shown in Fig.6. The type of the body of a method declaration is a subtype of the return type. The special variable `this` is bound in every method declaration, and for every method declaration in roles, a variable `super` is also bound. A class declaration is wellformed if all the methods declared in that class are wellformed. A role declaration is wellformed if all the methods declared in that role are wellformed. A context declaration is wellformed if all the methods and roles declared in that context are wellformed.

Finally, we show the properties of ϵ, which is every well-typed expression evaluates to a value or an expression containing casts, `newBind`, `unbind`, or `swap` that cannot be reduced further.

Theorem 1 (Subject Reduction). *If $\Gamma \vdash e : T$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : T'$ for some T' <: T.*

Theorem 2 (Progress). *If $\emptyset \vdash e : T$ and e is neither (1) a value, (2) an expression containing $(X.R)(\texttt{new}\ C(\bar{e}), \bar{r})$ where $(\texttt{new}\ X(\bar{b})).R(\bar{d}) \notin \bar{r}$ for some $\bar{b}, \bar{d}$, (3) an expression containing $e_0.R.\texttt{newBind}((\texttt{new}\ C(\bar{e}), \bar{r}))$ where $e_0.R(\bar{d}) \in \bar{r}$ for some $\bar{d}$, (4) an expression containing $((X.R)(\texttt{new}\ C(\bar{e}), \bar{r})).\texttt{unbind}()$ where $(\texttt{new}\ X(\bar{b})).R(\bar{d}) \notin \bar{r}$ for some $\bar{b}, \bar{d}$, and (5) an expression containing $((X.R)(\texttt{new}\ C(\bar{e}), \bar{r})).\texttt{swap}((\texttt{new}\ D(\bar{c}), \bar{s}))$ where either $(\texttt{new}\ X(\bar{b})).R(\bar{d}) \in \bar{s}$ or $(\texttt{new}\ X(\bar{b})).R(\bar{d}) \notin \bar{r}$ for some $\bar{b}, \bar{d}$, then $e \longrightarrow e'$ for some e'.*

Remark. Because of ϵ's ability to assume and discard roles at run time, the progress theorem shows that there are some unreliabilities associated to adaptable objects. The result shows that we have to accept the possibility that access to role's fields or methods or `newBind`/`unbind`/`swap` operations can fail at run time. In the full language, of course, such a failure does not stop whole the program; it generates an exception that can be caught by a surrounding exception handler.

One may consider that a more satisfactory type soundness result would be obtained by changing the typing rules. For example, we may change the rule T-NEW to make the result type be $(C, \bar{r})$ so that the T-BIND rule can check that in the type $(C, \bar{r})$ of `newBind`'s first argument e, there are no instance of $X.R$ in $\bar{r}$. However, it is hard to make this approach cooperate with other constructs such as `if` statements. With imperative features, some dynamic checking should be necessary. Even in purely functional languages, there may be a situation where

we want to change which role the class instance is bound with according to the condition of `if` expression, but putting emphasis on type-safety prevents providing such flexibility. On the other hand, EpsilonJ's way of thinking is to provide convenient idioms such as downcasting that most conventional languages support, to make programmers flexibly bind roles and objects at their own risk. To our knowledge there are no pieces of work on object adaptation that carefully inspect the semantics of downcasting.

4 Discussions and Related Work

While developing ϵ, we found some significant differences between EpsilonJ and ϵ. Firstly, in ϵ every role instance binds with a class instance, and role instance methods and fields can be accessed only through type casting. EpsilonJ does not hold this property and we can write unsafe programs by explicitly accessing role instances. Another contribution of this work w.r.t. EpsilonJ is that it provides solid information for language processor implementation. For example, current implementation of EpsilonJ is based on reflective APIs, which results in significant performance degradation [26]. On the other hand, ϵ indicates that we can employ a more "natural" way to implement the language; e.g. a class may have a field that contains a set of role instances with which the class instance binds. Furthermore, type casting is realized as an operation that selects a role instance from that set, and `super` calls are modeled as delegations. Indeed, we have developed an EpsilonJ translator to Java based on this idea [20].

The programming language powerJava[3] is a quite similar language with EpsilonJ, in that roles and collaboration fields are the first class constructs, interaction between roles are encapsulated, and objects can participate in the interaction by assuming one of its roles. As in ϵ, the type of role depends on the enclosing context instance. However, powerJava lacks the feature of role groups that is a powerful mechanism of getting role instances associated with the context instance reflectively. Furthermore, no formalization is given for powerJava.

Delegation Layers[21] and Object Teams[14] provides more flexible object based composition of collaborations. For example, Delegation Layers combines the mechanism of delegation[18, 22] and virtual classes[19, 7], or Family Polymorphism[10]; roles may be represented by virtual classes, and composition is instance-base using delegation mechanism. Both of these approaches, however, do not successfully represent object adaptation described in this paper. For example, in ϵ the object after assuming a role may dynamically throw the role away, and even the thrown role may be assumed by another object and states held in the role instance are taken over by the latter object.

There are pieces of literature that formalize the feature of extending objects at run-time. Ghelli presented foundations for extensible objects with roles based on Abadi-Cardelli's object calculi[1], where coexistence of different methods introduced by incompatible extensions is considered [12]. Gianantonio et al. presented a calculus $\lambda Obj+$[13], an extension of λObj[11] with a type assignment system that allows self-inflicted object extension still statically catching

the "message not found" errors. Drossopoulou et al. proposed a type-safe core language *Fickle*[9] that allows re-classification of objects, a mechanism of dynamically changing object's belonging classes which share the same "root" superclass. On the other hand, ϵ focuses on a foundation of object adaptation for Java-like languages (based on FJ) and the feature of assuming roles that are thrown by other objects (by `swap` operation).

Mixins[6, 2, 17] are similar to roles in EpsilonJ in that mixins form partial definitions that can be reused with a number of classes that conform the *requirements* of mixins. Even though mixin composition is originally performed at compile time, dynamic composition of mixins is also studied in a core calculus[4], and such kind of object level inheritance is also studied as *wrappers*[8, 5]. Dynamic trait (a stateless mixin) substitution is also studied in [23]. All of these pieces of work put more emphasis on type-safety, while ϵ supports more sophisticated mechanism such as the `swap` operation and object level downcasting to roles.

EpsilonJ supports context-oriented programming (COP)[15] in that contexts (layers in COP terms) are named first-class entities that can be referred to explicitly at run-time, and context-dependent object behavior can be changed by activating/deactivating contexts from anywhere in the code. In EpsilonJ, such activation/deactivation is performed by type casting.

5 Concluding Remarks

This paper reports a minimum core calculus of Epsilon model that has the notable feature of representing object adaptation. The calculus ϵ provides a precise, formal definition of all key essential features of Epsilon model. Its type system assures that the computation does not go wrong, even though some exceptional cases concerning downcasting exist. The formalization clarifies the essential features of object adaptation and provides solid information for program analysis and language processor implementation. For example, ϵ suggests a natural way to implement EpsilonJ, which has been partly achieved by the latest implementation.

References

1. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
2. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam – designing a Java extension with mixins. *ACM TOPLAS*, 25(5):641–712, 2003.
3. M. Baldoni, G. Boella, and L. van der Torre. Interaction between objects in powerJava. *Journal of Object Technology*, 6(2):5–30, 2007.
4. Lorenzo Bettini, Viviana Bono, and Silvia Likavec. Safe and flexible objects with subtyping. *Journal of Object Technology*, 4(10):5–29, 2005.
5. Lorenzo Bettini, Sara Capecchi, and Elena Giachino. Weatherweight Wrap Java. In *SAC'07*, pages 1094–1100, 2007.
6. G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA 1990*, pages 303–311, 1990.

7. Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP'98*, volume 1445 of *LNCS*, pages 523–549, 1998.
8. Martin Buchi and Wolfgang Weck. Generic wrappers. In *ECOOP 2000*, volume 1850 of *LNCS*, pages 201–225, 2000.
9. Sophia Drossopoulou, Ferruccio Damiani, and Mariangiola Dezani-Ciancaglini. Fickle: Dynamic object re-classification. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 130–149, 2001.
10. Eric Ernst. Family polymorphism. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 303–327, 2001.
11. K. Fisher, F. Honsell, and J.C̃. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.
12. Giorgio Ghelli. Foundations for extensible objects with roles. *Information and Computation*, (175):50–75, 2002.
13. Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A lambda calculus of objects with self-inflicted extension. In *OOPSLA'98*, pages 166–178, 1998.
14. Stephan Hermann. Object Teams: Improving modularity for crosscutting collaborations. In *Net Object Days 2002*, volume 2591 of *LNCS*, 2002.
15. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.
16. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
17. Tetsuo Kamina and Tetsuo Tamai. McJava – a design and implementation of Java with mixin-types. In *2nd ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *LNCS*, pages 398–414. Springer, 2004.
18. Gunter Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP'99*, volume 1628 of *LNCS*, pages 351–366, 1999.
19. Ole Lehrmann Madsen and Birger Moller-Pdersen. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA'89*, pages 397–406, 1989.
20. S. Monpratarnchai and T. Tamai. The design and implementation of a role based language, EpsilonJ. In *Proc. ECTI-CON 2008*, pages 37–40, 2008.
21. Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP 2002*, volume 2374 of *LNCS*, pages 89–110, 2002.
22. Klaus Ostermann and Mira Mezini. Object-oriented composition untangled. In *OOPSLA'01*, pages 283–299, 2001.
23. Charles Smith and Sophia Drossopoulou. *Chai*: Traits for Java-like languages. In *ECOOP 2005*, volume 3586 of *LNCS*, pages 453–478, 2005.
24. T. Tamai. The language specification of EpsilonJ. http://www.graco.c.u-tokyo.ac.jp/t̃amai/pub/epsilon/spec.txt.
25. Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. An adaptive object model with dynamic role binding. In *International Conference on Software Engineering (ICSE 2005)*, pages 166–175, 2005.
26. Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. Objects as actors assuming roles in the environment. In *Software Engineering for Multi-Agent Systems V*, volume 4408 of *LNCS*, pages 185–203, 2007.
27. D. Thomas and A. Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.

Dealing with Numeric Fields in Termination Analysis of Java-like Languages *

Elvira Albert[1], Puri Arenas[1], Samir Genaim[2], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] CLIP, Technical University of Madrid (UPM), Spain

Abstract. Termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Though several tools exist which are able to prove termination of non-trivial programs, when one tries to apply them to realistic programs, there are still a number of open problems. In the case of Java-like languages, one of such problems is to find a practical solution to prove termination when the termination behaviour of loops is affected by *numeric fields*. We have performed statistics on the Java libraries to see how often this happens in practice and we found that in 12.95% of cases, the number of iterations of loops (and therefore termination) explicitly depends on values stored in fields and, in the vast majority of cases, such fields are numeric. Inspired by the examples found in the libraries, this paper identifies a series of difficulties that need to be solved in order to deal with numeric fields in termination and propose some ideas towards a lightweight analysis which is able to prove termination of sequential Java-like programs in the presence of numeric fields.

1 Termination Analysis and Numeric Fields

Termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Termination analysis is about the study of *loops*, which are the program constructs which may introduce non-termination. Loops may correspond to iterative constructs or to recursion. The boolean conditions which determine whether the loop should be executed again or not are called *guards*. Automated techniques for proving termination are typically based on analyses which track *size* information, such as the value of numeric data or array indexes, or the size of data structures. In particular, analysis should keep track of how the (size of the) data involved in loop guards changes when the loop goes through its iterations. This information is used for specifying a *ranking function* for the loop [14], which is a function which strictly decreases on a

* This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.

well-founded domain at each iteration of the loop, thus guaranteeing that the loop will be executed a finite number of times.

In the last two decades, a variety of sophisticated termination analysis tools have been developed. Several analyses and tools exist, primarily for less-widely used programming languages, including term rewrite systems [8], and logic and functional languages [11, 6, 10]. Termination-proving techniques are also emerging in the imperative paradigm [5, 7, 8], even for dealing with large industrial code [7].

Termination analysis of realistic object-oriented programming languages faces new difficulties due to the existence of advanced features such as exceptions, virtual method invocation, references, heap-allocated data-structures, objects, fields. Focusing on Java, termination analyzers for Java bytecode programs [1] and for Java source [9] are being developed which are able to accurately handle a good number of the features mentioned above. However, interesting open problems still remain. In particular, it is well known that the *heap* poses important difficulties to static analysis. Some reasons for this are that the heap is a global data structure whose contents are not accessed using named variables, but rather using (possibly chained) references. Therefore, the same location in the heap may be modified using different aliased references and, furthermore, references may be reassigned several times, and thus they may point to different locations during execution. When loop guards involve information stored in the heap, such as object *fields*, tracking size information becomes rather complex and accurate aliasing information is required in order to track all possible updates of the corresponding fields (see e.g. [12]).

A partial solution to this problem is already solved by the *path-length* domain [9] which allows proving termination of loops which traverse acyclic heap-allocated data structures (i.e., linked lists, trees, etc.). Path-length is an abstract domain which, for reference values, provides a safe approximation of the length of the longest reference chain reachable from it. Unfortunately, though the path-length domain is a useful abstraction for fields which contain references, it does not capture any information about fields which contain numbers. In this work we look into the Sun implementation of the Java libraries for J2SE 1.4.2 in order to estimate how often loop termination depends on *numeric* values stored in fields and to try to come up with sufficient conditions for termination which are able to cover a large fraction of those loops whose termination is not provable using current techniques, such as those in [9, 1].

2 Motivating Examples from the Java Libraries

Since termination is an undecidable problem, all techniques for proving termination provide sufficient (but not necessary) conditions for termination. Therefore, for any termination proving technique it is possible to find terminating programs where the given technique fails to prove termination. Thus, usually the practicality of termination analyses is measured by applying the analyses to a representative set of real programs. In this work, the design of the analysis is

driven by common programming patterns for loops that we have found in the Java libraries. By looking at Sun's implementation of the J2SE (version 1.4.2_13) libraries, which contain 71432 methods, we have found 7886 loops (for, while, and do) from which 1021 (12.95%) explicitly involve fields in their guards. By inspecting these 1021 loops, we have observed, among others, the following three kinds of common patterns in the Java libraries.

Pattern #1: Loops in this category use numeric fields as bounds for loop counters and, moreover, the value of those fields is not updated within the loop. This is demonstrated in the following loop of the method public void or(BitSet set) of library java.util.BitSet, where unitsInUse is a field of type int:

```
for(; i<set.unitsInUse; i++) bits[i]=set.bits[i];
```

Pattern #2: Loops in this category are similar to those in the previous category. The difference is that, rather than corresponding to the value of a numeric field, the bound of the loop counter corresponds to the length of an array which is stored in a field. In this case, even if the elements of the array may be updated within the loop, if the field itself does not, the length of the array remains constant. This is demonstrated in the following example, corresponding to method public void fixupVariables(java.util.Vector vars, int globalsSize) of library org. apache.xpath.functions.FunctionMultiArgs where m_args is a field of type Expression[ ]:

```
for(int i=0; i<m_args.length; i++) m_args[i].fixupVariables(vars,globalsSize);
```

Pattern #3: Loops in this category use numeric fields as loop counters, which means that the field value is updated within the loop, but none of the references in the *path* to the field (in this example, the chain just consists of the reference this) are re-assigned within the loop, i.e., all updates correspond to the same object on the heap. This is demonstrated in the following loop of the method public synchronized void setLength(int newLength) in the library java.lang.StringBuffer, in which count is a field of type int:

```
for(; count<newLength; count++) value[count] = '\0';
```

In this paper we concentrate on proving termination of loops that fall in the above categories by providing (uniform) conditions under which proving termination of such loops becomes possible. The Java libraries include also other patterns such as loops that: (1) increase/decrease an integer variable until it reaches a given upper/lower bound; (2) traverse a non-cyclical data structure or an array; (3) look for an element in an input stream, which is common in classes that manipulate structured text such as parsing XML documents; and (4) look for a non-null element in a given array in a circular way, which is very common in the multi-threading classes. The first two patterns are the major part of the loops, and they are already handled in [1]. The other patterns are planned for future research and are not addressed in this paper.

3 Dealing with Fields in Termination

In a Java-like language, objects are stored in the *heap* and they are accessed by means of references (or pointers). References can take the value null or *point* to an object in the heap. Given a reference l which points to an object o, $l.f$ denotes the value of the field f in the object o. We say that a syntactic construction of the form $l.f$ is a *field access*. Each field f has a unique signature, which consists of the class where it is declared, its type, and its name.

Objects are global in that they survive the execution of methods. Typically, when a method starts execution, a large number of objects may exist in the heap. One approach to analyzing programs with objects is to compute an abstraction of the heap (see [13]) which approximates the execution context of each method. This usually requires computing abstractions of all possible objects in the program, which might turn out to be too expensive in practice if one wants to deal with real programs. However, in most cases, only a small fraction of such objects affects the execution of the method. We seek for a more lightweight approach which tries to approximate the contents of only a subset of the objects in the heap. The approach must remain correct by making safe assumptions about the objects (and fields) whose contents are not taken into consideration.

Another disadvantage of computing an abstraction of the heap, in addition to its computational complexity, is that we end up obtaining termination information which is *context-dependent*. Though context dependent analysis is in principle more precise, the results obtained are not extrapolable to other execution contexts. In particular, in the case of libraries, ideally we would like to prove termination in a *context-independent* way, i.e., regardless of what the contents of the heap are when the method is executed.

We now introduce the concept of *local* field access. In particular, we are interested in finding field accesses which are local to a *loop*. Though termination analysis in our context aims at proving termination of methods, in the rest of the paper we will concentrate on *loops* since they are the main subject of termination analysis.

Definition 1 (local field access). *We say that a field access $l.r_1.\ldots.r_n.f$, where f is a numeric field, is* local *to a loop L if*

(i) No prefix of $l.r_1.\ldots.r_n$ changes its value within L, i.e., they remain constant.
(ii) If the value of $l.r_1.\ldots.r_n.f$ changes within L, then all write accesses have to be done explicitly through the field access $l.r_1.\ldots.r_n.f$.

Condition (i) guarantees that all occurrences of the field access within the loop refer to the same memory location in the heap. Note that the prefixes of $l.r_1.\ldots.r_n$, i.e., l, $l.r_1$, $l.r_1.r_2$, ... are references which altogether form a chain to an object where the numeric field f is stored. Condition (ii) guarantees that all write accesses to the field can be syntactically identified. Note that this condition can be violated due to aliasing, since we can have different field access which update the same memory location.

Given a loop L, we denote by *g-fields(L)* the set of field accesses $l.r_1 \ldots r_n.f$, where f is a numeric field, which explicitly appear inside the guard of L. For instance, for the three loops in Section 2, the sets *g-fields(L)* are, respectively, {this.unitsInUse}, {m_args.length} and {this.count}. These three fields are locally accessed within their corresponding loops. The practical implication is: if we ensure that a field in *g-fields*(L) is local, then we are able to treat this field in the same way as if it were a local variable, as regards the analysis of L. Essentially, given a loop L, the analysis proceeds as follows:

1. Compute the set *g-fields*(L).
2. Compute the set *l-g-fields*(L), which is the subset of *g-fields*(L) which contains the field accesses whose locality condition has been proved.
3. Analyze the termination of L by considering those field accesses in *l-g-fields*(L) as if they were local variables.

The method is applied locally to all nested loops in L. Note that the termination of a method is ensured if all loops *involved* in its body are terminating. By involved we mean not only those loops occurring explicitly in the body but also those coming from possible calls to some other methods.

3.1 Syntactic Inference of the Locality Condition on Field Accesses

The above approach is practical only if we provide effective mechanisms to prove the locality condition on field accesses. In this section, we consider only loops that do not contain method invocations. Later, in Section 4, we take method invocations into account. Now, we present sufficient *syntactic* conditions for ensuring that a field access is local. The following conditions ensure that a numeric field access $l.r_1.\ldots.r_n.f$ is local to a loop L:

1. The reference variable l remains constant in L. This can be ensured by checking that there is no assignment to l within L.
2. All reference fields $l.r_1, \ldots, l.r_1 ... r_n$ are constant in L. This can be ensured by checking that there is no assignment within L to a field with the same signature as any of r_i.
3. All assignments to a field with the same signature as f in L are done through the field access $l.r_1.\ldots.r_n.f$.

Let us briefly explain each of the above conditions. Conditions 1 and 2 ensure point (i) of Definition 1. The reason why we separate it into two conditions is due to the way in which it is syntactically checked in each case. For the reference variable l, we check that there is no assignment to it. These conditions guarantee that we do not incorrectly consider a loop of the form *while (l.size < 10) {l.size++; l=new C(); }* as terminating. Note that this loop is not guaranteed to terminate since l potentially changes the location of *size* and hence its value.

Condition 2 guarantees that we do not change any of the intermediary reference fields $l.r_1, \ldots, l.r_1 ... r_n$. Note that if we modify a reference field $l.r_1 ... r_i$ then

we fail to ensure constancy of the local field access. For instance, we would fail to prove termination of this loop *while (l.r$_1$.size < 10)* {*l.r$_1$.size++; l'.r$_1$=z;* }. This is a safe assumption, as without knowledge about the aliasing of l and l', we might be changing the reference to *size*.

Condition 3 is a sufficient condition to ensure that the field is not updated due to possible aliasing with another object (point (*ii*) in Definition 1). This condition is not satisfied in a loop of the form *while (l.size < 10)* {*l.size++; l'.size--;* } and therefore we do not prove termination for it. This is reasonable, as l and l' might be aliased during the execution.

Example 1. Reconsider the third loop in Section 2. For clarity, we replace the access to the field count to explicitly include the this path variable:

```
for(; this.count<newLength; this.count++) value[this.count] = '\0';
```

We can prove that this.count is local to the loop by checking the syntactic conditions stated above: the reference this does not change; and all updates to this.count are done through the field access this.count. The key point is that, since this.count is local, we can safely treat it as local variable. Consequently, existing termination analysers [3] are able to infer that this.count is increasing at each iteration. Besides, as newLength remains constant in the loop, the analyzer finds out that newLength-this.count is a decreasing well-founded measure and thus termination is guaranteed. □

4 Termination with (Virtual) Method Invocations

In this section, we address the more challenging problem of proving the termination of loops which contain method invocations. As notation, we denote by $M(L)$ the set of methods transitively invoked within the scope of a loop L. We now study what are the conditions that the methods in $M(L)$ must satisfy in order to preserve the locality condition on *g-fields*(L).

Consider a method m invoked within L, we distinguish three possible scenarios. In the first two ones, the implementation of m is available at analysis time and thus we can apply the techniques to detect local field accesses to the code in m. As our method is purely syntactic, in order to check the conditions on m, first we must do a *renaming* between the variables in the call and the formal parameters in m, as parameter passing does. Note that, when a method m is invoked from a reference l, the *this* reference in m is renamed to l in order to check the conditions. In the first scenario, method m does not modify the value of the (numeric) field, whereas in the second one it does. In the third one, the implementation of m either it is not available (i.e., it is an abstract or native method) or it has been redefined by means of subclassing. We aim at proving *modular* termination of the loop by making assumptions on m. We study these scenarios in more detail below.

Scenario 1. Consider method test$_1$ at the top of the right-hand column in Fig. 1. Due to dynamic dispatching, the execution of a.m$_1$() can correspond to method

```
class A {
  int f,g;
  int m1(){return 1;}
};
abstract class B extends A {
  int m1(){return 2;}
  void m2(){
    f = f + 1;
  }
  abstract void m3();
};
class C extends B {
  void m3() { g=g-1; }
};
```

```
void test1(A a,int k) {
  while (a.f < k) a.f = a.f + a.m1();
}
void test2(B b,int k) {
  while (b.f < k) b.m2();
}
void test3(B a,int k) {
  while (a.f < k){
     a.m3();
     a.f = a.f + a.m1();}
}
```

Fig. 1. Termination with fields and method invocations

m_1 in class A or to method m_1 in class B. Since, in both cases, the reference variable a remains constant and the field a.f is not updated within either implementation of m_1, we can guarantee that the field access a.f is local to (the loop in) test_1. Proving termination now is straightforward since both implementations of m_1 return a positive number.

Scenario 2. Now, we consider the case that, even if the field access is local to the loop, the field is updated during the execution of the invoked method. This happens, for example, in method test_2 where the call $\mathsf{b.m}_2()$ increments the value of b.f. Indeed, method m_2 is responsible for the termination of test_2. In this case, we need to track the variations in the field b.f in an inter-procedural manner. One way to do it is by *inlining* the invoked method. However, this cannot always be done, as it is problematic for recursive methods. Another approach is to transform the methods in such a way that they carry as additional parameters the fields that must be tracked. When we have virtual invocations and several instances of the same method can be executed at runtime, we need to do such transformation to all the possible instances. Doing it at the level of Java would require a more sophisticated transformation, since parameters are passed by value. It could, however, be easily integrated in a termination analyzer like [1], as it works on an intermediate representation with permits multiple output parameters. We plan to develop this part in an extended version of this work.

Scenario 3. If the code of a method m in $M(L)$ is not available or the implementation of the method has been redefined, unfortunately we can say very little about the termination of L. For instance, if m is an abstract method, it is customary that the user defines a new class which implements m and it is always possible that it modifies the fields which affect the termination of the loop. Also, the new implementation might introduce callbacks which endanger termination. Clearly, one possibility is, once the implementation is available, to re-analyze

the loop with the new method. More interestingly, we can try to prove *modular* termination of the loop by assuming that (1) the method terminates, (2) it does not update any field access in *g-fields* and (3) it does not have callbacks. Once the new implementation is available, we actually have to ensure that the method m does not introduce a termination problem in L by checking the first two syntactic conditions in Sect. 3.1 as well as proving termination of m by applying our method to m again. For instance, consider method test_3, which is similar to test_1, but where a call to the (abstract) method m_3 has been added in the body of the loop. Assume that the class C is not available, then we make the assumption that m_3 is terminating and does not update a.f. Under these assumptions, we can prove modular termination of the loop. Consider now that the user defines class C at the bottom. Trivially, this method terminates and besides we can ensure that a.f is never updated from it. Note that, if the update inside m_3 was on f instead of on g, we would fail to ensure that that m_3 does not interfere with the guard. Indeed, the loop does not terminate in this case.

4.1 Method Invocations in the Java Libraries

It is common to find loops for scenarios 1 and 3 in the Java libraries. For instance, the loop of *Pattern* #2 of Section 2 is an example of scenario 3. The method fixupVariables invoked by m_args[i] is an abstract method of the library org.apache.xpath.Expression. The code is not available, thus we can only aim at proving termination modularly. We first make the assumption that fixupVariables will not introduce a termination problem in the loop. Under this assumption, we can prove termination of the loop. Note that, for actual implementation of fixupVariables, we will have to check that the local access condition holds and that it terminates.

We found many loops for scenario 1. For instance, the following loop appears in method public int indexOf(Object elem) of the library java.util.ArrayList:

```
for (int i = 0;i<size;i++)
    if (elem.equals(elementData[i])) return i;
```

where size is a field of type int. Its termination depends on the termination of the calls to elem.equals(elementData[i]), where elem and elementData[i] are objects of class java.lang.Object. The implementation of equals is available and contains as unique instruction return (this==obj), which ensures the local field access of size. Thus the loop is definitely terminating. It is rare in the libraries to find loops for scenario 2, indeed we have not found any. Though we believe it is necessary to provide solutions for them in order to handle the termination of user-defined programs which rely on the libraries and define methods which actually update the fields.

It is important to note that the solution we have proposed for this scenario is valid as long as the implementation of the missing methods does not use static fields. The reason for this is that static fields can be, similarly to global variables, used in the code without being passed as arguments to the method. Therefore, the set of classes reachable from a method signature, as obtained by

the procedure above, is not guaranteed to be a safe approximation of the actual classes reached by execution in the presence of static fields.

5 Perspectives for Future Work

The state of the practice in termination analysis is moving beyond less-widely used programming languages to realistic *object-oriented* languages. This paper draws attention to some difficulties that need to be solved if object *fields* are to be supported by termination analyzers. In particular, tracking size information becomes rather complex, and accurate *aliasing* information is required in order to track all possible updates of the corresponding fields. Motivated by examples found in the Java libraries, we have proposed some ideas towards dealing with numeric fields in a practical manner. The perspectives on the application of our technique include to infer *termination annotations* for as many methods in the Java libraries as possible. Applying termination tools on realistic programs which use libraries is a challenging problem, as there are many dependencies between the library classes and, in our experience, even small applications require analyzing a high number of library methods. By using precomputed annotations, the analyzer can safely assume the termination of those annotated methods in the Java libraries (and those that they depend upon).[3]

Although our ideas have not been experimentally evaluated yet, we believe that most of the patterns found in the libraries match those presented in Sec. 2. We, nevertheless, plan to improve the accuracy of the analysis in order to cover a broader range of patterns. For instance, as a starting point, we have proposed to check the local field access condition on those fields which appear explicitly in the guards, denoted *g-fields*. There are, of course, other possibilities and enhancements:

- Ideally, we should try to prove the locality condition not only on *g-fields*, but also on those fields which may interact with *g-fields*. For instance, in a loop of the form *while (l.size < 10)* {*l.size+= l'.size;* }, unless we track some information about $l'.size$ (in this case, its sign would suffice), we will fail to prove termination. Unfortunately, it is not always trivial to determine the minimal set of fields which may interact with *g-fields*. In particular, a simple syntactic inspection is not enough.
- To simplify the above point, another idea would be to try and prove the locality condition on *all* fields which appear inside the scope of the loop. This approach would be in general more accurate (e.g., would solve the above problem) but more expensive. Importantly, even if not all fields are local to the loop, the termination analysis proceeds (step 3 in Sect. 3). As long as the non-local field accesses do not affect the termination behaviour, the analysis can still succeed to prove termination.

[3] Note that precomputed assertions are valid as long as the user does not redefine methods which have been used (and analysed) to infer the assertions.

– Another interesting refinement is to consider not only the fields which appear explicit in the guards but also those which are accessed through *getter* methods like *while (l.getSize() < 10)* {...}. For this, we should go through the code of the methods invoked in the loop guards and identify those fields. A simple solution to this problem is inlining the method. Afterwards, the same basic techniques explained in the paper could be applied.

It can be seen that in some cases there is an accuracy vs efficiency tradeoff and also that, what it is optimal for one example might not be good for others. We need to perform experimental evaluation to assess the different options.

From an implementation perspective, we plan to enhance the COSTA system [3] with the ideas presented in this paper. COSTA is a cost and termination analyzer which works directly on the bytecode (and has no knowledge about the source Java). The termination module is based on the techniques proposed in [1] and the cost module on the method described in [2]. To carry out the implementation, the first issue is to incorporate the syntactic conditions to prove whether fields are accessed locally. Condition 3 can be easily checked on the bytecode by seeing that there is no `putfield` to the corresponding field signatures. Checking that the object does not change (conditions 1 and 2) requires to track dependencies between stack variables and local variables. This happens because, in the bytecode, the access to a field is done by first pushing the variable (on which the condition is to be checked) to the stack and then the field is accessed from the stack variable. This check can be done syntactically in most cases due to the elimination of stack variables [4]. Once the syntactic conditions are checked, we will implement the extensions to treat fields as local variables during analysis. This is straightforward to do in COSTA, as the tool converts the bytecode into a rule-based representation where the local variables (and the stack positions) appear as arguments of these rules. We can just add the required fields as additional arguments to them. Size analysis will directly treat them as it does with local variables in order to infer how they increase/decrease over the program.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *Proc. of FMOODS*, LNCS. Springer-Verlag, 2008. To appear.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *ESOP'07*, LNCS, 2007.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: A Cost and Termination Analyzer for Java Bytecode. In *Proc. of BYTECODE Workshop*, ENTCS. Elsevier, 2008. To appear.
4. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In *Proc. SAC*. ACM Press, 2008.
5. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
6. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *J. Log. Program.*, 41(1):103–123, 1999.

7. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
8. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *IJCAR*, 2006.
9. P. Hill, E. Payet, and F. Spoto. Path-length analysis of object-oriented programs. In *Proc. EAAI*. Elsevier, 2006.
10. C. Lee, N. Jones, and A. Ben-Amram. The size-change principle for program termination. In *Proc. POPL*. ACM, 2001.
11. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In *ICLP*, 1997.
12. C. Marché and C. Paulin-Mohring. Reasoning about java programs with aliasing and frame conditions. In J. Hurd and T. F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2005.
13. A. Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In M. J. Irwin and K. D. Bosschere, editors, *LCTES*, pages 54–63. ACM, 2006.
14. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.

Fixing the Java Module System, in Theory and in Practice

Rok Strniša

Computer Laboratory, University of Cambridge
Rok.Strnisa@cl.cam.ac.uk
http://www.cl.cam.ac.uk/~rs456/

Abstract. The proposed Java Module System (JAM) has two major deficiencies, as noted in our previous work: (a) its unintuitive class resolution can often give unexpected results; and (b) only a single instance of each module is permitted, which forces sharing of data and types, and so makes it difficult to reason about module invariants. Since JAM will be a part of Java 7, solving these problems before its release would benefit the majority of Java developers and users.

In this paper, we propose modest changes to the module language, and to the semantics of the class resolution, which together allow the module system to handle more scenarios in a clean and predictable manner. To develop confidence, both theoretical and practical, in our proposals, we: (a) formalise the improved module system, iJAM; (b) prove mechanized type soundness results; and, (c) give a proof-of-concept implementation that closely follows the formalisation; these are in Ott, Isabelle/HOL, and Java, respectively.

The formalisation is itself modular: iJAM is based on our previous formalization of JAM (LJAM), which extends Lightweight Java (LJ). LJ has shown to be a good base language, allowing a high reuse of the definitions and proof scripts, which made it possible to carry out this development relatively quickly, on the timescale of the language evolution process.

1 Introduction

Currently, Java supports only a very limited form of component-level information hiding and reuse. Two Java Community Processes, JSR-277 [1] and JSR-294 [2], are developing the Java Module System (JAM), which will be a part of Java 7.

In our previous work [3], we analysed, partly developed, and fully formalized the core part of the proposed module system, producing the Lightweight Java Module System (LJAM) [4]. We also discussed the key design decisions related to the development of the core of JAM, and their alternatives, pointed out several deficiencies with module instantiation and class resolution, and gave tentative proposals for solving these problems. In particular, the two main problems are:

- JAM's unintuitive class resolution can easily lead to unexpected results.
- JAM has highly inflexible module instantiations that makes it difficult to reason about module invariants.

Our provisional ideas included reversing the class resolution algorithm, and allowing each repository to hold module instances of any visible module definition.

In this paper, we develop and precisely define clean solutions that make class resolution intuitive and flexible (through class renaming and an adapted resolution algorithm), and allow users to control the sharing of module instances (through user-specified policies). The solutions are modelled on top of LJAM, producing improved JAM (iJAM). We give a precise definition of iJAM's syntax, type system, and operational semantics. Furthermore, we prove in Isabelle/HOL [5] type soundness theorems for Lightweight Java (LJ) [6] (the language LJAM is based on), for LJAM and for iJAM. As a proof of concept, we also implement a module system on top of Java, which can follow the semantics of either LJAM or iJAM. More specifically, our contributions are:

- precise and clean solutions to problems previously identified with JAM (§3);
- formalization of the solutions, producing iJAM (§4);
- Isabelle/HOL type soundness proofs for LJ, LJAM, and iJAM (§5); and
- an implementation that can model both LJAM and iJAM (except for sharing through renamed classes, which requires a small change to the JVM) (§7).

The details of our semantics, proofs, and implementation are available online [6, 4, 7].

2 A Short Overview of the Java Module System

The Java Module System (JAM) is a proposal for a module system integrated into Java, aimed at solving the hierarchical information hiding problem and the DLL/JAR-hell problem.

JAM introduces a few new concepts to Java, the most important of which is the *module*, or *superpackage*. A JAM module encapsulates Java packages, making even public *members* (classes or interfaces) of these packages by default invisible outside the module. Specific public members can be made visible by explicitly *exporting* them. A module can *import* other modules, which allows its members to see the (recursively) exported public members of the (recursively) imported modules.

JAM modules are specified by module developers in *module files*. The user syntax (the abstract syntax for what the user writes) of a module file is the following:

$$\mathbf{superpackage}\ mn\ \{\ \overline{\mathbf{member}pn};\ \overline{\mathbf{import}m};\ \overline{\mathbf{export}fqn};\ \}$$

Here pn is a package name (which can include dots), m and mn are module names, and fqn is a fully-qualified class name. The overbars indicate lists.

A module file is compiled into a *module definition*, which contains class files for its members, and the information specified in the module file. A module definition can be installed into a *repository*, and then instantiated to create a *module instance* (intended to be implemented with a classloader [8]), which is linked up with instances of module definitions it imports. The directed graph of module instances is contained in a structure called the *module hierarchy* (MH).

A repository is used for storing, finding, (un-)installing, and instantiating module definitions. These *actions* are performed by system administrators and passively by the module system itself. Repositories can be composed into a *repository hierarchy* to further control the visibility between modules.

3 The Key Problems and Their Solutions

In this section, we analyse JAM's two key problems: unintuitive class resolution, and inflexible module instantiation. For each, we show what they are, the reasons for corresponding design choices, their implications in practice, and how they can be fixed.

3.1 Unintuitive Class Resolution

The first key problem with JAM concerns its definition of class resolution. The procedure (recursively) searches the imported module definitions (following the order in the module file) before the client module. This is done in order to prevent anyone from overriding the core library classes,[1] and to promote the sharing of static data and types. In JAM, due to an oversimplified relation between modules, the importing module has no control over which exported classes of the imported modules are visible. The combination of the two properties lead to poor support for module interface evolution.

Suppose, for example, that the developers of a module, *XMLParser*, release an update, which makes some new functionality available through a new (and exported) class, `ParserX`. Because the new *XMLParser* module is compatible with the old one, the developers only change its micro version number, which means that the modules that previously imported the old version will now likely import the new one automatically (due to commonly-used flexible version constraints). However, if an importing module, e.g. *XSLT*, already contained a class named "`ParserX`",[2] then any reference to a class `ParserX` in *XSLT* will now incorrectly resolve to `ParserX` in *XMLParser*.

Any changes to the underlying language are highly undesirable due to various compatibility issues. Because of this, we cannot introduce proper namespaces to the source language, which would allow class references such as `XMLParser::ParserX`.

In our previous work, we suggested inverting the class resolution algorithm. However, with that approach the user can override the core library classes. To prevent this, and to obtain a more intuitive semantics, the solution is to search the core library module, then the module itself, and finally the imported modules (recursively). In the exceptional cases where more classes should not be overriden, one can imagine requiring highest administrator privileges to put these classes into the core library module.

Continuing with the above scenario, suppose the developers of `XSLT` (which contains its own `ParserX`) now want to use the new functionality given by the new *XMLParser*'s `ParserX`. In JAM, this is not possible without name refactoring in the members of *XSLT*. To prevent such refactoring, which would likely disrupt the development in a large project, we could use the well-established solution of module-boundary renaming. That is, allow specific classes to be renamed locally as they are imported. We propose boundary renaming for JAM that will allow, for example, the developer of *XSLT* to use *XMLParser*'s `ParserX` under the name `ImportedParser` using:

import XMLParser **with** ParserX **as** ImportedParser;

Similarly, one can disambiguate classes coming from different imported modules.

[1] The *core module* (Java's core library classes) is logically the root of the import graph.

[2] One might argue that due to the class naming conventions this scenario is unlikely, but one also has to realize that the naming conventions arose *because* of the lack of namespace control.

3.2 Inflexible Module Instantiation

In JAM, each module definition can only have a *single* module instance, i.e. JAM's module generators are severely restricted. This means that all clients necessarily have to share the module's static data and types, which is considered desirable, because it saves space, and prevents many possible class casting exceptions. However, suppose we have two module definitions, *XSLT* and *ServletEngine*, both of which depend on a third one, *XMLParser*. Furthermore, suppose that *XSLT* and *ServletEngine*: (i) rely on conflicting invariants of the internal state of *XMLParser*; or (ii) must run concurrently to achieve a high throughput, but *XMLParser* does not guarantee correct operation in such a concurrent environment.

In JAM, the only solution available to us in case (i) is to make the two invariants somehow compatible. In case (ii), we need to rewrite *XMLParser*'s code (assuming we have access), in order to add sufficient locking to handle concurrent accesses from multiple users. Both alternatives are often time-consuming and error-prone tasks, but are necessary if *XSLT* and *ServletEngine* have to share data or types through *XMLParser*.

However, the sharing of data or types through a common import is infrequent, especially across different programs. In all such cases, we can replicate the common import as required. This way the users can maintain conflicting invariants on separate instances of a module definition, since they use independent static data. We can also avoid unneeded contention in the imported module definition, allowing all importers to execute static methods in parallel without worrying about breaking each other's invariants.

The fundamental point here is that we should give module developers a choice of whether they want a shared or a new instance of an imported module. Here, we give an informal overview of the alternatives that allow for any possible sharing scenario:[3]

IMPORT OPTION	SHORT DESCRIPTION
import m	Uses JAM's default sharing policy. Can be overriden by **replicating** — see below.
import shared m	Explicitly requests a shared instance of m.
import own m	Requests a separate instance of m.
import m **as** amn	Requests an instance, which is shared under name amn.[4]

However, there are cases where the developer of a module would want to specify module's own *replicating policy*. If they know that a module is not concurrency-safe, then they would tag it with **replicating** and so prevent sharing; if they wanted to track some system-wide information, they would tag it with **singleton** and so force sharing:

ANNOTATION	SHORT DESCRIPTION
(no annotation)	Instantiation depends solely on the importer's policy.
replicating	Default import of this module results in a new instance.
singleton	Always shares a single instance (ignores importer's policy).

[3] Keywords are introduced for the sake of clarity. A real system might use different syntax.

[4] If another module imports m as amn then they share the same instance. This option is here to cover the general case.

Collecting these alternatives, we have three different types of dependency between the importing and the imported superpackage: *shared*, *own*, and *as*. The following table summarizes the above-described interaction between different annotations: [5]

IMPORTING \ IMPORTED	*default*	**replicating**	**singleton**
default	*shared*	*own*	*shared*
shared	*shared*		
own	*own*		
as	*as*		

The '**replicating**' flag says "use this module as shared at your own risk," whereas the '**singleton**' flag says "this module only makes sense if there is a single instance of it," hence we do not allow the importing module to override the latter. The above table shows how we put the intended semantics before safety in a concurrent environment.

4 Formalization

In this section, we describe the formalization of the above-mentioned solutions in improved JAM (iJAM), a modified version of our previously developed LJAM. As LJAM, iJAM, too, was formalized with Ott [9]. Due to the lack of space, we only briefly overview the interesting parts of the formalization. The full definitions (including all the semantic rules) can be found online [6, 4, 7].

The user syntax for iJAM's module files is

$$repl\ \mathbf{superpackage}\ mn\ \{\ \overline{\mathbf{member}\ pn;}\ \overline{imp;}\ \overline{\mathbf{export}\ fqn;}\ \}$$

Comparing this to the user syntax of LJAM's module files shown in §2, the definition is now prefixed with $repl$, and $\overline{imp;}$ has replaced $\overline{\mathbf{import}m;}$. The definition of the two new entities (and the import dependency construct imp_dep) is shown in Fig. 1.

The combination of the two statically defined replication policies $repl$ and imp results in the actual replication policy imp_dep used at runtime — see the table in §3.2.

As in LJAM, the module instances are stored in repositories' caches. However, LJAM's caches simply mapped module definitions (md's) to their instances: $md \rightharpoonup mi$. In iJAM, the import dependencies need to be taken into account, so the cache type becomes: $md \rightharpoonup (imp\_dep \rightharpoonup mi)$.[6]

The module hierarchy MH stores the connection between module instances. In LJAM, this was simply $mi \rightharpoonup \overline{mi}$, but in iJAM each imported module instance is also associated with the appropriate boundary renaming of class names, so the module hierarchy's structure becomes $mi \rightharpoonup \overline{mi\ br}$. This way the class lookup function can easily update the name of the class it is looking for when crossing module boundaries.

The well-formedness relations were updated to reflect the new structures. The semantics of the administration actions, e.g. module initialization, were changed to account for different import dependencies and boundary renaming. Class resolution was updated as described in §3.1. Unfortunately, lack of space prevents any more detail here.

[5] This table might seem more complex than the relation it represents. It is not clear whether there is a simpler relation, which gives the same expressivity and convenience.

[6] Since module instances resulting from **Own** mi can only be linked against once, we could have excluded them from the cache.

$repl$	$::=$		replication modifier
	$\mid$		default
	$\mid$	$\mathbf{replicating}$	replicating
	$\mid$	$\mathbf{singleton}$	singleton
imp	$::=$		import statement
	$\mid$	$\mathbf{import}\ m\ br$	default
	$\mid$	$\mathbf{import\ shared}\ m\ br$	shared
	$\mid$	$\mathbf{import\ own}\ m\ br$	own
	$\mid$	$\mathbf{import}\ m\ \mathbf{as}\ amn\ br$	as
br	$::=$		boundary renaming
	$\mid$		no renaming
	$\mid$	$\mathbf{with}\ fqn_1\ \mathbf{as}\ fqn'_1,\ ...,\ fqn_k\ \mathbf{as}\ fqn'_k$	renaming pairs
imp_dep	$::=$		import dependency
	$\mid$	$\mathbf{Shared}$	default import
	$\mid$	$\mathbf{Own}\ mi$	instance of imported module
	$\mid$	$\mathbf{As}\ amn$	ref. to imported module

Fig. 1. Some of the entities introduced by iJAM

5 Mechanically Proving Type-Soundness

By providing Isabelle/HOL [5] homomorphisms for language productions to Ott, the tool is able to generate theorem prover definitions of the meta-types and semantic rules.

5.1 Type-Soundness Theorems

We prove type soundness by proving progress and type preservation properties. The progress property states that: if a configuration $(P, L, H, \overline{s})$ (where P is a program, L a variable mapping, H a heap, and $\overline{s}$ statements to execute) is well-formed in some type environment Γ, and there are still some statements $\overline{s}$ left to execute, then there exists some configuration *config*, which the current configuration reduces to in one step ($\longrightarrow$).

Theorem 1 (Progress).

$$\Gamma \vdash (P, L, H, \overline{s}) \ \wedge\ \overline{s} \neq [] \implies \exists \mathit{config}.\ (P, L, H, \overline{s}) \longrightarrow \mathit{config}$$

The type preservation property states that: if *config* is a well-formed configuration in some type environment Γ, and *config* reduces in one step to config' through either statement reduction ($\longrightarrow$), or administrator action reduction ($\xrightarrow{a}$) in case of LJAM and iJAM, config' is well-formed in some greater ($\subseteq_m$) type environment Γ'.

Theorem 2 (Type Preservation).

$$\begin{array}{l}\Gamma \vdash \mathit{config} \wedge (\mathit{config} \longrightarrow \mathit{config}' \vee \mathit{config} \xrightarrow{a} \mathit{config}') \\ \quad \implies \exists \Gamma'.\ \Gamma \subseteq_m \Gamma' \wedge \Gamma' \vdash \mathit{config}'\end{array}$$

5.2 Our Experience

The semantic rules are defined as inductively-defined relations. Some of these relations, e.g. the class resolution, are easier to deal with in a theorem prover if expressed as functions. For those, we wrote Isabelle/HOL functions, and proved equivalence between the Ott-generated relation and the corresponding function.

When defining a function with non-primitive recursion in Isabelle/HOL, one also has to prove its termination. Especially in the case of `find_path_f`, the function that finds the inheritance path for a particular class, proving termination was a challenge. In our first attempt, we defined the relation/function so that it failed if 'the path found so far' was greater than the number of classes in the program. This worked fine until we had to prove well-formedness of a program with an extra module instance — the size of the program increased, so we could not prove semantic preservation for this function in general, since the function failed in one case, and not in the other. This forced us to define the acyclicity property among the class definitions, a property that is independent of the size of the program; however, we still had to prove the preservation of this property. Similarly, we had to define an acyclicity property among module instances to prove termination and well-formedness of `find_cld_in_imports_f`, a function finding class definitions among module imports.

All three formalizations also use a non-standard subtyping relation: a type τ is a subtype of τ' *iff* a class definition corresponding to τ is in the inheritance path of τ'. We then derive type reflexivity and transitivity, not vice versa. This definition makes it relatively easy to prove lemmas such as method type preservation.

One of the key lemmas in the LJAM's proof, which iJAM's new class resolution breaks, is `find_cld_same_ctx`. It says that if we look for a class with name fqn in context ctx and we find a class definition cld' in context ctx', then we will get the same result if we start the search at ctx' instead:

Lemma 1 (`find_cld_same_ctx`) **.**

$$\begin{array}{l}\text{find\_cld\_f } P\ ctx\ fqn = \text{Some } (ctx', cld') \Longrightarrow \\ \text{find\_cld\_f } P\ ctx'\ fqn = \text{Some } (ctx', cld')\end{array}$$

In iJAM, this lemma did not hold any more, because fqn was not necessarily the same as the fully-qualified name of cld'. We modified the lemma by replacing fqn in the goal with $(\text{full\_name\_f } cld')$. However, the lemma was still false, because the two function calls first search within the core libraries, each with a possibly different class name, which can therefore lead to a different result. If we could not use the modified lemma in iJAM's proof, we would not be able to reuse large parts of LJAM's proof.

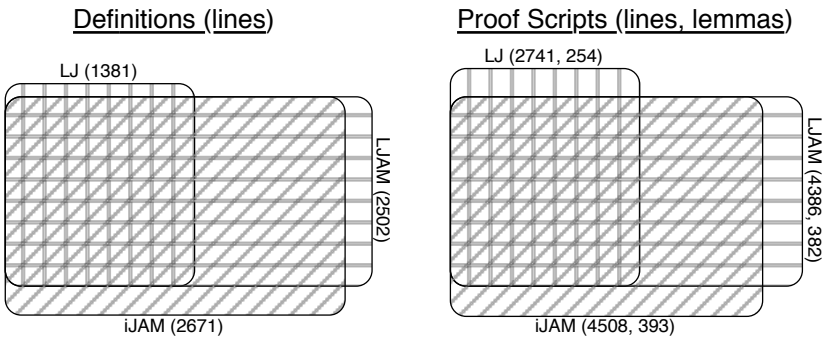
We solved this problem by placing a well-formedness condition on boundary renaming, which prevents module definitions from renaming a class *from* and *to* a name already exported by the core library module. Actually, already the *from* part makes the modified lemma hold again,[7] but we added the *to* part, too, in order to avoid unexpected class resolution results. These restrictions ensure that a class reference resolves to a core library class *iff* the reference is a name of a class exported by the core library.

[7] It is still not clear to us whether iJAM is type-sound without this restriction.

6 Reuse in Definitions and Proof Scripts

Since LJAM is an extension of LJ, and iJAM is based on LJAM, they share much of the semantic definitions. In particular, the language statements, e.g. the method call statement, have syntactically identical definitions in all three languages; also syntactically identical are the statement well-formedness and reduction relations — that is, the semantics of these relations differ through different definitions of the syntactically identical judgements (e.g. the class resolution judgement) used in their rules.

The proof scripts for the progress and well-formedness of the statement reduction relation are practically identical for all the three languages (5 lines out of 350 lines differ). This is achieved by carefully abstracting the key lemmas, e.g. the lemma for the 'method type preservation' property (the lemma is too large to show here). Due to such abstractions, we were able to achieve high reuse in both the definitions and the proof scripts as shown by the following two diagrams (relative area corresponds to relative no. of lines of definition/proof script):



7 The Implementation

Our proof-of-concept implementation runs on top of Java; the module's code can contain any valid Java code. We implemented module files (with JavaCC [10]), repositories, module definitions, module instances, the module initialization mechanism according to iJAM's semantics, and a classloader, which respects iJAM's class resolution. The user can also enter compatibility mode, where the system behaves according to LJAM's semantics instead. We do not implement realistic compilation of module files, an administration console, or eager typechecking as specified by LJAM's and iJAM's semantics — typechecking is done lazily, as is normal in Java.

Each module instance needs to create distinct types, as well as replicate the static state of its classes. This can be achieved by having a classloader per one module instance. Each module object holds references to modules it imports, to which it delegates the class resolution as necessary. Because it is not possible to map two distinct names to the same `Class` object, sharing through renamed classes is not supported; however, we believe that only a small change to the JVM is required to allow this.

Using iJAM's user syntax presented in §4, we here give a simple example:

```
superpackage XMLParser     {member; export Parser;}
superpackage XSLT          {member; import XMLParser; export Config;}
superpackage ServletEngine {member; import XMLParser; export Config;}
superpackage WebCalendar   {member; import XSLT;
                            import ServletEngine with Config as SEConfig;}
```

Both `Config` classes use the *XMLParser*::`Parser`, which tracks the number of its instances. *WebCalendar*::`Main` (not exported) uses *XSLT*::`Config` and `SEConfig` (*ServletEngine*::`Config`). Running `Main` outputs:

```
XSLT::Config: using 1. instance of Parser.
ServletEngine::Config: using 2. instance of Parser.
```

This indicates that there is only one *XMLParser* instance. One way to achieve multiple instances of the module is to tag it with **replicating**; in that case, the output shows `1. instance` in both cases. The full source is available online [7].

8 Related Work

In this section, we compare iJAM to the other module systems and related language features. In particular, we focus on class resolution, boundary renaming of classes, sharing of static data and types, and separate compilation.

In module systems based on the ML module system, the underlying language usually supports module-aware class references. Examples of such systems are MOBY [11] and CGEN [12]. The classes are looked up in the user module first, only then in the importing ones, unless the user refers to a specific imported module. These systems normally allow type renaming through external names of exported (visible) types.

Jiazzi [13] and ComponentJ [14] are two examples of module systems based on Units [15]. Although renaming is not supported, these systems provide powerful ways of combining components. Jiazzi gives module capabilities to packages, so package names in class references refer to imported modules — this simplifies the class resolution, but changes the semantics of the underlying Java language.

Bauer et al. describe a module system [16], which supports the renaming of both classes and modules, allows module-aware class references, and uses those to guide the class resolution — both of these changes, although useful, change the underlying language, which we are trying to avoid.

MJ [17], a module system similar to JAM, checks the module constraints and sets the appropriate `CLASSPATH` for the standard Java compiler. Its access control mechanism is quite expressive, allowing sophisticated relationships between different modules, such as selective importing and exporting, hiding, and sealing. Surprisingly, no renaming is supported. The problem of ambiguous class references seems to be ignored.

Of the above systems, only MOBY, Jiazzi and CGEN support separate compilation. MOBY achieves this with various restrictions to the language, whereas Jiazzi and CGEN require user-specified module interfaces. SMARTJAVAMOD [18] supports separate compilation through generation and verification of compositional constraints [19].

.NET assemblies [20] represent well-defined boundaries of security, namespaces, and versions. They are similar to JAM module definitions in many respects. Fusion, the

assembly binder for .NET, finds, loads, and binds assemblies before execution. Compilation of an assembly requires all of its imports to be present, so that the created bytecode can contain only explicit references — if there are many possible ways of resolving a class reference, an error is thrown at compile-time.

Currently the most widespread module framework for Java, OSGi [21] is a highly-customizable framework built on top of Java that promotes service-oriented programming. OSGi has a similar class resolution scheme to JAM, which we have argued here is counter-intuitive. Additionally to JAM, it, like MJ, supports *dynamic imports*, where an import dependency is resolved lazily at runtime — this increases the expressive power, but also lowers type safety guarantees, allowing type errors to happen also at runtime. As far as we are aware, it also has no module sharing control, boundary renaming, or namespace hiding, which means that it, too, suffers from the problems described in §3.

All of the systems mentioned so far (except OSGi) use a single copy of static data per application, i.e. modules are used as static libraries. In OSGi and JAM, modules are used as dynamic libraries, where multiple programs share types and static data.

Family polymorphism [22–24] is a flexible and transparent mechanism for code reuse. It gives classes some properties of modules, addresses hierarchical information hiding, and allows improved forms of inheritance. However, it does not address packaging, distribution, deployment, import renaming, or separate compilation.

9 Conclusions and Future Work

In this paper, we have presented and solved some of the most important problems with the JAM. We formalized the solutions in a language called iJAM, an extension of our previously developed LJAM. We defined the syntax, the type system, and the operational semantics of iJAM. In Isabelle/HOL, we proved type soundness for LJ, LJAM, and iJAM. Furthermore, we built a proof-of-concept implementation on top of Java, which can follow the semantics of either LJAM or iJAM. All the definitions, proofs, source code, documentation, and other documents can be found online [3, 4, 6, 7].

A good module system enables the user to restrict herself from making unintended dependencies. To prevent many unintended class name dependencies, JAM allows selective *exporting*. Our work substantially improves on this by providing support for boundary renaming (more expressive than selective *importing*). We also allow multiple module instances, which help with preventing unintended data/type dependencies.

The formalization tools were key to this work. Ott found many consistency errors with the definitions automatically. This gave more courage to experiment with alternative definitions, and allowed compiler-error based regression testing. By mechanizing the meta-theory in Isabelle, it was easy to identify incorrectly formulated judgements and incomplete relations. Through abstractions in the definitions and the proof scripts, we achieved a high-level of reuse in both, which substantially sped up the process.

We plan on trying to use the recently developed Isabelle code generation tools [25] to generate reference implementations of the systems, which could then be compared to existing ones. Formalising OSGi is an interesting option for future work, which would allow detailed comparison of the two systems. This would likely help the industrial attempts to provide clean interoperability between the two systems.

Acknowledgments

We would like to thank Peter Sewell and Matthew Parkinson for their support, ideas, and useful comments on earlier versions of this paper. We acknowledge the support of EPSRC grants DTA-RG44132, EP/C510712, and GR/T11715/01.

References

1. Sun Microsystems, Inc.: JSR-277: Java™ Module System. http://jcp.org/en/jsr/detail?id=277 (October 2006) Early Draft.
2. Sun Microsystems, Inc.: JSR-294: Improved Modularity Support in the Java™ Programming Language. http://jcp.org/en/jsr/detail?id=294
3. Strniša, R., Sewell, P., Parkinson, M.: The Java Module System: Core Design and Semantic Definition. In: Proc. of OOPSLA'07, ACM (October 21-25, 2007) 499–514
4. Strniša, R.: Lightweight Java Module System (LJAM). http://www.cl.cam.ac.uk/~rs456/ljam/ (March 2007)
5. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
6. Strniša, R., Parkinson, M.: Lightweight Java (LJ). http://www.cl.cam.ac.uk/~rs456/lj/ (September 2006)
7. Strniša, R.: improved Java Module System (iJAM). http://www.cl.cam.ac.uk/~rs456/iJAM/ (November 2007)
8. Liang, S., Bracha, G.: Dynamic Class Loading in the Java Virtual Machine. In: Proc. of OOPSLA'98. (October 18-22, 1998) 36–44
9. Sewell, P., Zappa Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective Tool Support for the Working Semanticist. In: Proc. of ICFP'07, ACM (October 1-3, 2007) 1–12
10. Kodaganallur, V.: Incorporating Language Processing into Java Applications: A JavaCC Tutorial. IEEE Software **21**(4) (2004) 70–77
11. Fisher, K., Reppy, J.: The Design of a Class Mechanism for MOBY. In: Proc. of PLDI'99, New York, NY, USA, ACM (May 1-4, 1999) 37–49
12. Sasitorn, J., Cartwright, R.: Component NextGen: A Sound and Expressive Component Framework for Java. SIGPLAN Not. **42**(10) (2007) 153–170
13. McDirmid, S., Flatt, M., Hsieh, W.: Jiazzi: New Age Components for Old Fashioned Java. In: Proc. of OOPSLA'01. Volume 36. (October 14-18, 2001) 211–222
14. Seco, J.C., Caires, L.: A Basic Model of Typed Components. In Bertino, E., ed.: Proc. of ECOOP'00. Volume 1850 of LNCS., Springer (June 12-16, 2000) 108–128
15. Flatt, M., Felleisen, M.: Units: Cool Modules for HOT Languages. SIGPLAN Not. **33**(5) (1998) 236–248
16. Bauer, L., Appel, A.W., Felten, E.W.: Mechanisms for Secure Modular Programming in Java. Software—Practice and Experience **33**(5) (2003) 461–480
17. Corwin, J., Bacon, D.F., Grove, D., Murthy, C.: MJ: A Rational Module System for Java and its Applications. In Crocker, R., Jr., G.L.S., eds.: Proc. of OOPSLA'03, ACM (October 26-30, 2003) 241–254
18. Ancona, D., Lagorio, G., Zucca, E.: Smart Modules for Java-like Languages. In: Proc. of FTfJP'05. (July 26, 2005)
19. Ancona, D., Damiani, F., Drossopoulou, S., Zucca, E.: Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In Palsberg, J., Abadi, M., eds.: Proc. of POPL'05, ACM (January 12-14, 2005) 26–37

20. DevelopMentor: Assemblies Module - .NET: Building Applications and Components with C# (January 2004)
21. OSGi™ Alliance: About the OSGi Service Platform. 4.1 edn. (November 2005)
22. Ernst, E.: gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark (1999)
23. Odersky, M., Zenger, M.: Scalable Component Abstractions. In: Proc. of OOPSLA '05, New York, NY, USA, ACM (October 16-20, 2005) 41–57
24. Clarke, D., Drossopoulou, S., Noble, J., Wrigstad, T.: Tribe: A Simple Virtual Class Calculus. In: Proc. of AOSD'07. (March 12-16, 2007)
25. Haftmann, F.: Code generation from Isabelle/HOL theories. (November 2007)

Constancy Analysis

Samir Genaim and Fausto Spoto

[1] CLIP, Technical University of Madrid (UPM), Spain
[2] Università di Verona, Italy
samir@clip.dia.fi.upm.es, fausto.spoto@univr.it

Abstract. A reference variable x is constant in a piece of code C if the execution of C does not modify the heap structure reachable from x. This information lets us infer purity of method arguments, an important ingredient during the analysis of programs dealing with dynamically allocated data structures. We define here an abstract domain expressing constancy as an abstract interpretation of concrete denotations. Then we define the induced abstract denotational semantics for Java-like programs and show how constancy information improves the precision of existing static analyses such as sharing, cyclicity and path-length.

1 Introduction

A major difference between pure functional/logic programming and imperative programming is that the latter uses destructive updates. That is, data structures are *mutable*: they are built *and later modified*. This can be both recognized as a superiority of imperative programming, since it allows one to write faster and simpler code, and as a drawback, since if two variables share a data structure then a destructive update to the data reachable from one variable may affect the data reachable from the other. This often leads to subtle programming bugs.

It is hence important to control what a method invocation modifies. Some methods do not modify the data structures reachable from their parameters. Others only modify those reachable from some but not all parameters. Namely, some parameters are *constant* or *read-only*, others may be modified. If all parameters of a method are constant, the method is *pure* [10]. Knowledge about purity is important since pure methods can be invoked in any order, which lets compilers apply aggressive optimizations; pure methods can be used in program assertions [7]; they can be skipped during many static analyses or more precisely approximated than other methods. This results in more efficient and more precise analyses. For instance, sharing analysis [11] can safely assume that sharing is not introduced during the execution of a pure method. In general, all static analyses tracking properties of the heap benefit from information about purity.

For these reasons, software specification has found ways of expressing purity of methods and constant parameters. The notable example is the Java Modeling Language [7], which uses the `assignable` clause to specify those heap positions that might be mutated during the execution of a method. Those clauses are manually provided and used by many static analyzers, such as ESC/Java [6]

and ChAsE [4]. However, those tools do not verify the correctness of the user-provided `assignable` clauses, or use potentially incorrect verification techniques. A formally correct verification technique is defined in [14], but has never been implemented. In [10] a formally correct analysis for purity is presented, it is based on a preliminary points-to and escape analysis, and an implementation exists and has been applied to some small size examples. In [8] a correct and precise algorithm for statically inferring the reference immutability qualifiers of the Javari language has been presented. The algorithm has been implemented in the Javarifier tool.

In this paper, we investigate an alternative technique aiming at determining which parameters of a method are constant. We use abstract interpretation [5] and perform a static analysis over the reduced product of the sharing domain in [11] (the *sharing component*) and a new abstract domain expressing the set of variables bound to data structures mutated during the execution of a piece of code (the *purity component*). The use of reduced product is justified since the sharing component helps the purity component during a destructive update, by identifying which variables share the updated data structure and hence lose their purity; conversely, the sharing component uses the purity component during method calls, since only variables sharing with non-pure parameters of a method m can be made to share during the execution of m.

Our technique is sometimes less precise than [10], since it does not use the field names (i.e., we do not keep information on which field has been updated, but rather that a field has been updated). However, it is implemented in a completely flow-sensitive and context-sensitive fashion, which improves its precision. Moreover, it is expressed in terms of Boolean formulas implemented through binary decision diagrams, resulting in fast analyses scaling to quite big programs. Our contributions are hence: (1) a definition of the reduced product of sharing and purity; (2) its application to large programs; (3) a comparison of the precision of sharing analysis alone with that of sharing analysis in reduced product with purity; and (4) an evaluation of the extra precision induced by the purity information during static analyses tracking properties of the heap, namely, possible cyclicity of data structures [9] and path-length of data structures [13].

The paper is organized as follows: Section 2 defines the syntax and semantics of a simple Object-Oriented language; Section 3 develops our constancy analysis for that language; Section 4 provides an experimental evaluation.

2 Our Simple Object-Oriented Language

This section presents syntax and denotational semantics of a simple Object-Oriented language that we use through the paper. Its commands are normalized versions of corresponding Java commands: the language supports reference and integer types; in method calls, only syntactically distinct variables can be actual parameters, which is a form of normalization and does not prevent them from being bound to shared data-structures at run-time; in assignments, the left hand side is either a variable or the field of a variable; Boolean conditions are kept

generic, they are conditions that are evaluated to either *true* or $false$; iterative constructs, such as the `while` loop, are not supported since they can be implemented through recursion. These assumptions are only for the sake of clear and simple presentation and can be relaxed without affecting subsequent results. A program has a set of *variables* $\mathcal{V}$ (including *out* and *this*) and a finite poset of *classes* $\mathbb{K}$. The *commands* of the language are

$$\begin{aligned} com ::= \quad & v := c \mid v := w \mid v := \texttt{new}\ \kappa \mid v := w + z \mid v := w.\texttt{f} \mid v.\texttt{f} := w \mid \\ & v := v_0.\texttt{m}(v_1, \ldots, v_n) \mid \texttt{if}\ e\ \texttt{then}\ com_1\ \texttt{else}\ com_2 \mid com_1; com_2 \end{aligned}$$

$v, w, z, v_0, v_1, \ldots, v_n \in \mathcal{V}$ are distinct variables, $c \in \mathbb{Z} \cup \{\texttt{null}\}$, $\kappa \in \mathbb{K}$ and e is a Boolean expression. The signature of a method $\kappa.\texttt{m}(t_1, \ldots, t_p){:}t$ refers to a method called `m` expecting p parameters of type $t_1, \ldots, t_p \in \mathbb{K} \cup \{\texttt{int}\}$, respectively, returning a value of type t and *defined* in class κ with a statement

$$t\ \texttt{m}(w_1{:}t_1, \ldots, w_n{:}t_n)\ \texttt{with}\ \{w_{n+1}{:}t_{n+1}, \ldots, w_{n+m}{:}t_{n+m}\}\ \texttt{is}\ com,$$

where $w_1, \ldots, w_n, w_{n+1}, \ldots, w_{n+m} \in \mathcal{V}$ are distinct, not in $\{out, this\}$ and have *type* $t_1, \ldots, t_n, t_{n+1}, \ldots, t_{n+m} \in \mathbb{K} \cup \{\texttt{int}\}$, respectively. Variables $w_1, \ldots, w_n$ are the *formal parameters* of the method and $w_{n+1}, \ldots, w_{n+m}$ are its *local variables*. The method also uses a variable *out* of type t to store its *return value*. For a given method signature $m = \kappa.\texttt{m}(t_1, \ldots, t_p) : t$, we define $m^b = com$, $m^i = \{this, w_1, \ldots, w_n\}$, $m^o = \{out\}$, $m^l = \{w_{n+1}, \ldots, w_{n+m}\}$ and $m^s = m^i \cup m^o \cup m^l$. Classes might declare fields of type $t \in \mathbb{K} \cup \{\texttt{int}\}$.

We use a denotational semantics, hence compositional, in the style of [15]. However, we use a more complex notion of state, which assumes an infinite set of *locations*. Basically, a state is a pair which consists of a frame and a heap, where a frame maps variables to values and a heap maps locations to objects. Note that since we assume a denotational semantics, a state has a single frame, rather than an activation stack of frames as it is required in operational semantics. We let $\mathbb{L}$ denote an infinite set of *locations*, and let $\mathbb{V}$ denotes the set of *values* $\mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$. A *frame* over a finite set of variables V is a mapping that maps each variable in V into a value from $\mathbb{V}$; a *heap* is a partial map from $\mathbb{L}$ into *objects*. An object is a pair that consists of its class tag κ and a frame that maps its *fields* (identifiers) into values from $\mathbb{V}$; we say that it *belongs to* class κ or *has* class κ. Given a class κ, we assume that $newobj(\kappa)$ return a new object where its fields are initialized to 0 or depending on their types. If ϕ is a frame and $v \in V$, then $\phi(v)$ is the value of variable v. If μ is a heap and $\ell \in \mathbb{L}$, then $\mu(\ell)$ is the object bound in μ to ℓ. If o is an object, then $o.tag$ denotes its class and $o.\phi$ denotes its frame; if f is a field of o, then sometimes we use $o.f$ to refer to (or set) its value instead of going through its frame.

Definition 1 (computional state). *Let V denotes the set of variables in scope at a given program point p. The set of possible* states *at p is*

$$\Sigma_V = \left\{ \langle \phi, \mu \rangle \;\middle|\; \begin{array}{l} 1.\ \phi \text{ is a frame over } V \text{ and } \mu \text{ is a heap} \\ 2.\ \mathrm{rng}(\phi) \cap \mathbb{L} \subseteq \mathrm{dom}(\mu) \\ 3.\ \forall \ell \in \mathrm{dom}(\mu).\ \mathrm{rng}(\mu(\ell).\phi) \cap \mathbb{L} \subseteq \mathrm{dom}(\mu) \end{array} \right\}$$

Conditions 2 *and* 3 *guarantee the absence of dangling pointers. Given* $\sigma = \langle \phi, \mu \rangle \in \Sigma_V$*, we use* ϕ_σ *and* μ_σ *to refer to its frame and heap respectively.* □

Now we define the notion of *Denotations* which are the input/output semantics of a piece of code. Basically they are mappings from states to states which describe how the input state is changed when the corresponding code is executed. *Interpretations* are a special case of denotations which provide a denotation for each method in terms of its input and output variables.

Definition 2. *A* denotation δ *from* V *to* V' *is a partial function from* Σ_V *to* $\Sigma_{V'}$*. We often refer to* $\delta(\sigma) = \sigma'$ *as* $(\sigma, \sigma') \in \delta$*. The set of denotations from* V *to* V' *is* $\Delta(V,V')$*. An* interpretation ι *maps methods to denotations and is such that* $\iota(m) \in \Delta(m^i, m^i \cup m^o)$ *for each method* $m = \kappa.\mathtt{m}(t_1, \ldots, t_p) : t$ *in the given program. The set of all possible interpretations is written as* $\mathbb{I}$*.* □

The denotational semantics associates a denotation to each command of the language. Let V denotes a set of variables. Let $\iota \in \mathbb{I}$. We define the *denotation for commands* $\mathcal{C}^\iota_V[\![\_]\!] : com \mapsto \Delta(V,V)$, as their input/output behaviour:

$$
\begin{aligned}
\mathcal{C}^\iota_V[\![v{:=}c]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto c]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}w]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \phi_\sigma(w)]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}\mathtt{new}\ \kappa]\!] &= \{(\sigma, \sigma[\mu_\sigma(\ell) \mapsto newobj(\kappa)]) \mid \sigma \in \Sigma_V,\ \ell \notin \mathrm{dom}(\mu_\sigma)\} \\
\mathcal{C}^\iota_V[\![v{:=}w + z]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \phi_\sigma(w) + \phi_\sigma(z)]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}w.f]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \mu_\sigma\phi_\sigma(w).f]) \mid \sigma \in \Sigma_V, \phi_\sigma(w) \neq \mathtt{null}\} \\
\mathcal{C}^\iota_V[\![v.f{:=}w]\!] &= \{(\sigma, \sigma[\mu_\sigma\phi_\sigma(v).f \mapsto \phi_\sigma(w)]) \mid \sigma \in \Sigma_V, \phi_\sigma(v) \neq \mathtt{null}\} \\
\mathcal{C}^\iota_V\left[\!\!\left[\begin{array}{r}\mathtt{if}\ e\ \mathtt{then}\ com_1 \\ \mathtt{else}\ com_2\end{array}\right]\!\!\right] &= \begin{array}{l}\{(\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_1]\!] \mid \sigma \models e \approx true\} \cup \\ \{(\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_2]\!] \mid \sigma \models e \approx false\}\end{array} \\
\mathcal{C}^\iota_V[\![com_1; com_2]\!] &= \{(\sigma, \sigma'') \mid (\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_1]\!] \wedge (\sigma', \sigma'') \in \mathcal{C}^\iota_V[\![com_2]\!]\}
\end{aligned}
$$

The denotation for a method call $\mathcal{C}^\iota_V[\![v{:=}v_0.\mathtt{m}(v_1, \ldots, v_p)]\!]$ should consider the denotation $\iota(m)$ (where m is the called method) and extend it to fit in the calling scope and update the variable v. Assume the method signature is $\mathtt{m}(t_1, \ldots, t_p){:}t$, and that we have a lookup procedure $\mathcal{L}$ that, for any given $\sigma \in \Sigma_V$, fetches the actual method that is called depending on the run-time class of v_0. Then the method call denotation is defined as follows:

$$
\left\{ (\sigma, \langle \phi_\sigma[v \mapsto \phi''_\sigma(out)], \mu''_\sigma \rangle) \;\middle\|\; \begin{array}{l} 1.\ \sigma \in \Sigma_V, \phi_\sigma(v_0) \in \mathrm{dom}(\mu_\sigma); \\ 2.\ m = \mathcal{L}(v_0, \sigma, \mathtt{m}(t_1, \ldots, t_p){:}t); \\ 3.\ (\sigma', \sigma'') \in \iota(m); \\ 4.\ \mu_\sigma \equiv \mu'_\sigma, \forall 0 \leq i \leq p.\ \phi_\sigma(v_i) = \phi'_\sigma(w_i) \end{array} \right\}
$$

The concrete denotational semantics of a program is the least fixpoint of the following transformer of interpretations [3].

Definition 3 (Denotational semantics). *The denotational semantics of a program* P *is defined as* $\bigcup_{i \geq 0} T^i_P(\iota_0)$ *, i.e. the* least fixed point *of* T_P *where* T_P *is:*

$$T_P(\iota) = \left\{ (m, X) \;\middle\|\; \begin{array}{l} 1.\ m \in P \\ 2.\ \sigma \in \Sigma_{m^s}, \forall v \in m^l.\ \phi_\sigma(v) = 0 \text{ or } \phi_\sigma(v) = \texttt{null} \\ 3.\ X = \{(\sigma|_{m^i}, \sigma'|_{m^i \cup m^o}) \mid (\sigma, \sigma') \in \mathcal{C}^\iota_{m^s}[\![m^b]\!]\} \end{array} \right\}$$

and $\iota_0 = \{(m, \emptyset) \mid m \in P\}$ *and* $\forall \iota_1, \iota_2 \in \mathbb{I}$ *the union* $\iota_1 \cup \iota_2$ *is defined as* $\{(m, X_1 \cup X_2) \mid m \in P, (m, X_1) \in \iota_1, (m, X_2) \in \iota_2\}$ □

3 Constancy Analysis

We want to design an analysis to infer definite information about constant data structures. This can be done by tracking data structures that are not modified (definite information), or by tracking data structures that might be modified (may information). We follow the latter approach as we believe it easier. In addition, we want to analyze methods in a context independent way, and later adapt the result to any calling context.

Example 1. Consider the following method:

```
A m(x:A, y:A) with {} is y:=y.next; x.next:=y; out:=y;
```

The only command that might modify the heap structure is "`x.next:=y`". Note that "`y:=y.next`" does not affect the heap structure but rather changes the heap location stored in `y`. This method might be called in different contexts where the actual parameters: (1) do not have any common data structure; or (2) have a common data structure. In the first case, "`x.next:=y`" might modify only the data structure pointed by the first argument. In the second case, it might modify a common data structure for `x` and `y`, and therefore we say that both arguments might be modified. We describe this behaviour by the Boolean formula $\check{x} \wedge (\check{y} \leftrightarrow x \check{\cdot} y)$, which is interpreted as: (1) in any calling context, the data structure the first argument points to when the method is called might be modified by the method (expressed by $\check{x}$); and (2) the data structure that the second argument points to when the method is called, might be modified by the method (expressed by $\check{y}$) iff x and y might share a data structure when the method is called (expressed by $x \check{\cdot} y$).

□

We define now the set of reachable heap locations from a given reference variable, which we need to define the notion of *constant heap structure*.

Definition 4 (reachable heap locations). *Let* μ *be a heap. The set of* locations reachable from $\ell \in \mathrm{dom}(\mu)$ *is* $L(\mu, \ell) = \cup\{L^i(\mu, \ell) \mid i \geq 0\}$ *where* $L^0(\mu, \ell) = \mathrm{rng}(\mu(\ell).\phi) \cap \mathbb{L}$ *and* $L^{i+1}(\mu, \ell) = \cup\{\mathrm{rng}(\mu(\ell')) \cap \mathbb{L} \mid \ell' \in L^i(\mu, \ell)\}$. *The set of reachable heap locations from* v *in* $\sigma \in \Sigma_V$*, denoted* $L_V(\sigma, v)$*, is* $\{\phi_\sigma(v)\} \cup L(\mu_\sigma, \phi_\sigma(v))$ *if* $\phi_\sigma(v) \in dom(\mu_\sigma)$*; and the empty set otherwise.* □

Definition 5 (constant reference variable). *A reference variables $v \in V$ is constant with respect to a denotation δ, denoted $\mathsf{c}(v, \delta)$, iff for any $(\sigma_1, \sigma_2) \in \delta$ all locations in $L_V(\sigma_1, v)$ are constant with across δ, namely $\forall \ell \in L_V(\sigma_1, v)$, $\mu_{\sigma_1}(\ell)$ and $\mu_{\sigma_2}(\ell)$ have the same class tag and agree on their* reference *field values.* □

The definition above considers modifications of fields of reference type only. The reason for concentrating on reference fields is that we have developed this analysis for a specific need which requires tracking updates only in the shape of the data structure (see Section 4). Tracking updates of integer fields can simply done by modifying the above definition to consider those updates. In what follows, a modification of a variable stands for a modification of the shape of the heap structure reachable from that variable.

Definition 6 (common heap location). *$x, y \in V$ have a common heap location (share) in a state $\sigma \in \Sigma_V$ if and only if $L_V(\sigma, x) \cap L_V(\sigma, y) \neq \emptyset$* □

We define now an abstract domain which captures a set of variables that *might* be modified by a concrete denotation.

Definition 7 (update abstract domain). *The update abstract domain U_V is a partial order $\langle \wp(V), \subseteq \rangle$. Its concretization function $\gamma_V : \mathsf{U}_V \to \Delta(V, V')$ is defined as $\gamma_V(X) = \{\delta \mid \forall v \in V.\ \neg \mathsf{c}(v, \delta) \to (v \in X)\}$.* □

As we have seen in Example 1, information about possible sharing between variables is important for a precise constancy analysis. There are many ways for inferring such information. Here, we use the pair-sharing domain [11]. Moreover, constancy information improves the precision of method calls in pair sharing analysis. This is because the execution of a method m can introduce sharing between non-constant parameters only. Hence we design an analysis over the (reduced) product of the update domain U_V and of the pair-sharing domain SH_V, denoted by $\mathsf{SH}{\times}\mathsf{U}_V$. Informally, the pair sharing domain abstracts an element $s \in \wp(\Sigma_V)$ to a set sh of symmetric pairs of the form (x, y) where $x, y \in V$. If $(x, y) \in sh$ then x and y *might* share in s, and if $(x, y) \notin sh$ then they cannot share, so that if $(x, x) \notin sh$ then x must be `null` in s. In what follows, instead of saying *might share* we simply say share.

Figure 1 defines abstract denotations for our simple language over $\mathsf{SH}{\times}\mathsf{U}_V$. They are Boolean functions corresponding to the elements of $\mathsf{SH}{\times}\mathsf{U}_V$. For a piece of code C, the Boolean variables:

- $x\check{}y$ and $x\hat{}y$ indicate if x and y *share* before and after executing C, respectively. Since pair sharing is symmetric, $x\check{}y$ and $y\check{}x$ are equivalent Boolean variables; and
- $\check{x}$ and $\hat{x}$ indicate if x is modified with respect to its value before and after C (by the program execution), respectively.

Each abstract denotation is defined in terms of a Boolean function $\varphi \wedge \psi$, where φ propagates (forward) *sharing* information and ψ propagates (backwards) *update* information. In what follows we explain the meaning of each abstract denotation:

$$
\begin{aligned}
\mathcal{A}^{\iota}_V[\![v{:=}\texttt{null}]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge \varphi_1 \\
&\quad -\varphi_1 = (\wedge\{\neg x \hat{\cdot} v \mid x \in V\}) \\
&\quad -\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v \check{\cdot} y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_V[\![v{:=}w]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge \varphi_1 \wedge \varphi_2 \\
&\quad -\varphi_1 = \wedge\{x \hat{\cdot} v \leftrightarrow x \check{\cdot} w \mid x \in V\backslash\{v\}\} \\
&\quad -\varphi_2 = w \check{\cdot} w \leftrightarrow v \hat{\cdot} v \\
&\quad -\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v \check{\cdot} y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_V[\![v{:=}\texttt{new}\ \kappa]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge v \hat{\cdot} v \wedge \varphi_1 \\
&\quad -\varphi_1 = (\wedge\{\neg x \hat{\cdot} v \mid x \in V \setminus \{v\}\}) \\
&\quad -\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v \check{\cdot} y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_V[\![v{:=}w.f]\!] &= \mathcal{A}^{\iota}_V[\![v{:=}w]\!] \\
\mathcal{A}^{\iota}_V[\![v.f{:=}w]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \wedge\{x \hat{\cdot} y \leftrightarrow x \check{\cdot} y \vee (x \check{\cdot} w \wedge y \check{\cdot} v) \mid x, y \in V\} \\
&\quad -\psi = \{\check{x} \leftrightarrow v \check{\cdot} x \vee \hat{x} \mid x \in V\} \\
\mathcal{A}^{\iota}_V[\![\texttt{if}\ e\ \ldots]\!] &= \mathcal{A}^{\iota}_V[\![c_1]\!] \vee \mathcal{A}^{\iota}_V[\![c_2]\!] \\
\mathcal{A}^{\iota}_V[\![c_1; c_2]\!] &= \mathcal{A}^{\iota}_V[\![c_1]\!] \circ \mathcal{A}^{\iota}_V[\![c_2]\!] \\
\mathcal{A}^{\iota}_V[\![v{:=}v_0.\mathtt{m}(v_1, \ldots, v_p)]\!] &= \phi \wedge \varphi \wedge \psi \\
&\quad \phi_m = \vee\{\iota(m) \mid m \text{ might be called }\} \\
&\quad \phi = \phi_m[s_i \mapsto v_i, out \mapsto v, this \mapsto v_0] \\
&\quad \varphi = \wedge\{x \hat{\cdot} y \leftrightarrow x \check{\cdot} y \vee \varphi_1 \mid x, y \in V\backslash\{v_0, \ldots, v_p\}\} \\
&\quad \varphi_1 = \vee\{(x \check{\cdot} v_i \wedge y \check{\cdot} v_j \wedge v_i \hat{\cdot} v_j \wedge (\check{v_i} \vee \check{v_j})) \mid i, j \in \{0, \ldots, p\}\} \\
&\quad \psi = \psi_1 \wedge (\check{v} \leftrightarrow \psi_3 \vee \psi_2(v)) \\
&\quad \psi_1 = \wedge\{\check{x} \leftrightarrow \hat{x} \vee \psi_2(x) \mid x \in V\backslash\{v, v_0, \ldots, v_p\}\} \\
&\quad \psi_2(x) = \vee\{(x \check{\cdot} v_i \wedge \check{v_i}) \mid i \in \{0, \ldots, p\}\} \\
&\quad \psi_3 = \{x \check{\cdot} y \wedge \hat{y} \mid y \in V\backslash\{v\}\}
\end{aligned}
$$

Fig. 1. Abstract Denotations over $\mathsf{SH}{\times}\mathsf{U}_V$

- $\mathcal{A}^{\iota}_V[\![v{:=}\texttt{null}]\!]$: (**SH**) sharing between $x, y \in V\backslash\{v\}$ is preserved ($\mathsf{Id}_{sh}(V\backslash\{v\})$); and nothing can share with v after C (φ_1). (**U**) $x \in V\backslash\{v\}$ is modified before C iff it is modified after C, and v is modified before C iff it shares with some y before C and y is modified after C.
- $\mathcal{A}^{\iota}_V[\![v{:=}w]\!]$: (**SH**) sharing between $x, y \in V\backslash\{v\}$ is preserved ($\mathsf{Id}_{sh}(V\backslash\{v\})$); since v becomes an alias for w then v can share with $x \in V\backslash\{v\}$ after C iff x shares with w before C (φ_1); and v can share with itself after C (i.e., not `null`) iff w shares with itself before C (φ_2). (**U**) the same as for "$v{:=}$`null`".
- $\mathcal{A}^{\iota}_V[\![v{:=}\texttt{new}\ \kappa]\!]$: the same as $\mathcal{A}^{\iota}_V[\![v{:=}\texttt{null}]\!]$ except that v shares with itself after executing the statement.
- $\mathcal{A}^{\iota}_V[\![v{:=}w.f]\!]$: the same as $\mathcal{A}^{\iota}_V[\![v{:=}w]\!]$ since the analysis is field insensitive.
- $\mathcal{A}^{\iota}_V[\![v.f{:=}w]\!]$: (**SH**) $x, y \in V$ share after C iff before C, they shared or x shared with w and y with v; (**U**) $x \in V$ is modified before C, iff it shares with v before C or x is modified after C.
- $\mathcal{A}^{\iota}_V[\![\texttt{if}\ e\ \ldots]\!]$: combines the branches through logical or.

- $\mathcal{A}^{\iota}_V[\![c_1; c_2]\!]$: combines $\mathcal{A}^{\iota}_V[\![c_1]\!]$ and $\mathcal{A}^{\iota}_V[\![c_2]\!]$. This is simply done by matching the output variables of the first denotation with the input variables of the second denotation.
- $\mathcal{A}^{\iota}_V[\![v{:=}v_0.\mathtt{m}(v_1, \ldots, v_p)]\!]$: (1) First we fetch the abstract denotations of all methods that might be called, and we combine them through logical or into ϕ_m; (2) Assuming that the method denotations use $s_i \neq v_i$ for the i-th formal parameter, we rename all sharing information by changing each s_i into v_i and *out* into v. We get ϕ. (3) We add sharing information for variables which are not in $V \backslash \{v, v_0, \ldots, v_p\}$. The sharing component φ states that x and y might share after the call iff they shared before (i.e. $x\check{\cdot}y$) or they shared with arguments v_i and v_j where v_i and v_j share after the call, and either v_i or v_j has been modified (expressed by φ_1); (4) We add the constancy information which states that $x \in V \backslash \{v\}$ is modified before iff it is modified after, or if it shares with a variable that is modified by the method. For v it is a bit different since we exclude the case that if v is modified after then it is modified before, since we possibly assign to it a new reference.

The abstract denotation for a method:

$$t\ \mathtt{m}(w_1{:}t_1, \ldots, w_n{:}t_n)\ \mathtt{with}\ w_{n+1}{:}t_{n+1}, \ldots, w_{n+m}{:}t_{n+m}\ \mathtt{is}\ com,$$

is then defined as $\phi_m = \exists V'.\ \mathcal{A}^{\iota}_V[\![com]\!] \wedge \varphi_1 \wedge \varphi_2$ where:

- $S = \{s_1, \ldots, s_n\}$ such that $S \cap m^s = \emptyset$, and $V = m^s \cup S$
- $\varphi_1 = \{\neg x\check{\cdot}y \mid x \in m^l \cup \{out\}, y \in m^s\}$
- $\varphi_2 = \{s_i\check{\cdot}x \leftrightarrow w_i\check{\cdot}x \mid 1 \leq i \leq n, x \in m^i\}$
- $V' = \{x\check{\cdot}y, x\hat{\cdot}y, \check{x}, \hat{x} \mid x \notin S \cup \{this, out\}, y \in V\} \cup \{o\check{u}t\}$

The idea is that we: (1) extend m^l to V in order to include shallow variable s_i for each method argument w_i; (2) compute $\mathcal{A}^{\iota}_V[\![com]\!]$; (3) add φ_1 which indicates that local variables are initialized to `null`; (4) add φ_2 which creates the connection between the shallow variables and the actual parameters; (5) eliminate all local information by removing the Boolean variables V'. The abstract denotational semantics can be then defined similar to the concrete one in Definition 3, where the initial method summaries are $false$ ans summaries are combined (during the fixpoint iterations) using the logical or $\vee$.

Example 2. Applying the above abstract semantics to the method defined in Example 1 results in a Boolean formula whose constancy component is $(\check{this} \leftrightarrow \hat{this}) \wedge \check{x} \wedge (\check{y} \leftrightarrow (x\check{\cdot}y \vee \hat{y}))$. For simplicity we ignore the part of ϕ_m that talks about sharing.

4 Experiments

We show here some experiments with our domain for sharing and constancy analysis. They have been performed with the JULIA analyzer [12] on a Linux

Program	M	Sharing		Non-Cyclicity	
		T	P	T	P
`JLex`	446	1595 (2324)	34.30% (34.84%)	506 (415)	34.03% (35.21%)
`JavaCup`	933	5707 (6486)	22.24% (23.76%)	853 (953)	59.23% (76.13%)
`Kitten`	2131	20976 (27824)	17.90% (19.11%)	2538 (3177)	36.34% (41.13%)
`jEdit`	3206	47408 (49356)	21.12% (21.28%)	4969 (5963)	43.49% (47.50%)
`Julia`	4028	79199 (129562)	9.71% (10.25%)	8014 (12018)	33.40% (38.17%)

Fig. 2. The effect of the purity component on Sharing and Non-Cyclicity. (M) number of methods; (T) run-time in milliseconds excluding preprocessing; (P) precision.

machine based on a 64 bits dual core AMD Opteron processor 280 running at 2.4Ghz, with 2 gigabytes of RAM and 1 megabyte of cache, by using Sun Java Development Kit version 1.5. All programs have been analyzed including all library methods that they use inside the `java.lang.*` and `java.util.*` hierarchies.

Figure 2 compares sharing analysis alone with sharing analysis in reduced product with constancy (Section 3), and its effect on non-cyclicity analysis [9]. In each column, numbers in parentheses correspond to the analysis using the reduced product. For each program, it reports the number of methods analyzed, including the libraries, and time and precision of the corresponding analysis with and without constancy. For sharing, the precision is the amount of pairs of variables of reference type that are proved not to share at the program points preceding the update of an instance field, the update of an array element or a method call. This is sensible since there is where sharing analysis is used by subsequent analyses. That figure suggests that the constancy component slightly improves the precision of sharing analysis. However, the importance of constancy is shown when we consider its effects on a static analysis that uses constancy information. This is the case of non-cyclicity analysis, which finds variables bound to non-cyclical data structures [9]. Figure 2 shows that the computation of cyclicity analysis after a simple sharing analysis leads to less precise results than the same computation after a sharing and constancy analysis. Here, precision is the number of field accesses that read the field of a non-cyclical object. This is sensible since there is where non-cyclicity is typically used.

The importance of constancy analysis becomes more apparent when it supports a static analysis that uses constancy, sharing and cyclicity information. This is the case of *path-length* [13]. It approximates the length of the maximal path of pointers one can follow from each variable. This information is the basis of a termination [1] and resource bound analyses [2] for programs dealing with dynamic data structures. Figure 3 shows the effects of constancy on path-length and termination analysis (available in [12]) of a set of small programs that do not use libraries except for `java.lang.Object`. Times are in milliseconds and precision is the number of methods proved to terminate. Constancy information

Program	M	T	P
`Init`	10	102 (140)	8 (8)
`List`	11	624 (512)	6 (11)
`Diff`	5	6668 (9040)	5 (5)
`Hanoi`	7	548 (868)	7 (7)
`BTree`	7	306 (415)	6 (7)
`BSTree`	10	234 (273)	9 (10)
`Virtual`	11	357 (418)	10 (11)
`ListInt`	11	767 (507)	6 (11)

Program	M	T	P
`Nested`	4	324 (447)	4 (4)
`Double`	5	270 (268)	5 (5)
`FactSum`	6	169 (178)	6 (6)
`Sharing`	7	309 (501)	6 (7)
`Factorial`	5	102 (196)	5 (5)
`Ackermann`	5	1308 (1732)	5 (5)
`BubbleSort`	5	871 (951)	5 (5)
`FactSumList`	8	278 (703)	7 (8)

Fig. 3. The effect of the purity information on Termination analysis. (M) number of methods; (T) run-time in milliseconds excluding preprocessing; (P) precision.

results in proving that all terminating methods terminate (only 2 methods of `Init` are not proved to terminate: they actually diverge). Without constancy information, many terminating methods are not proved to terminate.

These experiments suggest that constancy information contributes to the precision of sharing, cyclicity, path-length and hence termination analysis. Computing constancy information with sharing requires more time than computing sharing alone (Figure 2). Performing other analyses by using the constancy information increases the times further (Figures 2 and 3). Nevertheless, this is justified by the extra precision of the results.

5 Acknowledgments

This work of Samir Genaim was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project, and a *Juan de la Cierva* Fellowship awarded by the Spanish Ministry of Science and Education. The authors would like to thank the anonymous referees for their useful comments.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In Gilles Barthe and Frank de Boer, editors, *Proceedings of the IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, Lecture Notes in Computer Science, Oslo, Norway, June 2008. Springer-Verlag, Berlin. To appear.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

3. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19-20:149–197, 1994.
4. N. Cataño and M. Huisman. Chase: A Static Checker for JML's Assignable Clause. In L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proc. of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2003.
5. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.
6. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, COMPAQ Systems Research Center, 1998.
7. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML. Technical Report 96-06p, Iowa State University, 2001.
8. Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, Paphos, Cyprus, July 9–11, 2008.
9. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of Verification, Model Checking and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110, Charleston, SC, USA, January 2006.
10. A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. In R. Cousot, editor, *Proc. of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215, Paris, France, 2005. Springer.
11. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In C. Hankin, editor, *Proc. of Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335, London, UK, September 2005.
12. F. Spoto. The JULIA Static Analyser. `profs.sci.univr.it/~spoto/julia`, 2008.
13. F. Spoto, P. M. Hill, and E. Payet. Path-Length Analysis for Object-Oriented Programs. In *Proc. of Emerging Applications of Abstract Interpretation*, Vienna, Austria, March 2006. `profs.sci.univr.it/~spoto/papers.html`.
14. F. Spoto and E. Poll. Static Analysis for JML's assignable Clauses. In G. Ghelli, editor, *Proc. of FOOL-10, the 10th ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, New Orleans, Louisiana, USA, January 2003. ACM Press. Available at `www.sci.univr.it/~spoto/papers.html`.
15. G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

A Universe-Type-Based Verification Technique for Mutable Static Fields and Methods – Work in progress –

A. J. Summers[1], S. Drossopoulou[1], and P. Müller[2]
[1] Imperial College London, [2] Microsoft Research, Redmond

Abstract. We present a novel technique for the verification of invariants in the setting of a Java-like language including static fields and methods. The technique is a generalisation of the existing Visibility Technique of Müller et al., which employs universe types.
In order to cater for mutable static fields, we extend this topology to multiple trees (a forest), where each tree is rooted in a class. This allows classes to naturally own object instances as their static fields. We describe how to extend the Visibility Technique to this topology, incorporating extra flexibility for the treatment of static methods.
We encounter a potential source of callbacks not present in the original technique, and show how to overcome this using an effects system. To allow flexible and modular verification, we refine our topology with a hierarchy of 'levels'.

1 Introduction

In this paper, we extend the Visibility Technique (VT for short) [10], a known visible states verification technique based on universe types, to cater for static fields and methods. When adding statics to verification, one needs to address the following questions:

1. Where in the topology do static fields appear?
2. May instance methods update static fields?
3. May static invariants mention the fields of objects of their class?
4. May instance invariants mention static fields of their class, or of other classes?
5. Can static methods break invariants of objects, and if so, of which objects?
6. Can instance methods break static invariants, and if so, of which classes?
7. What proof obligations are necessary before a call to a static method?
8. What proof obligations are necessary before a call to an instance method?

In this paper, we explore these questions in the context of VT, and extend the technique and heap topology to handle static fields. In the process, we encounter a potential source of callbacks not present in VT, and devote much of this paper to solving this problem. We develop an approach involving a combination of effect annotations and refinements to the heap topology using levels. We then extend the technique to allow more expressive invariants.

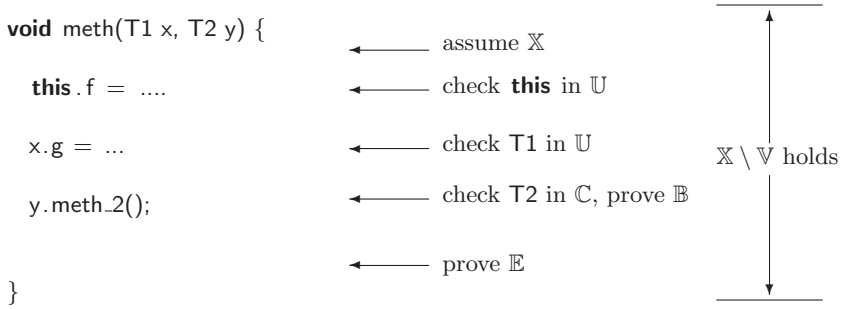


Fig. 1. Illustration of the use of the seven components.

In Sec. 2 we give the background to visible states verification techniques, universe types, and VT. In Sec. 3 we discuss the first two questions from above. In Sec. 4 we address the others, give a first attempt to an extension of VT, and argue that it is sound. We refine our approach with improved calculations of effects in Sec. 5, and with more powerful static class invariants in Sec. 6. In Sec. 7 we conclude. Proof sketches can be found in the longer version of our work, at `http://www.doc.ic.ac.uk/~ajs300m/papers/staticsFull.pdf`.

2 Background

Visible state verification techniques are defined around the notion of *visible states*, which correspond to the beginning and the end of any method call. At these visible states, the invariants of certain objects (exactly *which* objects depends on the contents of the call stack, and on the particular technique) are guaranteed to hold.

Several visible states techniques have been suggested, *e.g.*, [12, 3, 10, 8], and they share many commonalities. As suggested in [2], these commonalities, as well as the differences, can be neatly distilled in terms of the following seven components:

$\mathbb{X}$ invariants expected to hold in visible states.
$\mathbb{V}$ invariants *vulnerable* to a method, *i.e.*, which may be broken while it executes.
$\mathbb{D}$ invariants that may depend on a given heap location[1].
$\mathbb{B}$ invariants that must be proven to hold before a method call.
$\mathbb{E}$ invariants that must be proven to hold at the end of a method body.
$\mathbb{U}$ permitted receivers for field updates.
$\mathbb{C}$ permitted receivers for method calls.

The use of these components should be clear from their description above, but is also shown in Fig. 1 through annotating a method meth1: $\mathbb{X}$ may be assumed to hold in the pre- and post-states of the method. Between these visible states, some object invariants may be broken, but $\mathbb{X} \setminus \mathbb{V}$ is guaranteed to hold. Field updates and method calls are allowed if the receiver object is in $\mathbb{U}$ and $\mathbb{C}$, respectively. Before a method call, $\mathbb{B}$ must be proven. At the end of the method body, $\mathbb{E}$ must be proven. Finally, assignments to **this**.f and x.g affect at most $\mathbb{D}$.

[1] This also characterises indirectly the locations an invariant may depend on.

In [2], five *soundness conditions* are presented, and it is proven that if these conditions are satisfied, then the technique is sound (the expected invariants hold at visible states). In this paper, we use the framework of [2] informally, since the technique presented here does not quite fit the present formalism. However, the soundness conditions still guided us in the design of our technique. Informally, the five sufficient soundness conditions can be described as follows:

Definition 1 (Soundness Conditions).

1. $\mathbb{X}_{m'} \setminus (\mathbb{X}_m \setminus \mathbb{V}_m) \subseteq \mathbb{B}$
   When a legal (according to the technique, i.e.*,* $\mathbb{C}$*) call is made to a method* m' *from a method* m*, all of the invariants which are both expected to hold by the new method* ($\mathbb{X}_{m'}$)*, and are not currently known to hold in the calling method (*i.e.*, not within* $\mathbb{X}_m \setminus \mathbb{V}_m$*), must be within the proof obligations made before the method call* ($\mathbb{B}$).
2. $\mathbb{V} \cap \mathbb{X} \subseteq \mathbb{E}$
   The invariants both expected ($\mathbb{X}$) *by and vulnerable to* ($\mathbb{V}$) *a method, must be within the proof obligations at the end of the method* ($\mathbb{E}$).
3. $\mathbb{V}_{m'} \setminus \mathbb{E}_{m'} \subseteq \mathbb{V}_m$
   If a (legal) method call is made to a method m' *from a method* m*, any invariants which are vulnerable to* m' *and not reestablished by* m'*, must be vulnerable to* m.
4. $\mathbb{D} \subseteq \mathbb{V}$
   Invariants depending on fields which may be legally modified (according to the technique, i.e.*,* $\mathbb{U}$*) by a method, are vulnerable to the method.*
5. $\mathbb{X}_{c'} \subseteq \mathbb{X}_c$ *and* $(\mathbb{V}_{c'} \setminus \mathbb{E}_{c'}) \subseteq (\mathbb{V}_c \setminus \mathbb{E}_c)$
   If a method is overridden, then in the subclass version, no more invariants may be expected or left broken than in the superclass version.

One such visible states technique, the Visibility Technique (VT), was developed on top of universe types [10] with the aim to guide the verification process, and to guarantee modularity. Universe types [9] organise the heap into a tree topology, in which each object is *owned* by another object, and where an object o considers another object o', as its peer if they have the same direct owner; it considers it its rep if it is its direct owner[2]. The *owner-as-modifier discipline* (hereafter OAM) restricts field updates and method calls, implying in particular that the receivers of methods are only allowed to be reps or peers. Thus, at any time in execution any receiver on the call stack[3] is directly followed either by a rep or a peer. In Fig. 2, note that calls may only go "down" or "sideways".

The seven components from before have the following meaning for VT (we simplify slightly with respect to visibility, and to the exact class whose invariant we are considering):

[2] We do not discuss any or readonly references, nor pure methods.

[3] consisting of a sequence of activation records, each of which contains the then-current receiver

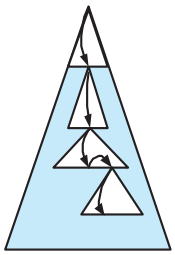


Fig. 2. Ownership Tree and Control Flow; the arrows show consecutive method calls and their receivers; note that calls go only "down", *i.e.*, to reps, or "sideways", *i.e.*, to peers. The shaded area indicates the area where objects satisfy their invariants.

𝕏 invariants of objects (reflexively, transitively) owned by peers.
𝕍 invariants of all transitive owners of the current receiver, plus invariants of peers of the current receiver.
𝔻 Invariants of peers and transitive owners may depend on the fields of an object.
𝔹 If the callee is a peer of the current receiver, then the invariants of all peers must be established. Otherwise, no proof obligations.
𝔼 the invariants of all visible peers.
𝕌 A field of an object may only be assigned to by the object's owner, or by any of its peers.
ℂ A call is allowed if the callee is a peer or rep of the current receiver.

It can be shown that these parameters satisfy the soundness conditions of Def. 1 [2]. In particular, 𝕏 and 𝕍 and the owner-as-modifier discipline, guarantee that at any given time in execution, all objects are valid, except for those directly owned by one of the receivers on the call stack, *cf.* Fig. 2.

3 Heap Topology for Static Fields

The fundamental premise of this work is that classes should be able to own objects in the same way that other objects can. For example, if the behaviour of a class depends on a static field (to manage object creation, etc.) then this static field naturally 'belongs' to the inner workings of the class: its representation. This gives a natural interpretation of static rep fields: they should be treated analogously to instance rep fields, but with a class as their owner [7].

Thus, we extend our heap topology to include classes. Classes are the 'roots' of trees in our topology. As there are generally several classes in a program, our topology should allow for several such trees; we work with a *forest*. Furthermore, with classes acting as roots, there is no longer a need for an abstract **root** entity; these class-rooted trees make up the entire picture. Note that there are no objects at the 'same level' as the class entities, and classes do not have owners. In this paper, we do not consider a notion of static peer fields.

We interpret static fields and methods as instance fields and methods of the corresponding class object. That is, the class object (or class for short) is the receiver for an execution of a static method. We expect that modifications to static fields will be achieved by calling a static method of the class that declares

the field. In other words, static methods may update the fields of their receiver class, just like instance methods in VT may update fields of their receiver object.

To summarise the ideas so far:

1. Each point in our heap topology corresponds to either an object or a class.
2. Objects (but not classes) each have exactly one owner (a class or an object).
3. The current receiver (on the stack) can be either an object or a class.

4 Basic Technique

Having defined a suitable heap topology, in this section we generalise VT to our setting.

A key aspect of our technique is that we preserve the OAM property of VT. In the following technique, control is only allowed to enter a tree in the heap topology via the 'root'; *i.e.*, by calling a static method on the class at the root of the tree. Instance method calls are restricted in the same way as in VT. This implies the following property, which will be useful for our reasoning:

Proposition 1. *A call stack (including the current method-call) always starts with a class receiver. If an object o is a receiver on the call stack, then the most recently-preceding class receiver on the call stack is the owner of the tree in which o resides.*

For the moment, we treat static invariants analogously to VT instance invariants. Therefore, they can only mention expressions which start with the static fields of the same class (since they have no peers).

How then, to handle static method calls? According to VT, a method call is only allowed if the current receiver is either the owner or a peer of the callee receiver. Since classes do not have either owners or peers, this would make static methods impossible to call. We initially considered allowing arbitrary static method calls. This immediately creates problems with callbacks; in particular, how do we know the invariants of the new receiver hold when we make the call? If our current call stack has already visited this class, we may have left invariants broken.

We solve this problem by the following rule: a static method may only be called on a class c, if c has not been a previous receiver on the call stack. However, this rule is slightly too restrictive, since it unnecessarily prohibits a static method of class c from calling another static method of class c. Our rule of thumb is:

> *A static method of c can be called if either c is the current receiver, or c is not already a receiver on the call stack.*

We are now in a position to define our technique in terms of the seven components. Compared with the description of VT, we need to extend $\mathbb{X}$ to reflect which invariants in other trees are expected, depending on the current call stack, and $\mathbb{C}$ to reflect the special rules for static method calls. The other five parameters

are straightforward generalisations of those for VT. We highlight the differences between our work and VT in italics, and point out the interpretation of these components with regard to a static method call in footnotes.

𝕏 invariants of objects (reflexively, transitively) owned by peers, *plus all invariants in trees not currently visited on the call stack*[4].

𝕍 invariants of all transitive owners of the current receiver, plus invariants of peers of the current receiver[5].

𝔻 Invariants of peers and transitive owners may depend on the field of an object *or class*[6].

𝔹 If the callee is a peer of the current receiver, then the invariants of all peers must be established. Otherwise, no proof obligations[7].

𝔼 the invariants of all visible peers[8].

𝕌 A field of an object *or class* may only be assigned to by its owner, or by any of its peers[9].

ℂ A call to an instance method is allowed if the callee is a `peer` or `rep` of the current receiver. *A call to a static method m on class c is allowed if either the current receiver is c itself, or else c is not on the current call stack.*

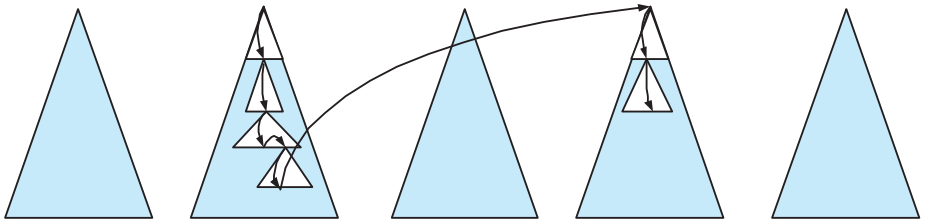


Fig. 3. Calls stacks across several trees, invariants hold in shaded areas.

When considering only the tree of the current receiver, the rules are essentially those of VT. However, the other trees either have none of their invariants expected, or all of them, depending on whether or not they have been visited on the current call stack. Furthermore, static methods are treated differently from

[4] For a static method, this amounts to all the invariants of the current tree, plus each unvisited tree.

[5] The only invariants vulnerable to a call of a static method in class c are the static invariants of c itself.

[6] The only invariants which are allowed to depend on a static field declared in class c are the static invariants of c.

[7] If a static method is called on a class c which is both caller and callee (a 'self' call), then the static invariants of c must be reestablished first.

[8] For a static method, the invariants of the class.

[9] A static field can only be assigned to by the class itself.

instance method calls, in that any call is permitted so long as the callee has not been a receiver prior to the current one on the call stack.

Since $\mathbb{C}$ depends on the current call stack, it is not possible to statically verify whether a method call will be legal. We therefore identify next a way of conservatively approximating when method calls are legal.

Effect Annotations. For each class c and method m, we require a set of *effects*, $\mathcal{E}ffs(c, m)$, predicting which classes may have static methods called on them as a result of calling m of c. $\mathcal{E}ffs(c, m)$ is a (possibly empty) set of class names. This is described by requirements 1-3 in Def. 2 below.

If, from within the body of a static method m of class c, we make a call to a (static or instance) method m' defined in class c' (with a different receiver), and if this method call may eventually result in a callback to c, then as a consequence of Def. 2, we must have $c \in \mathcal{E}ffs(c', m')$. Therefore, we can rule out dangerous callbacks on c by insisting that any method which is called from a static method of c does not contain c in its effects. This is described through the method restriction in item 4 of Def. 2.

Definition 2 (Valid Effects and Method Restrictions).

1. *Within the body of a method m of class c, if there is a call $e.m'(\ldots)$ and e has static type c', then $\mathcal{E}ffs(c', m') \subseteq \mathcal{E}ffs(c, m)$.*
2. *Within the body of a method m of class c, if there is a call $c'.m'(\ldots)$ to a static method m' of class c', then*
   (a) *$\mathcal{E}ffs(c', m') \subseteq \mathcal{E}ffs(c, m)$ and*
   (b) *if m is an instance method or $c \neq c'$ [10], then $c' \in \mathcal{E}ffs(c, m)$.*
3. *If c' is a subclass of c which overrides a method m, then $\mathcal{E}ffs(c', m) \subseteq \mathcal{E}ffs(c, m)$.*
4. *A static method m of c is legal, only if $c \notin \mathcal{E}ffs(c, m)$.*

Soundness. We focus on the first item from Def. 1: the guarantee that when a method call is made, the invariants expected in the new method will hold (because they have been preserved, or proven before the call is made). We claim that the other points can be easily established.

In the technique presented, all invariants may only depend on the fields of peers (if any) and any objects transitively owned. Furthermore, fields may only be modified by peers. Therefore, we have the following property:

Proposition 2 (Broken Invariants). *If, at runtime, the invariants of an object (or class) do not hold, then one of the receivers on the call stack (possibly the current one) must be the object (or class) itself or one of its peers.*

[10] *i.e.*, a static method always may call another static method from the same class.

To demonstrate that our restrictions using effects (Def. 2) are sufficient to guarantee that our desired notion of valid method call ($\mathbb{C}$) is always adhered to, we need a deeper discussion of possible sequences of calls. We require some notation to capture these sequences; we wish to track the receiver-method pairs from (consecutive) fragments of the call stack. We write (c, m) for a call of static method m on class c, and (o, m) for a call of instance method m on object o. For any receivers r,r' (which may each be either classes or objects), we write $(r, m)\ \overline{call}\ (r', m')$ to denote a sequence of legal calls[11] beginning with m and ending with m', *i.e.*, method m on receiver r calls some method m_1 on some receiver r_1, etc., which eventually leads to calling method m' on receiver r'. These sequences of calls correspond to consecutive regions of a call stack, in which only the receiver and method information is retained. Note that such sequences need not begin from the initial (class) receiver of a call stack. We consider only call-sequences which are legal according to our technique.

We can now show how calls are restricted by the effect annotations:

Proposition 3 (Effects are Conservative).

1. *For any call-sequence* $(o, m)\ \overline{call}\ (c', m')$*, if* c *is the dynamic class of* o*, then* $c' \in \mathcal{E}ffs(c, m)$*.*
2. *For any call-sequence* $(c, m)\ \overline{call}\ (c', m')$*, if either* $c \neq c'$ *or any of the intermediate receivers in* $\overline{call}$ *are not* c*, then* $c' \in \mathcal{E}ffs(c, m)$*.*
3. *Any call-sequence* $(c, m)\ \overline{call}\ (c, m')$ *consists only of calls where* c *is the receiver.*
4. *If* o *and* o' *are peers, then any call-chain* $(o, m)\ \overline{call}\ (o', m')$ *features only peers of* o *(and* o'*) as receivers.*

Finally, we can prove that the invariants of a new receiver are always guaranteed by the proof obligations in the technique:

Theorem 1.

1. *If a static method* m *is to be called on* c*, then the proof obligations imposed by the technique guarantee that* c*'s invariants hold.*
2. *If an instance method* m *is to be called on* o*, then the proof obligations imposed by the technique guarantee that* o*'s invariants hold.*

5 Refined Effects

The effects as described so far require annotations for *all* classes used in a program. This requirement leads to a high annotation burden, compomises information hiding, and limits the usability of the technique presented so far, as the following example illustrates.

[11] *i.e.*, calls which are permissible according to Def. 2.

Example 1 (Method Overriding and Effects). Consider the String class of the Java API. An implementation of this class can exploit that fact that strings are immutable in Java, and so share instances of objects, by using static fields from class String to maintain a 'pool' of used String instances. This would imply that the constructor String calls String static methods, and would have String in its effects. Consider now that we want to write a class which overrides the equals() method inherited from Object:

```
class MyClass extends Object{
  boolean equals(Object o)
  {
    System.out. println (new String("equals() called"));
    return this == o;
  }
}
```

Obviously, we need to have String$\in \mathcal{E}ffs$(MyClass,equals), and because of Def. 2 (item 3), we also need that String$\in \mathcal{E}ffs$(Object,equals). But, it is unlikely that this effect was predicted when the class Object was given effect annotations. Therefore, this method definition would be illegal. This illustrates an annotation problem (annotations may need recomputing), an information-hiding problem (our code should not need to know how String is implemented), and a usability problem (our technique forbids this method declaration).

To alleviate this burden, we introduce a refinement, whereby we group classes in a linear hierarchy of 'levels', such that the code of lower-level classes does not mention the higher-level classes[12]. The intuition is that library classes should have been previously verified and belong on a 'lower level' than the classes which the programmer is now writing. We express the levels through a function $\mathcal{L}vl(\_)$ which maps classes to integers.

Definition 3 (Valid Levels). *c mentions* $c' \Rightarrow \mathcal{L}vl(c) \geq \mathcal{L}vl(c')$.

Because classes in the lower levels do not 'know about' classes in the upper levels, it is impossible for them to make static calls on the classes in the upper levels (*cf.* Fig. 4). Therefore, if we consider verification of the topmost level, then when a call is made down to a lower level, the effect annotations are no longer necessary. [13] Thus, we refine our effect annotation sets to only mention classes on the same level as the method being verified. The new conditions on effects (in which differences in comparison with Def. 2 are shown in roman font) are:

Definition 4 (Refined Effects).

[12] For example, we could consider the Java API classes (*e.g.*, Object and String) to be on a lower level than our classes, and it would be naturally guaranteed that the API classes do not mention ours.

[13] To handle dynamic binding, we require the effects of methods that override methods in lower levels to be empty and, thus, independent of the effects of the overridden method.

1. If c' is in $\mathcal{E}ffs(c,m)$ then $\mathcal{L}vl(c') = \mathcal{L}vl(c)$.
2. *Within the body of a method m of class c, if there is a call $e.m'(\ldots)$ and e has static type c',* and $\mathcal{L}\mathbf{vl}(\mathbf{c}) = \mathcal{L}\mathbf{vl}(\mathbf{c}')$, *then* $\mathcal{E}ffs(c',m') \subseteq \mathcal{E}ffs(c,m)$.
3. *Within the body of a method m of class c, if there is a call $c'.m'(\ldots)$ to a static method m' of class c'* and $\mathcal{L}\mathbf{vl}(\mathbf{c}) = \mathcal{L}\mathbf{vl}(\mathbf{c}')$, *then:*
   (a) $\mathcal{E}ffs(c',m') \subseteq \mathcal{E}ffs(c,m)$
   (b) *if m is either an instance method or $c \neq c'$, then $c' \in \mathcal{E}ffs(c,m)$.*
4. *If c' is a subclass of c which overrides a method m, then*
   (a) If $\mathcal{L}\mathbf{vl}(\mathbf{c}) = \mathcal{L}\mathbf{vl}(\mathbf{c}')$, then $\mathcal{E}ffs(c',m) \subseteq \mathcal{E}ffs(c,m)$
   (b) If $\mathcal{L}\mathbf{vl}(\mathbf{c}) < \mathcal{L}\mathbf{vl}(\mathbf{c}')$, then $\mathcal{E}\mathbf{ffs}(\mathbf{c}',\mathbf{m}) = \emptyset$
5. *A static method m of c is legal, only if $c \notin \mathcal{E}ffs(c,m)$.*

The refined conditions given permit smaller effects sets for methods than those of Def. 2. Considering the example at the start of the section, it is no longer necessary (or indeed, allowed) for String to be in $\mathcal{E}ffs$(MyClass,equals).

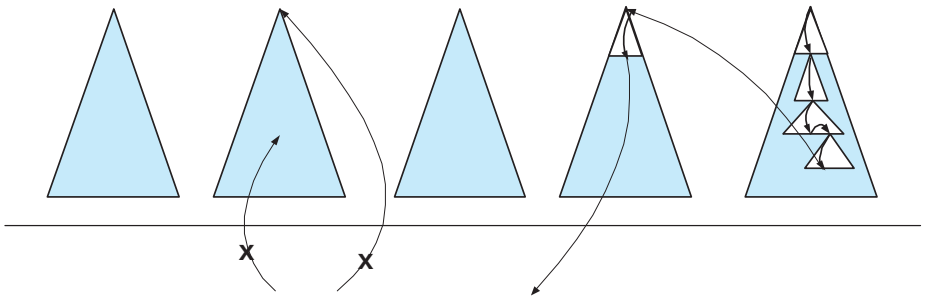


Fig. 4. Trees in one level. The current level may call into the lower level, but no calls from the lower level may come into the current level. The level of an object is determined by the class that transitively owns the object, not by the object's type.

Soundness. As in the previous section, we focus on ensuring that the proof obligations made before method calls are always sufficient to guarantee the expected invariants. Furthermore, we make the assumption here that we are only interested in verifying the 'top-level'; we assume that the classes on lower levels have already been verified. This can be used to construct an inductive verification of the entire class-structure, if needed, but also allows us a more-modular approach; once the classes on a lower level have been verified, we need not repeat the process if we are only adding classes to higher levels.

We write $\mathcal{L}vl(o)$ for the level of an object, defined to be the level of the class which transitively owns the object (*i.e.*, the class which is the 'root' of the appropriate tree). We can then show the following property:

Proposition 4 (Levels do not Increase through Calls).

1. *If object o is transitively owned by class c, and if c' is the dynamic class of o, then $\mathcal{L}vl(c) \geq \mathcal{L}vl(c')$.*
2. *For any call-sequence (c, m) $\overline{call}$ (o, m'), where $\overline{call}$ consists exclusively of instance method calls, if c' is the dynamic class of o, then $\mathcal{L}vl(c) \geq \mathcal{L}vl(c')$.*
3. *For any sequence of calls (r_1, m_1) $\overline{call}$ (r_2, m_2), in which r_1, r_2 can be any receivers,* i.e.*, classes or objects, we have $\mathcal{L}vl(r_1) \geq \mathcal{L}vl(r_2)$.*
4. *For any call-sequence (c, m) $\overline{call}$ (c, m'), for all the intermediate receivers r, we have $\mathcal{L}vl(r) = \mathcal{L}vl(c)$.*

This allows us to construct similar arguments to those in the previous section, regarding soundness of method calls. Proposition 2 still holds for this refinement. Proposition 3 holds in the restricted case that all receivers involved are from the top-level. Theorem 1 then holds for all such receivers.

Remarks. We have allowed the organisation of levels to be very flexible, and thus the effects and levels can be used to complement each other in various different ways. Considering the extreme case of only one level, we return to our original effects proposal from the previous section, in which all the work must be done by the effects. On the other hand, if every class has a level to itself, we essentially impose a total ordering on classes (which may not be possible within our restrictions, for all programs), and no effect annotations are required at all. In practice, we envisage that the levels will be used to separate away previously written library classes from those being currently developed and verified.

6 Extended Technique

So far, static invariants cannot mention the fields of instance objects, and instance invariants cannot mention static fields. It seems reasonable to question whether this is enough. For example, if we wished to write a class MyThread in which each instance object was assigned a unique identifier id, we might like an invariant to express that distinct MyThread objects have different ids[14]. These kinds of invariants involve both static fields and instance fields. It is desirable to extend our technique to handle these more-expressive invariants. We could allow instance invariants to mention static fields (of the same class, and perhaps superclasses) in their invariants. The alternative approach is, instead of enriching instance invariants, to enrich *static* invariants with the ability to quantify over *all* instances of a class. In fact, any instance invariant mentioning static fields can always be expressed as a static invariant by adding a quantified object to replace all the mentions of **this**. However, enriching static invariants in this way can be more general if we allow multiple quantifiers. If we wanted to express the described invariant of MyThread, we could do so by the static invariant forall MyThread o_1, o_2: $o_1 \neq o_2 \Rightarrow o_1.\mathsf{id} \neq o_2.\mathsf{id}$. However, it is not clear how to express this at the level of an instance invariant (without quantifiers).

[14] This is an actual invariant of the Thread class in the Java API.

We choose to add the ability to quantify over fields of instances in static invariants. In static invariants of class c, if o is a quantified object variable, the only fields of o which may be mentioned in the invariants are those declared in class c. This restriction corresponds to the notion of *subclass separation* described for VT (see [10] for details).

Remark. Although it is true that any instance invariant mentioning static fields can be encoded as a static invariant quantifying over instances, this does not quite mean the two are interchangeable with respect to our technique. The reason is that although these invariants express the same properties, because one is an invariant per object, and one is an invariant of the class, they will be expected to hold at different times.

To work out exactly what changes were needed to our technique in order to retain soundness, we were guided by the soundness conditions of [2] (*cf.* Def. 1). Essentially, having made a change to our $\mathbb{D}$ parameter (by changing which invariants can depend on instance fields), the conditions presented there implied the minimal necessary changes to the other parameters of our technique in order to restore soundness. We highlight the differences between the new and the previous technique through the use of italics.

$\mathbb{X}$ invariants of objects (reflexively, transitively) owned by peers, plus all invariants in trees not currently visited on the call stack.

$\mathbb{V}$ invariants of all transitive owners of the current receiver, plus invariants of peers of the current receiver, *and their classes.*

$\mathbb{D}$ Invariants of peers and transitive owners may depend on the field of an object or class. *Additionally, static invariants of the class in which the field is declared.*

$\mathbb{B}$ Before making a method call, *the invariants of the classes of all of the peers of the current receiver must be established.* Furthermore, if the callee is a peer of the current receiver, then the invariants of all peers must be established.

$\mathbb{E}$ the invariants of all visible peers, *and their classes.*

$\mathbb{U}$ A field of an object (or class) may only be assigned to by its owner, or by any of its peers.

$\mathbb{C}$ A call to an instance method is allowed if the callee is a `peer` or `rep` of the current receiver. A call to a static method m on class c is allowed if either the current receiver is c itself, or else c is not on the current call stack.

Soundness. Informally, the soundness of this extended technique follows from the soundness of the previously-presented versions, as follows:

Proposition 2 no longer holds. Namely, because of the extended language of invariants in this new version of our technique, it is possible for many more methods to cause such invariants to break. However, our technique does *not* allow these invariants to remain broken in any more visible states than was previously allowed. Essentially, any invariants which are broken due to the quantification over instances now possible, will always be reestablished at the next visible state (either the end of the method call, or before the next method call; whichever is

the sooner). This is reflected in our $\mathbb{B}$ and $\mathbb{E}$ defined above. Therefore, although Proposition 2 does not hold, Theorem 1 can still be proved, essentially because enough extra proof obligations are imposed before a method call takes place.

7 Conclusions, Related Work, and Future Work

We have outlined a verification technique based on VT, catering for static fields, methods, and invariants. In the process, we extended the usual heap topology of ownership types, and tackled potential callbacks through a combination of effects, levels, and the OAM discipline.

Universe types as implemented in JML [5] require static fields to be readonly. JML's static invariants may only refer to static fields, while instance invariants may refer to both static and instance fields [6, Sec. 8.2]. In JML and in our work, both instance and static invariants are supposed to hold in visible states [10]. In JML's universe types, static methods are executed relative to the context of the object who called the static method. This allows one to implement static factory methods, which create new objects in the context of their caller. We can extend our approach to support factory methods by incorporating *ownership transfer* [11], allowing a method to create a new object, but to postpone the decision of assigning it an owner.

In [7], Leino and Müller extend the Boogie methodology [1] to static invariants: static fields may be reps; class invariants may mention static rep fields and also quantify over objects of their class. The callback problem is solved by making explicit the state in which static invariants may be assumed to hold, and by enclosing expressions that potentially break the static invariant of a class in expose blocks. In order to support abstraction in method specifications, a *validity ordering* is used to allow a class to implicitly expect the static invariants of 'smaller' classes. This issue is similar to one of the motivations for introducing our levels. The validity ordering, however, has the side-effect for static initialisation that subclasses be initialised before superclasses.

In Jacobs et al.'s work [4], Spec# annotations are suggested to cater for local reasoning in the presence of multithreading. Again, static fields may be reps, and static invariants may depend on the (transitively) owned objects. Both our system and theirs need to address potential circularities: ours in order to avoid visiting classes in an inconsistent state, and [4] in order to prevent deadlocks. They require a partial ordering of locks, which, in a way, corresponds to our levels. Two locks on the 'same level' are not allowed to be consecutively acquired. In contrast, we permit method calls between classes on the same level, if the effects allow it. Our work may be seen as the visible-states-based counterpart of [4, 7].

We have not discussed static initialisation in this paper. In brief, we expect to be able to incorporate the Java semantics for static initialisation. In terms of our topology, initialisation is best modelled by considering that the tree owned by a class comes into existence at the moment static initialisation of the class begins (and is initially empty, apart from the owning class). Static initialisers may

assume all of the invariants of lower levels, and no others (since the restrictions on method calls are not respected by the execution of static initialisers). Exploring these issues in more detail will be the subject of future work. We also plan to complete the formal presentation of our work, and to study class visibility, modularity, readonly fields, pure methods, and factory methods.

Acknowledgements. This paper has been greatly improved by comments and generous feedback from Adrian Francalanza and the (anonymous) FTfJP reviewers. We are also grateful to Nick Cameron, Werner Dietl, and Jayshan Raghunandan for discussions on static methods and verification. This work was funded in part by the IST-2005-015905 MOBIUS project.

References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, LNCS. Springer, 2005.
2. S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, 2008.
3. K. Huizing and R. Kuiper. Verification of object-oriented programs using class invariants. In *FASE*, volume 1783 of *LNCS*, pages 208–221. Springer, 2000.
4. B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. *Electronic Notes on Theoretical Computer Science special issue on Thread Verification (TV06),*, 174:23–47, 2007.
5. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual—section on Universe annotations, February 2007. `www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_18.html#SEC205`.
6. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Available from `http://www.jmlspecs.org`, May 2008.
7. K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *Formal Methods*, 2005.
8. Y. Lu, J. Potter, and J. Xue. Object Invariants and Effects. In *ECOOP*, volume 4609 of *LNCS*, pages 202–226. Springer, 2007.
9. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.
10. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
11. P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 461–478. ACM, 2007.
12. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.

Subtyping Existential Types

Stefan Wehr and Peter Thiemann

Institut für Informatik, Universität Freiburg
{wehr,thiemann}@informatik.uni-freiburg.de

Abstract Constrained existential types are a powerful language feature that subsumes Java-like interface and wildcard types. But existentials do not mingle well with subtyping: subtyping is already undecidable for very restrictive settings. This paper defines two subtyping relations by extracting the features specific to existentials from current language proposals (JavaGI, WildFJ, Scala). Both subtyping relations are undecidable. The paper also discusses the consequences of removing existentials from JavaGI and possible amendments to regain their features.

1 Introduction

Constrained existential types (also called "bounded existential types" [5, 12]) arise from the need for structured and partial data abstraction and information hiding. They have found uses for modeling object oriented languages in general [2], as well as for modeling specific features such as Java wildcards [3,4,18,19] and Java-like interface types in the JavaGI language [21]. In fact, JavaGI supports general existential types and provides interface types as a special case supported by syntactic sugar. Building directly on existential types has several advantages compared to interface types: they allow the general composition of interface types, they encompass Java wildcards, and they enable meaningful types in the presence of multi-headed interfaces.[1]

Work on the type checker for an implementation of JavaGI uncovered the consequences of supporting general existential types. They do not only introduce a wealth of complexity into the type system (something we can live with) but they may also cause nontermination in the type checker (something we cannot live with): JavaGI's subtyping relation with existential types is undecidable.

After establishing some background on JavaGI (§ 2), we define two calculi with constrained existential types and subtyping. The first calculus (§ 3) is a subset of JavaGI's formalization [21]. The second calculus (§ 4) supports existential types with lower and upper bounds, very much like Scala [13] and formal systems for modeling Java wildcards [3, 4, 18]. We prove that the subtyping relations of both calculi are undecidable.

Furthermore, we discuss alternative design options for JavaGI that avoid the use of general existential types but keep the remaining features (§ 5). Finally, we review related work (§ 6) and conclude (§ 7). Detailed proofs can be found in an accompanying technical report [22].

[1] JavaGI provides multi-headed interfaces that abstract over a family of types.

2 Background

JavaGI [21] is a conservative extension of Java 1.5 that generalizes Java's interface concept to incorporate the essential features of Haskell type classes [8,9,20]. This generalization allows for retroactive and constrained interface implementations, binary methods, static methods in interfaces, default implementations for interface methods, and multi-headed interfaces (interfaces over families of types). Furthermore, JavaGI generalizes Java-like interface types to existential types. This section only discusses the features relevant to this paper, namely retroactive interface implementations and existential types, and ignores the rest.

2.1 Retroactive Interface Implementations

A class definition in Java must specify all interfaces that the class implements. In contrast, JavaGI enables programmers to add implementations for interfaces to existing classes at any time. For example, Java rejects the use of a `for`-loop to iterate over the characters of a string because the class `String` does not implement the interface `Iterable`:[2]

```
for (Character c : someString) { ... } // illegal in Java
```

As the definition of class `String` is fixed, there is no hope of getting this code working. In contrast, JavaGI allows the retroactive implementation of `Iterable`:[3]

```
implementation Iterable<Character> [String] {
  public Iterator<Character> iterator() {
    return new Iterator<Character>() {
      private int index = 0;
      public boolean hasNext() { return index < length(); }
      public Character next() { return charAt(index++); }
    };
  }
}
```

This *implementation definition* specifies that the *implementing type* `String`, enclosed in square brackets `[]`, implements the interface `Iterable<Character>`. The definition of the `iterator` method can use the methods `length` and `charAt` because they are part of `String`'s public interface.

2.2 Constrained Existential Types

Java uses the name of an interface as an *interface type* that denotes the set of all types implementing the interface. Instead of interface types, JavaGI features *constrained existential types* (*existentials* for short) and provides syntactic sugar for recovering interface types. For example, the interface type `List<String>` abbreviates the existential type ∃ `X where X implements List<String> . X`. The *implementation constraint* "`X implements List<String>`" restricts instantiations

[2] Java's enhanced `for`-loop allows to iterate over arrays and all types implementing the `Iterable<X>` interface, which contains a single method `Iterator<X> iterator()`.

[3] We ignore the `remove` method of the `Iterator` interface.

of the type variable `X` to types that implement the interface `List<String>`. Thus, the existential type denotes the set of all types implementing `List<String>`, exactly like the synonymous interface type.

Existentials are more general than interface types. For instance, the existential `∃ X where X implements List<String>, X implements Set<String> . X` denotes the set of all types that implement both `List<String>` and `Set<String>`. Java supports such intersections of interface types only for specifying bounds of type variables. Existentials also encompass Java wildcards [3, 4, 18, 19]. For instance, the existential type `∃ X where X extends Number . List<X>` corresponds to the wildcard type `List<? extends Number>`.[4]

JavaGI allows implementation definitions for existentials. For example, a programmer may write an implementation definition to specify that all types implementing `List<X>` also implement `Iterable<X>`. Such a definition is feasible because iterators can be implemented using only operations of the `List` interface. The example also demonstrates that JavaGI supports *generic implementation definitions*, which are parameterized by type variables.

```
implementation<X> Iterable<X> [∃ L where L implements List<X> . L] {
  Iterator<X> iterator() { return new Iterator<X>() {
      /* as for String, replacing length with size and charAt with get. */ };
  }
}
```

A Java programmer would have to implement `Iterable` from scratch for every class that implements `List`. Abstract classes do not help with this problem because Java does not support multiple inheritance.

3 Subtyping Existential Types with Implementation Constraints

This section introduces $\mathcal{EX}_{impl}$, a subtyping calculus with existentials and implementation constraints. $\mathcal{EX}_{impl}$ is a subset of Core–JavaGI from the original formulation of JavaGI's type system [21]. It does not model all aspects of JavaGI, but contains only those features that make subtyping undecidable.

3.1 Definition of $\mathcal{EX}_{impl}$

Fig. 1 defines the syntax, as well as the entailment and subtyping relations of $\mathcal{EX}_{impl}$. A type T is either a type variable X or an existential $\exists X \texttt{ where } \overline{P} \,.\, X$. For simplicity, there are no class types, existentials have a single quantified type variable, and the body of an existential must be the quantified type variable.[5] Overbar notation $\overline{\xi}$ denotes a sequence $\xi_1, \ldots, \xi_n$ of syntactic entities with $\bullet$ standing for the empty sequence. Sometime, the sequence $\overline{\xi}$ stands for the set $\{\overline{\xi}\}$. Existentials are considered equal up to renaming of bound type variables, reordering of constraints, and elimination of duplicate constraints.

[4] Because `List` is an interface, `∃ X where X extends Number . List<X>` stands for `∃ X,L where X extends Number, L implements List<X> . L`

[5] The body of an existential is the part after the ".".

$$T, U, V, W ::= X \mid \exists X \,\texttt{where}\, \overline{P}\,.\, X$$
$$P, Q, R ::= X \,\texttt{implements}\, I\texttt{<}\overline{T}\texttt{>}$$
$$def ::= \texttt{interface}\ I\texttt{<}\overline{X}\texttt{>} \mid \texttt{implementation<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\ \texttt{[}T\texttt{]}$$

$$\text{E}_1\text{-IMPL}\ \frac{\texttt{implementation<}\overline{X}\texttt{>}\ I\texttt{<}\overline{T}\texttt{>}\ \texttt{[}U\texttt{]} \in \Theta}{\Theta; \Delta \Vdash [\overline{V/X}]\,(U \,\texttt{implements}\, I\texttt{<}\overline{T}\texttt{>})} \qquad \text{E}_1\text{-LOCAL}\ \frac{P \in \Delta}{\Theta; \Delta \Vdash P} \qquad \text{S}_1\text{-REFL}\ \Theta; \Delta \vdash T \leq T$$

$$\text{S}_1\text{-TRANS}\ \frac{\Theta; \Delta \vdash T \leq U \qquad \Theta; \Delta \vdash U \leq V}{\Theta; \Delta \vdash T \leq V} \qquad \text{S}_1\text{-OPEN}\ \frac{\Theta; \Delta, \overline{P} \vdash X \leq T \qquad X \notin \mathsf{ftv}(\Theta, \Delta, T)}{\Theta; \Delta \vdash \exists X \,\texttt{where}\, \overline{P}\,.\, X \leq T}$$

$$\text{S}_1\text{-ABSTRACT}\ \frac{(\forall i)\ \Theta; \Delta \Vdash [T/X]P_i}{\Theta; \Delta \vdash T \leq \exists X \,\texttt{where}\, \overline{P}\,.\, X}$$

Fig. 1. Type syntax, entailment, and subtyping for $\mathcal{EX}_{impl}$.

An implementation constraint P has the form X `implements` $I\texttt{<}\overline{T}\texttt{>}$ and constrains the type variable X to types that implement the interface $I\texttt{<}\overline{T}\texttt{>}$. In comparison with upper bounds for type variables, implementation constraints allow more precise typings, especially for binary methods [1]. An interface without type parameters is written I instead of $I\texttt{<}\bullet\texttt{>}$.

A definition *def* in $\mathcal{EX}_{impl}$ is either an interface or an implementation definition. Interface and implementation definitions do not have method signatures or bodies, because they do not matter for the entailment and subtyping relation of $\mathcal{EX}_{impl}$. Moreover, $\mathcal{EX}_{impl}$ does not support interface inheritance. A program environment Θ is a finite set of definitions *def*, and a type environment Δ a finite set of constraints P, where Δ, P abbreviates $\Delta \cup \{P\}$.

The entailment relation $\Theta; \Delta \Vdash T \,\texttt{implements}\, I\texttt{<}\overline{T}\texttt{>}$ expresses that type T implements interface $I\texttt{<}\overline{T}\texttt{>}$. A type implements an interface either because it corresponds to an instance of a suitable implementation definition (rule E$_1$-IMPL) or because the type environment contains the constraint (rule E$_1$-LOCAL). The notation $[\overline{T/X}]$ stands for the capture-avoiding substitution replacing each X_i with T_i. Full JavaGI uses the entailment relation (among other things) to verify that the instantiation of a generic class or method fulfills the implementation constraints associated with that class or method.

The subtyping relation $\Theta; \Delta \vdash T \leq U$ states that T is a subtype of U. It is reflexive and transitive as usual. Rule S$_1$-OPEN opens an existential on the left-hand side of a subtyping judgment by moving its constraints into the type environment. The premise $X \notin \mathsf{ftv}(\Theta, \Delta, T)$ ensures that the existentially quantified type variable is sufficiently fresh and does not escape from its scope. Rule S$_1$-ABSTRACT deals with existentials on the right-hand side of a subtyping judgment. It states that T is a subtype of some existential if all constraints of the existential hold after substituting T for the existentially quantified type variable.

While developing a type soundness proof for Core–JavaGI, we verified that the subtyping relation of $\mathcal{EX}_{impl}$ supports the usual principle of subsumption: we can always promote the type of an expression to some supertype without causing runtime errors.

3.2 Undecidability of Subtyping in $\mathcal{EX}_{impl}$

We prove undecidability of subtyping in $\mathcal{EX}_{impl}$ by reduction from Post's Correspondence Problem (PCP). It is well known that PCP is undecidable [7, 17].

Definition 1 (PCP). *Let $\{(u_1, v_1) \ldots, (u_n, v_n)\}$ be a set of pairs of non-empty words over some finite alphabet Σ with at least two elements. A solution of PCP is a sequence of indices $i_1 \ldots i_r$ such that $u_{i_1} \ldots u_{i_r} = v_{i_1} \ldots v_{i_r}$. The decision problem asks whether such a solution exists.*

Theorem 1. *Subtyping in $\mathcal{EX}_{impl}$ is undecidable.*

Proof. Let $\mathcal{P} = \{(u_1, v_1), \ldots, (u_n, v_n)\}$ be a particular instance of PCP over the alphabet Σ. We can encode $\mathcal{P}$ as an equivalent subtyping problem in $\mathcal{EX}_{impl}$ as follows. First, words over Σ must be represented as types in $\mathcal{EX}_{impl}$.

```
interface E       // empty word ε
interface L<X>    // letter, for every L ∈ Σ
```

Words $u \in \Sigma^*$ are formed with these interfaces through nested existentials. For example, the word `AB` is represented by

$$\exists X\,\texttt{where}\,X\,\texttt{implements}\,\texttt{A<}\exists Y\,\texttt{where}\,Y\,\texttt{implements}\,\texttt{B<}\exists Z\,\texttt{where}\,Z\,\texttt{implements}\,\texttt{E}\,.\,Z\texttt{>}\,.\,Y\texttt{>}\,.\,X$$

The abbreviation $\exists I\texttt{<}\overline{T}\texttt{>}$ stands for the type $\exists X\,\texttt{where}\,X\,\texttt{implements}\,I\texttt{<}\overline{T}\texttt{>}\,.\,X$. Using this notation, the word `AB` is represented by $\exists\texttt{A<}\exists\texttt{B<}\exists\texttt{E>>}$.

Formally, we define the representation of a word u as $[\![u]\!] = u \,\#\, \exists\texttt{E}$, where $u \,\#\, T$ is the concatenation of a word u with a type T:

$$\varepsilon \,\#\, T \triangleq T \qquad\qquad Lu \,\#\, T \triangleq \exists L\texttt{<}u \,\#\, T\texttt{>}$$

Two interfaces are required to model the search for a solution of PCP:

```
interface S<X,Y>    // search state
interface G         // search goal
```

The type $\exists\texttt{S<}[\![u]\!], [\![v]\!]\texttt{>}$ represents a particular search state where we have already accumulated indices $i_1, \ldots, i_k$ such that $u = u_{i_1} \ldots u_{i_k}$ and $v = v_{i_1} \ldots v_{i_k}$. To model valid transitions between search states, we define implementations of `S` for all $i \in \{1, \ldots, n\}$ as follows:

```
implementation<X,Y> S<u_i#X, v_i#Y> [∃ S<X,Y>]        (1)
```

The type $\exists\texttt{G}$ represents the goal of a search, as expressed by the following implementation:

```
implementation<X> G [∃ S<X,X>]                        (2)
```

To get the search running we ask whether there exists some $i \in \{1, \ldots, n\}$ such that $\Theta_{\mathcal{P}}; \emptyset \vdash \exists\texttt{S<}[\![u_i]\!], [\![v_i]\!]\texttt{>} \leq \exists\texttt{G}$ is derivable. The program $\Theta_{\mathcal{P}}$ consists of the interfaces and implementations just defined. In our technical report [22], we prove that the given PCP instance $\mathcal{P}$ has a solution if and only if there exists some $i \in \{1, \ldots, n\}$ such that $\Theta_{\mathcal{P}}; \emptyset \vdash \exists\texttt{S<}[\![u_i]\!], [\![v_i]\!]\texttt{>} \leq \exists\texttt{G}$ is derivable. □

$$N, M ::= C\texttt{<}\overline{X}\texttt{>} \mid \texttt{Object}$$
$$T, U, V, W ::= X \mid N \mid \exists \overline{X}\,\texttt{where}\,\overline{P}\,.\,N$$
$$P, Q, R ::= X\,\texttt{extends}\,T \mid X\,\texttt{super}\,T$$

$$\text{E}_2\text{-EXTENDS}\ \frac{\Delta \vdash T \leq U}{\Delta \Vdash T\,\texttt{extends}\,U} \qquad \text{E}_2\text{-SUPER}\ \frac{\Delta \vdash U \leq T}{\Delta \Vdash T\,\texttt{super}\,U}$$

$$\text{S}_2\text{-REFL}\ \Delta \vdash T \leq T \qquad \text{S}_2\text{-TRANS}\ \frac{\Delta \vdash T \leq U \qquad \Delta \vdash U \leq V}{\Delta \vdash T \leq V}$$

$$\text{S}_2\text{-OBJECT}\ \Delta \vdash T \leq \texttt{Object} \qquad \text{S}_2\text{-EXTENDS}\ \frac{X\,\texttt{extends}\,T \in \Delta}{\Delta \vdash X \leq T} \qquad \text{S}_2\text{-SUPER}\ \frac{X\,\texttt{super}\,T \in \Delta}{\Delta \vdash T \leq X}$$

$$\text{S}_2\text{-OPEN}\ \frac{\Delta, \overline{P} \vdash N \leq T \qquad \overline{X} \cap \mathsf{ftv}(\Delta, T) = \emptyset}{\Delta \vdash \exists \overline{X}\,\texttt{where}\,\overline{P}\,.\,N \leq T} \qquad \text{S}_2\text{-ABSTRACT}\ \frac{T = [\overline{U/X}]N \qquad (\forall i)\ \Delta \Vdash [\overline{U/X}]P_i}{\Delta \vdash T \leq \exists \overline{X}\,\texttt{where}\,\overline{P}\,.\,N}$$

Fig. 2. Syntax, Entailment, and Subtyping for $\mathcal{EX}_{uplo}$

4 Subtyping Existential Types with Upper and Lower Bounds

This section considers the calculus $\mathcal{EX}_{uplo}$, which is similar in spirit to $\mathcal{EX}_{impl}$, but supports upper and lower bounds for type variables and no implementation constraints. Other researchers [3,4,18] use formal systems very similar to $\mathcal{EX}_{uplo}$ for modeling Java wildcards [19]. It is not the intention of $\mathcal{EX}_{uplo}$ to provide another formalization of wildcards, but rather to expose the essential ingredients that make subtyping undecidable in a calculus as simple as possible.

4.1 Definition of $\mathcal{EX}_{uplo}$

Fig. 2 defines the syntax and the entailment and subtyping relations of $\mathcal{EX}_{uplo}$. A class type N is either `Object` or an instantiated generic class $C\texttt{<}\overline{X}\texttt{>}$, where the type arguments must be type variables. A type T is a type variable, a class type, or an existential. Unlike in $\mathcal{EX}_{impl}$, existentials in $\mathcal{EX}_{uplo}$ may quantify over several type variables and the body of an existential must be a class type. A constraint P places either an upper bound ($X\,\texttt{extends}\,T$) or a lower bound ($X\,\texttt{super}\,T$) on a type variable X. Type environments Δ are defined as for $\mathcal{EX}_{impl}$.

Class definitions and inheritance are omitted from $\mathcal{EX}_{uplo}$. The only assumption is that every class name C comes with a fixed arity that is respected when applying C to type arguments. There are some further restrictions:

(1) If $T = \exists \overline{X}\,\texttt{where}\,\overline{P}\,.\,N$, then $\overline{X} \neq \bullet$ and $\overline{X} \subseteq \mathsf{ftv}(N)$.
(2) If $T = \exists \overline{X}\,\texttt{where}\,\overline{P}\,.\,N$ and $P \in \overline{P}$, then $P = Y\,\texttt{extends}\,T$ or $P = Y\,\texttt{super}\,T$ with $Y \in \overline{X}$. That is, only bound variables may be constrained.
(3) A type variable must not have both upper and lower bounds.[6]

Constraint entailment ($\Delta \Vdash T\,\texttt{extends}\,U$ and $\Delta \Vdash U\,\texttt{super}\,T$) uses subtyping ($\Delta \vdash T \leq U$) to check that the constraint given holds. The subtyping rules for $\mathcal{EX}_{uplo}$ are similar to those for $\mathcal{EX}_{impl}$, except that `Object` is now a supertype of every type and that rules S$_2$-EXTENDS and S$_2$-SUPER use assumptions from Δ.

[6] Modeling Java wildcards requires upper and lower bounds for the same type variable in certain situations.

$$\tau^+ ::= \mathsf{Top} \mid \forall\alpha_0{\leq}\tau_0^- \ldots \alpha_n{\leq}\tau_n^- \,.\, \neg\, \tau^- \qquad (n \in \mathbb{N})$$
$$\tau^- ::= \alpha \mid \forall\alpha_0 \ldots \alpha_n \,.\, \neg\, \tau^+ \qquad (n \in \mathbb{N})$$
$$\Gamma^- ::= \emptyset \mid \Gamma^-, \alpha{\leq}\tau^-$$

D-TOP
$$\Gamma \vdash \tau \leq \mathsf{Top}$$

D-VAR
$$\frac{\tau \neq \mathsf{Top} \qquad \Gamma \vdash \Gamma(\alpha) \leq \tau}{\Gamma \vdash \alpha \leq \tau}$$

D-ALL-NEG
$$\frac{\Gamma, \alpha_0{\leq}\phi_0 \ldots \alpha_n{\leq}\phi_n \vdash \tau \leq \sigma}{\Gamma \vdash \forall\alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma \leq \forall\alpha_0{\leq}\phi_0 \ldots \alpha_n{\leq}\phi_n \,.\, \neg\, \tau}$$

Fig. 3. Syntax and Subtyping for $F^D_{\leq}$

4.2 Undecidability of Subtyping in $\mathcal{EX}_{uplo}$

The undecidability proof of subtyping in $\mathcal{EX}_{uplo}$ is by reduction from $F^D_{\leq}$ [14], a restricted version of $F_{\leq}$ [5]. Pierce defines $F^D_{\leq}$ for his undecidability proof of $F_{\leq}$ subtyping [14].

Fig. 3 defines the syntax and the subtyping relation of $F^D_{\leq}$. A Type τ is either a n-positive type, τ^+, or a n-negative type, τ^-, where n is a fixed natural number standing for the number of type variables (minus one) bound at the top-level of the type. A n-negative type environment Γ^- associates type variables α with upper bounds τ^-. The polarity (+ or −) characterizes at which positions of a subtyping judgment a type or type environment may appear. For readability, we often omit the polarity and leave n implicit.

A *n-ary subtyping judgment* in $F^D_{\leq}$ has the form $\Gamma^- \vdash \sigma^- \leq \tau^+$, where Γ^- is a n-negative type environment, σ^- is a n-negative type, and τ^+ is a n-positive type. Only n-negative types appear to the left and only n-positive types appear to the right of the $\leq$ symbol. The subtyping rule D-ALL-NEG compares two quantified types $\sigma = \forall\alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma'$ and $\tau = \forall\alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\alpha_n \,.\, \neg\, \tau'$ by swapping the left- and right-hand sides of the subtyping judgment and checking $\tau' \leq \sigma'$ under the extended environment $\Gamma, \alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\tau_n$. The rule is correct with respect to $F_{\leq}$ because we may interpret every $F^D_{\leq}$ type as an $F_{\leq}$ type:

$$\forall\alpha_0 \ldots \alpha_n \,.\, \neg\, \sigma' = \forall\alpha_0{\leq}\mathsf{Top} \ldots \forall\alpha_n{\leq}\mathsf{Top}.\forall\beta{\leq}\sigma' \,.\, \beta \;(\beta \text{ fresh})$$
$$\forall\alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\alpha_n \,.\, \neg\, \tau' = \forall\alpha_0{\leq}\tau_0 \ldots \forall\alpha_n{\leq}\tau_n.\forall\beta{\leq}\tau' \,.\, \beta \qquad (\beta \text{ fresh})$$

Using these abbreviations, every $F^D_{\leq}$ subtyping judgment can be read as an $F_{\leq}$ subtyping judgment. The subtype relations in $F^D_{\leq}$ and $F_{\leq}$ coincide for judgments in their common domain [14].

It is sufficient to consider only *closed judgments*. A type τ is closed under Γ if $\mathsf{ftv}(\tau) \subseteq \mathsf{dom}(\Gamma)$ (where $\mathsf{dom}(\alpha_1{\leq}\tau_1, \ldots, \alpha_n{\leq}\tau_n) = \{\alpha_1, \ldots, \alpha_n\}$) and, if $\tau = \forall\alpha_0{\leq}\tau_0 \ldots \alpha_n{\leq}\tau_n \,.\, \neg\, \sigma$, then no α_i appears free in any τ_j. A type environment Γ is closed if $\Gamma = \emptyset$ or $\Gamma = \Gamma', \alpha{\leq}\tau$ with Γ' closed and τ closed under Γ'. A judgment $\Gamma \vdash \tau \leq \sigma$ is closed if Γ is closed and τ, σ are closed under Γ.

We now come to the central theorem of this section.

Theorem 2. *Subtyping in $\mathcal{EX}_{uplo}$ is undecidable.*

Proof. The proof is by reduction from $F^D_{\leq}$. Fig. 4 defines a translation from $F^D_{\leq}$ types, type environments, and subtyping judgments to their corresponding $\mathcal{EX}_{uplo}$ forms. The translation of an n-ary subtyping judgment assumes the

$$
\begin{aligned}
[\![\mathsf{Top}]\!]^+ &= \texttt{Object} \\
[\![\forall \alpha_0 \leq \tau_0^- \ldots \alpha_n \leq \tau_n^- \,.\, \neg\, \tau^-]\!]^+ &= \neg\, \exists Y, \overline{X^{\alpha_i}}\ \texttt{where}\ X^{\alpha_0}\ \texttt{extends}\ [\![\tau_0]\!]^- \ldots \\
&\qquad X^{\alpha_n}\ \texttt{extends}\ [\![\tau_n]\!]^-, Y\ \texttt{extends}\ [\![\tau]\!]^- \,.\, C^{n+2}\texttt{<}Y, \overline{X^{\alpha_i}}\texttt{>} \\
[\![\alpha]\!]^- &= X^\alpha \\
[\![\forall \alpha_0 \ldots \alpha_n \,.\, \neg\, \tau^+]\!]^- &= \neg\, \exists Y, \overline{X^{\alpha_i}}\ \texttt{where}\ Y\ \texttt{extends}\ [\![\tau]\!]^+ \,.\, C^{n+2}\texttt{<}Y, \overline{X^{\alpha_i}}\texttt{>} \\
[\![\emptyset]\!]^- &= \emptyset \\
[\![\Gamma, \alpha \leq \tau^-]\!]^- &= [\![\Gamma]\!]^-, X^\alpha\ \texttt{extends}\ [\![\tau]\!]^- \\
[\![\Gamma^- \vdash \tau^- \leq \sigma^+]\!] &= [\![\Gamma]\!]^- \vdash [\![\tau]\!]^- \leq [\![\sigma]\!]^+ \\
\neg\, T &\equiv \exists X\ \texttt{where}\ X\ \texttt{super}\ T \,.\, D^1\texttt{<}X\texttt{>}
\end{aligned}
$$

Fig. 4. Reduction from $F^D_{\leq}$ to $\mathcal{EX}_{uplo}$

existence of two $\mathcal{EX}_{uplo}$ classes: C^{n+2} accepts $n+2$ type arguments, and D^1 takes one type argument. The superscripts in $[\![\cdot]\!]^+$ and $[\![\cdot]\!]^-$ indicate whether the translation acts on positive or negative entities.

An n-positive type $\forall \alpha_0 \leq \tau_0^- \ldots \alpha_n \leq \tau_n^- \,.\, \neg\, \tau^-$ is translated into an negated existential. The existentially quantified type variables $X^{\alpha_0}, \ldots, X^{\alpha_n}$ correspond to the universally quantified type variables $\alpha_0, \ldots, \alpha_n$. The bound $[\![\tau]\!]^-$ of the fresh type variable Y represents the body $\neg\, \tau^-$ of the original type. We cannot use $[\![\tau]\!]^-$ directly as the body because existentials in $\mathcal{EX}_{uplo}$ have only class types as their bodies. The translation for n-negative types is similar to the one for n-positive types. It is easy to see that the $\mathcal{EX}_{uplo}$ types in the image of the translation meet the restrictions defined in Section 4.1. Type environments and subtyping judgments are translated in the obvious way.

A negated type, written $\neg\, T$, is an abbreviation for an existential with a single `super` constraint: $\neg\, T \equiv \exists X\ \texttt{where}\ X\ \texttt{super}\ T \,.\, D^1\texttt{<}X\texttt{>}$, where X is fresh. The `super` constraint simulates the behavior of the $F^D_{\leq}$ subtyping rule D-ALL-NEG, which swaps the left- and right-hand sides of subtyping judgments.

We now need to verify that $\Gamma \vdash \tau \leq \sigma$ is derivable in $F^D_{\leq}$ if and only if $[\![\Gamma \vdash \tau \leq \sigma]\!]$ is derivable in $\mathcal{EX}_{uplo}$. The "$\Rightarrow$" direction is an easy induction on the derivation of $\Gamma \vdash \tau \leq \sigma$. The "$\Leftarrow$" direction requires more work because the transitivity rule S_2-TRANS (Fig. 2) involves an intermediate type which is not necessarily in the image of the translation. Hence, a direct proof by induction on the derivation of $[\![\Gamma \vdash \tau \leq \sigma]\!]$ fails. To solve this problem, we give an equivalent definition of the $\mathcal{EX}_{uplo}$ subtyping relation that does not include an explicit transitivity rule. See the technical report [22] for details and the full proofs. □

5 Lessons Learned

What are the consequences of this investigation for the design of JavaGI? While existentials are powerful and unify several diverse concepts, they complicate the metatheory of JavaGI considerably. Also, subtyping with existentials is undecidable even under severe restrictions.

The initial development of JavaGI's metatheory uses existentials and imposes several restrictions to ensure decidability of subtyping. However, these restrictions are difficult to explain to users of JavaGI because they seem ad-hoc. Our present view is that existentials may not be worth all the trouble. After all, JavaGI's main feature is its very general and powerful interface concept (which this paper does not explore). Hence, the upcoming revision of JavaGI's design has all features of the original design but it does not require existentials in their full generality. It gives up some of the power in favor of simplicity. Several other features make up for the lack of existentials and experience will show whether this design is satisfactory.

In fact, the upcoming revision of JavaGI copes with all the uses of existentials in JavaGI [21] as mentioned in the introduction.

General composition of interface types. The revised design supports Java-like interface types and intersections thereof.

Wildcards. The revised design does not encode wildcards through existentials but supports them directly.

Meaningful types for multi-headed interfaces. The revised design supports special multi-headed interface types.

Examination of the undecidability proof in § 3 reveals that all types involved are (encodings of) interface types, thus subtyping remains undecidable even if regular interface types replace existentials. The real culprit for undecidability is the ability to provide implementation definitions for existentials or interface types. Moreover, such implementation definitions also prevent the assignment of minimal types to expressions, see the technical report [22] for an example.[7]

Hence, the revised design disallows implementation definitions for interface types. This restriction is rather severe because it prevents useful implementation definitions such as the one given in § 2.2, which implements `Iterable<T>` for all types implementing `List<T>`. *Abstract implementation definitions* are a possible cure. They look similar to regular implementation definitions but do not contribute to constraint entailment. Instead, they serve as blueprints for regular implementation definitions. Here is a revision of the example from § 2.2:

```
abstract implementation<X> Iterable<X> [List<X>] { /* body as before */ }
```

Regular implementation definitions may now inherit code from the abstract implementation. For example:

```
implementation<X> Iterable<X> [LinkedList<X>] extends [List<X>]
implementation<X> Iterable<X> [ArrayList<X>] extends [List<X>]
```

A disadvantage of abstract implementation definitions is that they do not induce a subtyping relation between the implementing type (`List<X>`) and the interface being implemented (`Iterable<X>`). While there is no problem for the concrete example (`List<X>` is a subinterface of `Iterable<X>` anyway), there are situations in which such a subtyping relation is desirable.

[7] Undecidability of subtyping in § 4 relies crucially on existentials with upper and lower bounds. If we removed lower bounds, then subtyping would become decidable. We do not consider this a viable option for JavaGI because it would require to add extra support for wildcards, leading to an overly complicated language design with existentials *and* wildcards.

6 Related Work

Kennedy and Pierce [10] investigate undecidability of subtyping under multiple instantiation inheritance and declaration-site variance. They prove that the general case is undecidable and present three decidable fragments. Our proof in § 3 is similar to theirs, although undecidability has different causes: Kennedy and Pierce's system is undecidable because of contravariant generic types, expansive class tables, and multiple instantiation inheritance, whereas undecidability of our system is due to the interaction of constraint entailment and subtyping caused by implementation definitions for existentials.

Pierce [14] proves undecidability of subtyping in $F_{\leq}$ by a chain of reductions from the halting problem for two-counter Turing machines. An intermediate link in this chain is the subtyping relation of $F^D_{\leq}$, which is also undecidable. Our proof in § 4 works by reduction from $F^D_{\leq}$ and is inspired by a reduction given by Ghelli and Pierce [6], who study bounded existential types in the context of $F_{\leq}$ and show undecidability of subtyping. Crucial to the undecidability proof of $F^D_{\leq}$ is rule D-ALL-NEG: it extends the typing context and essentially swaps the sides of a subtyping judgment. In $\mathcal{EX}_{uplo}$, rule S_2-OPEN and rule S_2-ABSTRACT together with lower bounds on type variables play a similar role.

Torgersen et al. [18] present WildFJ as a model for Java wildcards using existential types. The authors do not prove WildFJ sound. Cameron et al. [4] define a similar calculus ∃J and prove soundness. However, ∃J is not a full model for Java wildcards because it does not support lower bounds for type variables. The same authors present with TameFJ [3] a sound calculus supporting all essential features of Java wildcards. WildFJ's and TameFJ's subtyping rules are similar to the ones of $\mathcal{EX}_{uplo}$ defined in § 4, so the conjecture is that subtyping in WildFJ and TameFJ is also undecidable. The rule XS-ENV of TameFJ is roughly equivalent to the rules S_2-OPEN and S_2-ABSTRACT of $\mathcal{EX}_{uplo}$.

Decidability of subtyping for Java wildcards is still an open question [11]. One step in the right direction might be the work of Plümicke, who solves the problem of finding a substitution φ such that $\varphi T \leq \varphi U$ for Java types T, U with wildcards [15, 16]. Note that undecidability of $\mathcal{EX}_{uplo}$ does not imply undecidability for Java subtyping with wildcards. The proof of this claim would require a translation from subtyping derivations in $\mathcal{EX}_{uplo}$ to subtyping derivations in Java with wildcards, something we did not address in this article.

The programming language Scala [13] supports existential types in its latest release. The subtyping rules for existentials (§ 3.2.10 and § 3.5.2 of the specification [13]) are very similar to the ones for $\mathcal{EX}_{uplo}$. This raises the question whether Scala's subtyping relation with existentials is decidable.

7 Conclusion

The paper investigates decidability of subtyping with existential types in the context of JavaGI, Java wildcards, and Scala. In all cases, subtyping is undecidable. For JavaGI, there are some design options that avoid fully general existentials without giving up much expressivity.

Acknowledgments We thank the anonymous FTfJP reviewers for feedback on an earlier version of this article. We particularly thank the second reviewer for her/his numerous and extensive comments.

References

1. K. B. Bruce, L. Cardelli, G. Castagna, J. Eifrig, S. F. Smith, V. Trifonov, G. T. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.
2. K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1-2):108–133, 1999.
3. N. Cameron, S. Drossopoulou, and E. Ernst. A model for Java with wildcards. In *22th European Conference on Object-Oriented Programming*, 2008. To appear.
4. N. Cameron, E. Ernst, and S. Drossopoulou. Towards an existential types model for Java wildcards. In *Workshop on Formal Techniques for Java-like Programs, informal proceedings*, 2007. `http://www.doc.ic.ac.uk/~ncameron/papers/cameron_ftfjp07_full.pdf`.
5. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17:471–522, Dec. 1985.
6. G. Ghelli and B. Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1-2):75–96, 1998.
7. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, third edition, 2006.
8. M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, UK, 1994.
9. S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proceedings 2nd European Symposium on Programming*, number 300 in Lecture Notes in Computer Science, pages 131–144. Springer-Verlag, 1988.
10. A. J. Kennedy and B. C. Pierce. On decidability of nominal subtyping with variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages, informal proceedings*, Jan. 2007. `http://foolwood07.cs.uchicago.edu/program/kennedy-abstract.html`.
11. K. Mazurak and S. Zdancewic. Type inference for Java 5: Wildcards, F-bounds, and undecidability. `http://www.cis.upenn.edu/~stevez/note.html`, 2006.
12. J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.
13. M. Odersky. The Scala language specification version 2.7, Apr. 2008. Draft, `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.
14. B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, 1994.
15. M. Plümicke. Java type unification with wildcards. In *Proceedings of 17th International Conference on Applications of Declarative Programming and Knowledge Management and 21st Workshop on (Constraint) Logic Programming*, pages 234–245, Oct. 2007.
16. M. Plümicke. Typeless programming in Java 5.0 with wildcards. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*. ACM, Sept. 2007.
17. E. L. Post. A variant of a recursivley unsolvable problem. *Bulletin of the American Mathematical Society*, 53:264–268, 1946.

18. M. Torgersen, E. Ernst, and C. P. Hansen. Wild FJ. In *International Workshop on Foundations of Object-Oriented Languages, informal proceedings*, 2005. `http://homepages.inf.ed.ac.uk/wadler/fool/program/14.html`.
19. M. Torgersen, E. Ernst, C. P. Hansen, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, Dec. 2004.
20. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, Jan. 1989. ACM Press.
21. S. Wehr, R. Lämmel, and P. Thiemann. JavaGI: Generalized interfaces for Java. In E. Ernst, editor, *21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 347–372, Berlin, Germany, July 2007. Springer-Verlag.
22. S. Wehr and P. Thiemann. Subtyping existential types. Technical Report 240, Universität Freiburg, June 2008. `ftp://ftp.informatik.uni-freiburg.de/documents/reports/report240/report00240.ps.gz`.