Proceedings of the First International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO)

Edited by Dave Clarke

institute of information and computing sciences, utrecht university

technical report UU-CS-2003-030

www.cs.uu.nl

International Workshop on Aliasing, Confinement and Ownership (IWACO)

Held in conjunction with ECOOP 2003
Darmstadt, Germany, July 21-25, 2003

Dave Clarke (Editor)
Institute of Information and Computing Sciences
Utrecht University

Tuesday, July 22, 2003

Contents

Preface

The power of objects lies in the flexibility of their interconnection structure. But this flexibility comes at a cost. Because an object can be modified via any alias, object-oriented programs are hard to understand, maintain, and analyse. Aliasing makes objects depend on their environment in unpredictable ways, breaking the encapsulation necessary for reliable software components, making it difficult to reason about and optimise programs, obscuring the flow of information between objects, and introducing security problems.

Aliasing is a fundamental difficulty, but we accept its presence. Instead we seek techniques for describing, reasoning about, restricting, analysing, and preventing the connections between objects and/or the flow of information between them.

So how then do we take arms against the sea of objects?

Topics the workshop aims to address include:

- models, type and other formal systems, programming language mechanisms, analysis and design techniques, patterns and notations for expressing object ownership, aliasing, confinement, uniqueness, and/or information flow.
- optimisation techniques, analysis algorithms, libraries, applications, and novel approaches exploiting object ownership, aliasing, confinement, uniqueness, and/or information flow.
- empirical studies of programs or experience reports from programming systems designed with these issues in mind.
- novel applications of aliasing management techniques such as ownership types, confined types, region types, and uniqueness.

Dave Clarke, Sophia Drosspoulou, James Noble
Workshop Organisers

Organisers

Dave Clarke	Utrecht University
Sophia Drossopoulou	Imperial College, London
James Noble	Victoria University of Wellington

Invited Speaker

Peter O'Hearn	Separation and Confinement

Program Committee

Jonathan Aldrich	University of Washington
Dave Clarke (chair)	Utrecht University
Doug Lea	SUNY Oswego
David Naumann	Stevens Institute of Technology
James Noble	Victoria University of Wellington
Peter O'Hearn	Queen Mary, University of London
Martin Rinard	MIT
Jan Vitek	Purdue University

Additional Reviewers

Anindya Banerjee, Sophia Drossopoulou, Matthew Smith, Cees Pierik, Tobias Wrigstad

Thanks

Utrecht University footed the bill for printing these proceedings.

Tobias Wrigstad helped with pdf hacking.

Schedule

8:50	Start: Heartfelt Welcome, Rules of Engagement
9:00-10:00	Invited Talk Separation and Confinement (1 hour) Peter O'Hearn (Queen Mary, University of London)
10:00-10:30	Mini-Session Safe Runtime Downcasts with Ownership Types (20 Minutes) Chandrasekhar Boyapati, Robert Lee, Martin Rinard Short Discussion
10:30-11:00	Break
11:00-12:30	Session 1 Cheaper Reasoning With Ownership Types (20 Minutes) Matthew Smith, Sophia Drossopoulou Formalising Eiffel References and Expanded Types in PVS (20 Minutes) Richard Paige, Jonathan Ostroff, Phillip Brooke Connecting Effects and Uniqueness with Adoption (20 Minutes) John Tang Boyland Short Discussion
12:30-13:30	Lunch
13:30-15:00	Session 2 Heap Monotonic Typestates (20 Minutes) Manuel Fähndrich, K. Rustan M. Leino Towards a Model of Encapsulation (20 Minutes) James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, Dave Clarke Discussion
15:00-15:30	Break
15:30-17:00	Session 3 Lightweight Confinement for Featherweight Java (20 Minutes) Tian Zhao, Jens Palsberg, Jan Vitek A Static Capability Tracking System (20 Minutes) Scott F. Smith, Mark Thober Discussion

Safe Runtime Downcasts With Ownership Types

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square, Cambridge, MA 02139
{chandra,rhlee,rinard}@lcs.mit.edu

Abstract. This paper describes an efficient technique for supporting safe runtime downcasts in a system with ownership types. This technique uses the type passing approach, but avoids the associated significant space overhead by storing only the runtime ownership information that is potentially needed to support safe downcasts. Moreover, this technique does not use any inter-procedural analysis, so it preserves the separate compilation model of Java. We implemented our technique in the context of Safe Concurrent Java, which is an extension to Java that uses ownership types to statically guarantee the absence of data races and deadlocks. Our approach is JVM-compatible: our implementation translates programs to bytecodes that can be run on regular JVMs.

1 Introduction

Ownership types [4, 5, 7, 12, 13] provide a statically enforceable way of specifying object encapsulation. The idea is that an object can *own* subobjects that it depends on, thus preventing them from being accessible outside. Object encapsulation enables local reasoning about program correctness in object-oriented programs. Ownership-based type systems have also been used for preventing data races [7] and deadlocks [4] in multithreaded programs, for preventing memory errors in programs that use region-based memory management [8], for supporting modular software upgrades in persistent object stores [6], for modular specification of effects clauses in the presence of subtyping [5, 7, 12] (so that they can be used as an alternative to data groups [16]), and for program understanding [2].

In ownership type systems, programmers parameterize classes and methods by owners. This enables the writing of generic code that can be used in many different contexts. The parameterization is somewhat similar to the proposals for parametric types for Java [1, 9, 18, 20]. Ownership type systems are primarily static type systems. The type checker uses the ownership type annotations to statically ensure the absence of certain classes of errors (e.g., data races in PRFJ [7]), but it is usually unnecessary to preserve the ownership information at runtime. However, languages like Java [15] are not purely statically typed languages. Java allows downcasts that are checked at runtime. To support safe runtime downcasts, the system must preserve some ownership information at runtime when ownership types are used in the context of a language like Java.

There are primarily three techniques for implementing parametric polymorphism in a language like Java. The *type erasure* approach [9, 10] is based on the idea of deleting type parameters (so Stack⟨T⟩ erases to Stack). But this approach will not preserve ownership information at runtime, so it is unsuitable for supporting safe runtime downcasts with ownership types. In the *code duplication* approach [1], polymorphism is supported by creating specialized classes/methods, each supporting a different instantiation of a parametric class/method. But since the parameters in ownership types are usually objects, this approach will lead to an unacceptably large number of classes/methods. In the *type passing* approach [18, 20, 19], information on type parameters is explicitly stored in objects and passed to code requiring them. But if the system stores the owners of every object at runtime, this approach has the potential drawback of adding a per-object space overhead. Java objects are typically small, so adding even a single field to every object increases the size of most objects by a significant fraction.

This paper describes an efficient technique for supporting safe runtime downcasts with ownership types. This technique uses the type passing approach, but avoids the associated significant space overhead by storing only the runtime ownership information that is potentially needed to support safe downcasts. Moreover, this technique does not use any inter-procedural analysis, so it preserves the separate compilation model of Java. We implemented our technique in Safe Concurrent Java [4, 7], which is an extension to Java that uses ownership types to guarantee the absence of data races and deadlocks in well-typed programs. Our approach is JVM-compatible: our implementation translates programs to bytecodes that can be run on regular JVMs [17].

We note that a similar approach has been used in [2] to implement safe runtime downcasts with ownership types.

The rest of this paper is organized as follows. Section 2 gives an overview of ownership types in subset of Safe Concurrent Java (SCJ). Section 3 describes how we support safe runtime downcasts. Section 4 concludes.

2 Mini Safe Concurrent Java

This section presents Mini Safe Concurrent Java (MSCJ), which is a subset of SCJ that prevents data races in well-typed programs. To simplify the presentation of key ideas behind our approach, the rest of the discussion in this paper will be in the context of MSCJ. Our implementation, however, works for the whole of SCJ and handles all the features of the Java language. The key to the MSCJ type system is the concept of object ownership. Every object in MSCJ has an owner. An object can be owned by another object, by itself, or by a special per-thread owner called thisThread. Objects owned by thisThread, either directly or transitively, are local to the corresponding thread and cannot be accessed by any other thread. Figure 1 presents an example ownership relation. We draw an arrow from object x to object y if x owns y. Our type system statically verifies that a program respects the ownership properties shown in Figure 2.

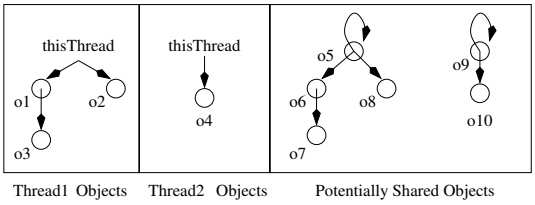

Fig. 1. An Ownership Relation

1. The owner of an object does not change over time.
2. The ownership relation forms a forest of rooted trees. The roots can have self loops.
3. To safely access an object, a thread must hold the lock on the *root owner* of that object (the root of the ownership tree that the object belongs to).
4. Every thread implicitly holds the lock on its corresponding thisThread owner. A thread can thus access objects owned by its thisThread without synchronization.

Fig. 2. Ownership Properties

Figure 3 shows the grammar for MSCJ. Figure 4 shows a TStack program in MSCJ. For simplicity, all the examples in this paper use an extended language that is syntactically closer to Java. A TStack is a stack of T objects. A TStack is implemented using a linked list. A class definition in MSCJ is parameterized by a list of owners. This parameterization helps programmers write generic code to implement a class, then create different objects of the class that have different protection mechanisms. In Figure 4, the TStack class is parameterized by thisOwner and TOwner. thisOwner owns the this TStack object and TOwner owns the T objects contained in the TStack. In general, the first formal parameter of a class always owns the this object. In case of s1, the owner thisThread is used for both the parameters to instantiate the TStack class. This means that the main thread owns TStack s1 as well as all the T objects contained in the TStack. In case of s2, the main thread owns the TStack but the T objects contained in the

$$
\begin{array}{rcl}
P & ::= & defn^*\ e \\
defn & ::= & \mathsf{class}\ cn\langle owner\ f^*\rangle\ \mathsf{extends}\ c\ \{field^*\ meth^*\} \\
c & ::= & cn\langle owner+\rangle \mid \mathsf{Object}\langle owner\rangle \\
owner & ::= & f \mid \mathsf{self} \mid \mathsf{thisThread} \mid e_{\mathrm{final}} \\
meth & ::= & t\ mn(arg^*)\ \mathsf{accesses}\ (e_{\mathrm{final}}{}^*)\ \{e\} \\
field & ::= & [\mathsf{final}]_{\mathrm{opt}}\ t\ fd = e \\
arg & ::= & [\mathsf{final}]_{\mathrm{opt}}\ t\ x \\
t & ::= & c \mid \mathsf{int} \\
 & & \\
e & ::= & \mathsf{new}\ c \mid x \mid x = e \mid e.fd \mid e.fd = e \mid e.mn(e^*) \mid e;e \mid \mathsf{let}\ (arg{=}e)\ \mathsf{in}\ \{e\} \mid \\
 & & \mathsf{synchronized}\ (e)\ \mathsf{in}\ \{e\} \mid \mathsf{fork}\ (x^*)\ \{e\} \\
e_{\mathrm{final}} & ::= & e
\end{array}
$$

$cn \in$ class names, $fd \in$ field names, $mn \in$ method names, $x \in$ variable names, $f \in$ owner names

Fig. 3. MSCJ Grammar

```
1   // thisOwner owns the TStack object, TOwner owns the T objects in the stack.
2   class TStack<thisOwner, TOwner> {
3     TNode<this, TOwner> head = null;
4     TStack() {}
5     void push(T<TOwner> value) accesses (this) {
6       TNode<this, TOwner> newNode = new TNode<this, TOwner>(value, head); head = newNode;
7     }
8     T<TOwner> pop() accesses (this) {
9       T<TOwner> value = head.value(); head = head.next(); return value;
10    }
11  }
12  class TNode<thisOwner, TOwner> {
13    T<TOwner> value; TNode<thisOwner, TOwner> next;
14    TNode(T<TOwner> v, TNode<thisOwner, TOwner> n) accesses (this) {
15      this.value = v; this.next  = n;
16    }
17    T<TOwner> value() accesses (this) { return value; }
18    TNode<thisOwner, TOwner> next() accesses (this) { return next; }
19  }
20  class T<thisOwner> { int x=0; }
21
22  TStack<thisThread, thisThread> s1 = new TStack<thisThread, thisThread>;
23  TStack<thisThread, self>       s2 = new TStack<thisThread, self>;
```

Fig. 4. Stack of T Objects in MSCJ

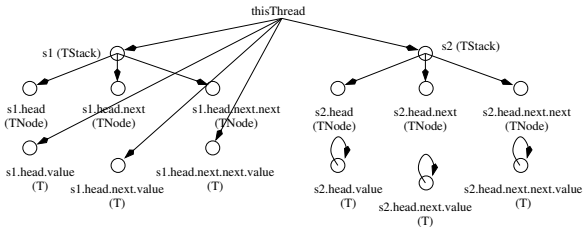

Fig. 5. Ownership Relation for TStacks s1 and s2

TStack own themselves. The ownership relation for the TStack objects s1 and s2 is depicted in Figure 5 (assuming the stacks contain three elements each). In MSCJ, a method can contain an accesses clause that specifies the objects the method accesses that must be protected by externally acquired locks. Callers are required to hold the locks on the root owners of the objects specified in the accesses clause before they invoke a method. In the example, the value and next methods in the TNode class assume that the callers hold the lock on the root owner of the this TNode object.

2.1 Static Type Checking

This section describes some of the important type checking rules. The full set of rules can be found in [4]. The core of our type system is a set of rules for reasoning about the typing judgment: P; E; $ls \vdash e : t$. P, the program being checked, is included here to provide information about class definitions. E is an environment providing types for the free variables of e. ls describes the set of locks that are statically known to be held when e is evaluated. t is the type of e.

The rule for accessing field $e.fd$ checks that e is a well-typed expression of some class type $cn\langle o_{1..n}\rangle$, where $o_{1..n}$ are actual owner parameters. It verifies that the class cn with formal parameters $f_{1..n}$ declares or inherits a field fd of type t and that the thread holds the lock on the root owner of e. Since t is declared inside the class, it might contain occurrences of this and the formal class parameters. When t is used outside the class, we rename this with the expression e, and the formal parameters with their corresponding actual parameters.

[EXPRESSION REFERENCE]

$$\frac{P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \qquad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \qquad P;\ E \vdash \text{RootOwner}(e) \in ls}{P;\ E;\ ls \vdash e.fd : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]}$$

The rule for invoking a method checks that the arguments are of the right type and that the thread holds the locks on the root owners of all expressions in the accesses clause of the method. The expressions and types used inside the method are renamed appropriately when used outside their class.

[EXPRESSION INVOKE]

$$\text{Renamed}(\alpha) \stackrel{\text{def}}{=} \alpha[e/\mathsf{this}][o_1/f_1]..[o_n/f_n][e_1/y_1]..[e_k/y_k]$$

$$\frac{\begin{array}{c} P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \qquad P \vdash (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{accesses}(e'^*)\ \{...\}) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash e_j : \text{Renamed}(t_j) \\ P;\ E \vdash \text{RootOwner}(\text{Renamed}(e'_i)) \in ls \end{array}}{P;\ E;\ ls \vdash e.mn(e_{1..k}) : \text{Renamed}(t)}$$

The rule for checking a method assumes that the locks on the root owners of all the expressions specified in the accesses clause are held. The rule then type checks the method body under this assumption.

[METHOD]

$$\frac{\begin{array}{c} E' = E,\ arg_{1..n} \qquad P;\ E' \vdash_{\mathsf{final}} e_i : t_i \qquad P;\ E' \vdash \text{RootOwner}(e_i) = r_i \\ P;\ E';\ \mathsf{thisThread},\ r_{1..r} \vdash e : t \end{array}}{P;\ E \vdash t\ mn(arg_{1..n})\ \mathsf{accesses}(e_{1..r})\ \{e\}}$$

The subtyping rule ensures that the parameters of the supertype are instantiated either with constants (self or thisThread) or with owners that are in scope, preserving the owner in the first position. The first owner must be preserved because the first owner in our system is special, in that it owns the this object.

[SUBTYPE]

$$\frac{\begin{array}{c} P;\ E \vdash cn_1\langle o_{1..n}\rangle \qquad P \vdash \mathsf{class}\ cn_1\langle f_{1..n}\rangle\ \mathsf{extends}\ cn_2\langle f_1\ o'^*\rangle\ \{...\} \\ \forall\ o'.\quad (o' = \mathsf{self}) \vee (o' = \mathsf{thisThread}) \vee (\exists\ j.\quad o' = f_j) \end{array}}{P;\ E \vdash cn_1\langle o_{1..n}\rangle <: cn_2\langle f_1\ o'^*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

3 Safe Runtime Downcasts

This section describes how we support safe runtime downcasts efficiently. We describe our technique in the context of Mini Safe Concurrent Java (MSCJ).

The type system for MSCJ described in Section 2 is a purely static type system. In fact, one way to compile and run a MSCJ program is to convert it into a Java program after type checking, by removing the type parameters and the accesses clauses. However, a language like Java is not a purely statically typed language. Java allows downcasts that are checked at runtime. To support safe downcasts, the system must preserve some ownership information at runtime when ownership types are used in the context of a language like Java. To express runtime casts, we extend the MSCJ grammar as follows.

$$e ::= \ldots \mid (cn\langle o_{1..n}\rangle)\ e$$

Fig. 6. Grammar Extensions to Support Runtime Casts

We next present the static type checking rules for casts. Casting an object to a supertype of its declared type is always safe. Casting to a subtype of the declared type requires runtime checking. Section 2.1 contains the subtyping rule.

[EXPRESSION UPCAST]

$$\frac{P;\ E;\ ls \vdash e : c_2 \qquad P;\ E \vdash c_2 <: c_1}{P;\ E;\ ls \vdash (c_1)\ e : c_1}$$

[EXPRESSION DOWNCAST (REQUIRES RUNTIME CHECK)]

$$\frac{P;\ E;\ ls \vdash e : c_1 \qquad P;\ E \vdash c_2 <: c_1}{P;\ E;\ ls \vdash (c_2)\ e : c_2}$$

To support downcasts, we store information on type parameters explicitly in objects and pass the information to code requiring the information. But if the system stores the owners of every object at runtime, this approach has the potential drawback of adding a per-object space overhead. Java objects are typically small, so adding even a single field to every object increases the size of most objects by a significant fraction. Our technique avoids the associated significant space overhead by storing only the runtime ownership information that is potentially needed to support safe downcasts. Our technique is based on two key observations about the nature of parameterization in ownership types.

The remainder of this section is organized as follows. Sections 3.1 and 3.2 describe the key observations that enable us to support downcasts efficiently. Sections 3.3 and 3.4 presents our technique for supporting safe downcasts.

3.1 Downcasts to Types With Single Owners

A key observation that enables efficient implementation of downcasts is as follows. Consider the code in Figure 7. In Line 16, object o1 of declared type Object⟨thisThread⟩ is downcast to type T⟨thisThread⟩. For this downcast, the owner of the declared type of o1 matches the owner of the type that o1 is being

```
 1  class T<thisOwner> {...}
 2  class TStack<thisOwner, TOwner> {...}
 3  class TStack2<thisOwner, TOwner> extends TStack<thisOwner, TOwner> {...}
 4
 5  Object<thisThread> o1, o2, o3;
 6  ...
 7  T<thisThread> t1;
 8  T<self>       t2;
 9  ...
10  TStack<thisThread, thisThread>  s1;
11  TStack<thisThread, self>        s2;
12  ...
13  TStack2<thisThread, thisThread> q1;
14  TStack2<thisThread, self>       q2;
15  ...
16  t1 = (T<thisThread>) o1;                  // Safe iff o1 belongs to class T
17  t2 = (T<self>)       o2;                  // Compile time error
18  ...
19  s1 = (TStack<thisThread, thisThread>)  o3; // Requires checking runtime ownership
20  s2 = (TStack<thisThread, self>)  o3;       // Requires checking runtime ownership
21  ...
22  q1 = (TStack2<thisThread, thisThread>) s1; // Safe iff s1 belongs to class TStack2
23  q2 = (TStack2<thisThread, self>) s1;       // Compile time error
```

Fig. 7. Runtime Downcasts

downcast into. Hence, this downcast is safe iff o1 belongs to class T at runtime. It is unnecessary to check ownership information at runtime for this downcast.

In general, for any subtype declaration where all the formal owner parameters in the subtype are included in the supertype, it is not necessary to check ownership information at runtime when an object is downcast from the supertype to the subtype. If the owners of the supertype match the owners of the subtype, then the downcast will be safe iff the object belongs to the appropriate class at runtime (e.g., Lines 16 and 22 in Figure 7). If the owners do not match, the downcast will always fail (e.g., Lines 17 and 23 in Figure 7).

The primary benefit of this observation is that whenever an object is downcast into a type with a single owner, it is unnecessary to check ownership information at runtime to ensure that the downcast is safe. Since a vast majority of classes in a system with ownership types have single owners, this implies that it is unnecessary to check ownership information at runtime for most of the downcasts. The only classes that usually have multiple owners are collection classes. The only times when it might be necessary to check ownership information at runtime to ensure that the downcast is safe is when an object is downcast into a type with multiple owners (e.g., Lines 19 and 20 in Figure 7).

3.2 Anonymous Owners

Another key observation that enables efficient implementation of downcasts is as follows. Consider the code in Figure 4. The TStack class in the figure is parameterized by thisOwner and TOwner. However, the owner parameter thisOwner is not used in the static scope where it is visible. Similarly, the owner parameter thisOwner for class T is not used in the body of class T. If an owner parameter is

not used, it is unnecessary to name the parameter. Our system allows programmers to use ⟨-⟩ for such anonymous owner parameters. Figure 8 shows how we extend the MSCJ grammar to support anonymous owner parameters. Figure 11 shows the TStack example in Figure 4 implemented using anonymous owners.

$$defn ::= \ldots \mid \mathsf{class}\ cn\langle\text{-}\ f^*\rangle\ \mathsf{extends}\ c\ \{field^*\ meth^*\}$$

Fig. 8. Grammar Extensions to Support Anonymous Owners

The primary benefit of having anonymous owners is that if an owner parameter of a class is not named, it is unnecessary to store the owner parameter of the class at runtime, or pass the owner parameter to code that uses the class at runtime. In a system with ownership types, the only classes that usually have named owners are collection classes with multiple owners. Examples include Vector⟨-,elementOwner⟩, Hashtable⟨-,keyOwner,valueOwner⟩, etc. But most classes have single owners that are anonymous. It is unnecessary to store ownership information for those classes, or pass ownership information to code that uses those classes. Thus, our system incurs a runtime space and time overhead only for code that uses classes with named owner parameters like the collection classes. The rest of the code has no overhead in our system.

3.3 Preserving Ownership Information at Runtime

This section describes how our system preserves ownership information at runtime for classes with named owner parameters in the context of MSCJ. We presented the grammar for MSCJ in Figure 3 with extensions in Figures 6 and 8. This section presents the rules for translating a MSCJ program into an equivalent program in a Java-like language without ownership types. If we did not have to support safe runtime downcasts, the translation process would have been simple. We could have converted a MSCJ program into an equivalent Java-like program by simply removing the owner parameters and the accesses clauses. However, to support safe runtime downcasts, we must preserve some ownership information in the translation process.

The core of our translation is a set of rules of the form: $(\mathcal{T}[\![C]\!]\ P\ E) = C'$. The rule translates a code fragment C to a code fragment C'. P, the program being checked, is included here to provide information about class definitions. E is an environment containing the formal owner parameters in scope in C. The translated code uses the $Owner class shown in Figure 9. The $Owner class

```
1  public class $Owner {
2    public static Object self = "self";
3
4    public static Object SELF() { return self; }
5    public static Object THISTHREAD() { return Thread.currentThread(); }
6  }
```

Fig. 9. The $Owner Class

$$
\begin{array}{ll}
(\mathcal{T}[\![P]\!]) & = (\mathcal{T}[\![defn^*\ e]\!]) \\
 & = (\mathcal{T}[\![defn]\!]\ P)^*\ (\mathcal{T}[\![e]\!]\ P\ \emptyset) \\
 & \\
(\mathcal{T}[\![defn]\!]\ P) & = (\mathcal{T}[\![\mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ cn'\langle o_{1..n'}\rangle\ \{field^*\ meth^*\}]\!]\ P) \\
 & = \mathsf{class}\ cn\ \mathsf{extends}\ cn' \\
 & \quad \{\mathsf{Object}\ \$f_{1..n}\ (\mathcal{T}[\![field]\!]\ P\ [f_{1..n}])^*\ (\mathcal{T}[\![method]\!]\ P\ [f_{1..n}])^*\} \\
 & \\
(\mathcal{T}[\![defn]\!]\ P) & = (\mathcal{T}[\![\mathsf{class}\ cn\langle\text{- } f_{2..n}\rangle\ \mathsf{extends}\ cn'\langle o_{1..n'}\rangle\ \{field^*\ meth^*\}]\!]\ P) \\
 & = \mathsf{class}\ cn\ \mathsf{extends}\ cn' \\
 & \quad \{\mathsf{Object}\ \$f_{2..n}\ (\mathcal{T}[\![field]\!]\ P\ [f_{2..n}])^*\ (\mathcal{T}[\![method]\!]\ P\ [f_{2..n}])^*\} \\
 & \\
(\mathcal{T}[\![meth]\!]\ P\ E) & = (\mathcal{T}[\![t\ mn(arg^*)\ \mathsf{accesses}\ (e_{\mathsf{final}}{}^*)\ \{e\}]\!]\ P\ E) \\
 & = (\mathcal{T}[\![t]\!]\ P\ E)\ mn\ ((\mathcal{T}[\![arg]\!]\ P\ E)^*)\ \{(\mathcal{T}[\![e]\!]\ P\ E)\} \\
 & \\
(\mathcal{T}[\![field]\!]\ P\ E) & = (\mathcal{T}[\![[\mathsf{final}]_{\mathrm{opt}}\ t\ fd = e]\!]\ P\ E) \\
 & = [\mathsf{final}]_{\mathrm{opt}}\ (\mathcal{T}[\![t]\!]\ P\ E)\ fd = (\mathcal{T}[\![e]\!]\ P\ E) \\
 & \\
(\mathcal{T}[\![arg]\!]\ P\ E) & = (\mathcal{T}[\![[\mathsf{final}]_{\mathrm{opt}}\ t\ fd]\!]\ P\ E) \\
 & = [\mathsf{final}]_{\mathrm{opt}}\ (\mathcal{T}[\![t]\!]\ P\ E)\ fd \\
 & \\
(\mathcal{T}[\![t]\!]\ P\ E) & = (\mathcal{T}[\![cn\langle owner+\rangle]\!]\ P\ E) \\
 & = cn \\
 & \\
(\mathcal{T}[\![t]\!]\ P\ E) & = (\mathcal{T}[\![\mathsf{int}]\!]\ P\ E) \\
 & = \mathsf{int} \\
 & \\
(\mathcal{T}[\![e]\!]\ P\ E) & = (\mathcal{T}[\![(cn\langle o_{1..n}\rangle)\ e]\!]\ P\ E) \\
 & = \{\$\mathrm{temp} = (cn)\ (\mathcal{T}[\![e]\!]\ P\ E); \\
 & \quad \mathsf{if}\ (\$\mathrm{temp}.\$f_2\ \mathsf{!=}\ (\mathcal{O}[\![o_2]\!]\ P\ E))\ \mathsf{throw\ new\ ClassCastException}; \\
 & \quad ...; \\
 & \quad \mathsf{if}\ (\$\mathrm{temp}.\$f_n\ \mathsf{!=}\ (\mathcal{O}[\![o_n]\!]\ P\ E))\ \mathsf{throw\ new\ ClassCastException}; \\
 & \quad \$\mathrm{temp}\} \\
 & \\
(\mathcal{T}[\![e]\!]\ P\ E) & = (\mathcal{T}[\![\mathsf{new}\ cn\langle o_{1..n}\rangle]\!]\ P\ E) \\
 & = \{\$\mathrm{temp} = \mathsf{new}\ cn; \\
 & \quad \$\mathrm{temp}.\$f_1 = (\mathcal{O}[\![o_1]\!]\ P\ E);\ ...;\ \$\mathrm{temp}.\$f_n = (\mathcal{O}[\![o_n]\!]\ P\ E); \\
 & \quad \$\mathrm{temp}\} \\
 & \\
 & \quad \text{where } (\mathsf{class}\ cn\langle f_{1..n}\rangle\ ...) \in P \\
 & \\
(\mathcal{T}[\![e]\!]\ P\ E) & = (\mathcal{T}[\![\mathsf{new}\ cn\langle o_{1..n}\rangle]\!]\ P\ E) \\
 & = \{\$\mathrm{temp} = \mathsf{new}\ cn; \\
 & \quad \$\mathrm{temp}.\$f_2 = (\mathcal{O}[\![o_2]\!]\ P\ E);\ ...;\ \$\mathrm{temp}.\$f_n = (\mathcal{O}[\![o_n]\!]\ P\ E); \\
 & \quad \$\mathrm{temp}\} \\
 & \\
 & \quad \text{where } (\mathsf{class}\ cn\langle\text{- } f_{2..n}\rangle\ ...) \in P
\end{array}
$$

$$
\begin{array}{lll}
(\mathcal{O}[\![o]\!]\ P\ E) = (\mathcal{O}[\![\mathsf{thisThread}]\!]\ P\ E) & & = \$\mathrm{Owner.THISTHREAD()} \\
(\mathcal{O}[\![o]\!]\ P\ E) = (\mathcal{O}[\![\mathsf{self}]\!]\ P\ E) & & = \$\mathrm{Owner.SELF()} \\
(\mathcal{O}[\![o]\!]\ P\ E) = (\mathcal{O}[\![f]\!]\ P\ [...\ f\ ...]) & & = \$\mathrm{f} \\
(\mathcal{O}[\![o]\!]\ P\ E) = (\mathcal{O}[\![\mathrm{e}]\!]\ P\ E) & & = (\mathcal{T}[\![e]\!]\ P\ E)
\end{array}
$$

$$
\begin{array}{lll}
(\mathcal{T}[\![e]\!]\ P\ E) = (\mathcal{T}[\![x]\!]\ P\ E) & & = x \\
(\mathcal{T}[\![e]\!]\ P\ E) = (\mathcal{T}[\![x = e]\!]\ P\ E) & & = x = (\mathcal{T}[\![e]\!]\ P\ E) \\
(\mathcal{T}[\![e]\!]\ P\ E) = (\mathcal{T}[\![e.fd]\!]\ P\ E) & & = (\mathcal{T}[\![e]\!]\ P\ E).fd \\
(\mathcal{T}[\![e]\!]\ P\ E) = (\mathcal{T}[\![e_1.fd = e]\!]\ P\ E) & & = (\mathcal{T}[\![e_1]\!]\ P\ E).fd = (\mathcal{T}[\![e]\!]\ P\ E) \\
(\mathcal{T}[\![e]\!]\ P\ E) = (\mathcal{T}[\![e_1.mn(e^*)]\!]\ P\ E) & & = (\mathcal{T}[\![e_1]\!]\ P\ E).mn((\mathcal{T}[\![e]\!]\ P\ E)^*) \\
(\mathcal{T}[\![e]\!]\ P\ E) = (\mathcal{T}[\![e_1;e_2]\!]\ P\ E) & & = (\mathcal{T}[\![e_1]\!]\ P\ E);(\mathcal{T}[\![e_2]\!]\ P\ E) \\
(\mathcal{T}[\![e]\!]\ P\ E) = (\mathcal{T}[\![\mathsf{let}\ (arg{=}e_1)\ \mathsf{in}\ \{e\}]\!]\ P\ E) & & = \mathsf{let}\ (arg{=}(\mathcal{T}[\![e_1]\!]\ P\ E))\ \mathsf{in}\ \{(\mathcal{T}[\![e]\!]\ P\ E[arg])\} \\
(\mathcal{T}[\![e]\!]\ P\ E) = (\mathcal{T}[\![\mathsf{synchronized}\ (e_1)\ \mathsf{in}\ \{e\}]\!]\ P\ E) & & = \mathsf{synchronized}\ ((\mathcal{T}[\![e_1]\!]\ P\ E))\ \mathsf{in}\ \{(\mathcal{T}[\![e]\!]\ P\ E)\} \\
(\mathcal{T}[\![e]\!]\ P\ E) = (\mathcal{T}[\![\mathsf{fork}\ (x^*)\ \{e\}]\!]\ P\ E) & & = \mathsf{fork}\ (x^*)\ \{(\mathcal{T}[\![e]\!]\ P\ E)\}
\end{array}
$$

Fig. 10. Translation Function

```
1  // TStack has an anonymous owner, TOwner owns the T objects in the stack.
2
3  class TStack<-, TOwner> {
4
5    TNode<this, TOwner> head = null;
6
7    TStack() {}
8    void push(T<TOwner> value) accesses (this) {
9      TNode<this, TOwner> newNode = new TNode<this, TOwner>(value, head); head = newNode;
10   }
11   T<TOwner> pop() accesses (this) {
12     T<TOwner> value = head.value(); head = head.next(); return value;
13   }
14 }
15
16 class TNode<thisOwner, TOwner> {
17
18   T<TOwner> value; TNode<thisOwner, TOwner> next;
19
20   TNode(T<TOwner> v, TNode<thisOwner, TOwner> n) accesses (this) {
21     this.value = v; this.next  = n;
22   }
23   T<TOwner> value() accesses (this) { return value; }
24   TNode<thisOwner, TOwner> next() accesses (this) { return next; }
25 }
26
27 class T<-> { int x=0; }
```

Fig. 11. TStack With Anonymous Owners

```
1  class T<-> {...}
2  class TStack<-, TOwner> {...}
3  class TStack2<-, TOwner> extends TStack<-, TOwner> {...}
4
5  Object<thisThread> o1;
6  Object<thisThread> o2;
7  ...
8  TStack<thisThread, thisThread>  s1 = new TStack<thisThread, thisThread>;
9  TStack<thisThread, self>        s2 = new TStack<thisThread, self>;
10 ...
11 TStack2<thisThread, thisThread> q1;
12 TStack2<thisThread, self>       q2;
13 ...
14 s1 = (TStack<thisThread, thisThread>)  o1;
15 s2 = (TStack<thisThread, self>)  o2;
16 ...
17 q1 = (TStack2<thisThread, thisThread>) s1;
18 q2 = (TStack2<thisThread, self>) s2;
19 ...
20 boolean b1 = (o1 instanceof TStack<thisThread, thisThread>);
21 boolean b2 = (o2 instanceof TStack<thisThread, self>);
```

Fig. 12. Client Code for TStack

```
1   // TStack has an anonymous owner, TOwner owns the T objects in the stack.
2
3   class TStack {
      Object $TOwner;
4
5     TNode head = null;
6
7     TStack(Object $TOwner) {
        this.$TOwner = $TOwner;
      }
8     void push(T value) {
9       TNode newNode = new TNode(this, $TOwner, value, head); head = newNode;
10    }
11    T pop() {
12      T value = head.value(); head = head.next(); return value;
13    }
14  }
15
16  class TNode {
      Object $thisOwner, $TOwner;
17
18    T value; TNode next;
19
20    TNode(Object $thisOwner, Object $TOwner, T v, TNode n) {
        this.$thisOwner = $thisOwner; this.$TOwner = $TOwner;
21      this.value = v; this.next  = n;
22    }
23    T value() { return value; }
24    TNode next() { return next; }
25  }
26
27  class T { int x=0; }
```

Fig. 13. Translation of TStack in Figure 11

```
1   class T {...}
2   class TStack {...}
3   class TStack2 extends TStack {...}
4
5   Object o1;
6   Object o2;
7   ...
8   TStack s1 = new TStack($Owner.THISTHREAD());
9   TStack s2 = new TStack($Owner.SELF());
10  ...
11  TStack2 q1;
12  TStack2 q2;
13  ...
14
15  s1 = (TStack) o1;
    if (s1.$TOwner != $Owner.THISTHREAD()) throw new ClassCastException();
16  s2 = (TStack) o2;
    if (s2.$TOwner != $Owner.SELF()) throw new ClassCastException();
17  q1 = (TStack2) s1;
    if (q1.$TOwner != $Owner.THISTHREAD()) throw new ClassCastException();
18  q2 = (TStack2) s2;
    if (q2.$TOwner != $Owner.SELF()) throw new ClassCastException();
19  ...
20  boolean b1 = ((o1 instanceof TStack) && (((TStack) o1).$TOwner == $Owner.THISTHREAD()));
21  boolean b2 = ((o2 instanceof TStack) && (((TStack) o2).$TOwner == $Owner.SELF()));
```

Fig. 14. Translation of TStack Client Code in Figure 12

contains two static methods that return objects that represent the thisThread owner and the self owner respectively. The translation rules are presented in Figure 10. Section 3.4 explains the translation process with examples.

3.4 Implementation

This section illustrates with examples how our implementation preserves ownership information at runtime for classes with named owner parameters. If a Safe Concurrent Java (SCJ) program is well-typed with respect to the rules for static type checking, our implementation translates the program into an equivalent Java program. (Actually, our implementation translates a SCJ program into Java bytecodes directly. But for ease of presentation, we will describe an equivalent translation into Java code.) The translation mechanism is illustrated in Figures 11, 12, 13, and 14. Figure 11 shows a TStack class with anonymous owners. Figure 12 shows client code that uses the TStack class. Figures 13 and 14 show the translation of the TStack code and the client code.

Classes Classes in the translated code contain extra owner fields, one for each named owner parameter. For example, in Figure 13, the translated TStack class has an extra $TOwner field. The translated TNode class has two extra fields: $thisOwner and $TOwner. The translated T class has no extra fields since the T class does not have any named owner parameters.

Constructors Constructors in the translated code contain extra owner arguments, one for each named owner parameter of the class. The constructors in the translated code initialize the owner fields of the class with the owner arguments of the constructor. For example, in Figure 13, the constructor for TStack has an extra $TOwner argument. The constructor initializes the $TOwner field of the TStack object from the $TOwner argument.

Allocation Sites Allocation sites in the translated code must pass extra owner arguments to constructors, one for each named owner parameter of the corresponding class. If the owner is an expression that evaluates to an object, the client code passes the object to the constructor. For example, in Figure 13, the push method in TStack passes the this object as the first argument to the TNode constructor. If the owner is a formal parameter, the client code passes the value of the formal parameter stored in one of its extra owner fields. For example, in Figure 13, the push method in TStack passes the value stored in the $TOwner field as the second argument to the TNode constructor. If the owner is thisThread or self, the client code passes the object returned by $Owner.THISTHREAD() or $Owner.SELF() to the constructor. For example, in Figure 14, the client code creates TStacks s1 and s2 by passing $Owner.THISTHREAD() and $Owner.SELF() to the TStack constructor respectively.

Casts Casts in the translated code not only check that the Java types match, but also check that the owners match. For example, in Figure 14, in Line 15, the translated code not only checks that o1 is of Java type TStack, but also checks that the owner of the T elements in the TStack is thisThread. In Line 16, the

translated code not only checks that o2 is of Java type TStack, but also checks that the owner of the T elements in the TStack is self.

InstanceOf The instanceof operation in the translated code returns true iff the Java types match and the owners match. For example, in Figure 14, in Line 20, instanceof returns true iff o1 is of Java type TStack and the owner of the T elements in the TStack is thisThread. In Line 21, instanceof returns true iff o2 is of Java type TStack and the owner of the T elements in the TStack is self.

Arrays The technique described in this paper does not support safe runtime downcasts to array types. This is because we cannot add extra owner fields to array objects in the translated code and yet remain JVM-compatible. If a programmer wants to downcast from java.lang.Object to an array type in our system, the programmer can create a wrapper object that contains the array object and perform the downcast on the wrapper object.

Parameterized Methods Parameterized methods are handled similar to parameterized classes. For ease of presentation, the MSCJ language we described in Section 2 has only parameterized classes but not parameterized methods. But our implementation handles both parameterized classes and parameterized methods. Named owner parameters of methods are explicitly passed as arguments to the methods in the translated code.

4 Conclusions

This paper describes an efficient technique for supporting safe runtime downcasts in a system with ownership types. This technique uses the type passing approach, but avoids the associated significant space overhead by storing only the runtime ownership information that is potentially needed to support safe downcasts. The technique preserves the separate compilation model of Java and is JVM-compatible: it translates programs to bytecodes that can be run on regular JVMs.

Acknowledgments

This research was supported by DARPA/AFRL Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-73513, and the Singapore-MIT Alliance.

References

1. O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1997.
2. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

3. C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. Technical Report TR-853, MIT Laboratory for Computer Science, June 2002.
4. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.
5. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, January 2003.
6. C. Boyapati, B. Liskov, L. Shrira, C. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2003.
7. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.
8. C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Programming Language Design and Implementation (PLDI)*, June 2003.
9. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.
10. R. Cartwright and G. Steele. Compatible genericity with run-time types for the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.
11. D. G. Clarke. Object ownership and containment. PhD thesis, University of New South Wales, Australia, July 2001.
12. D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.
13. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.
14. D. G. Clarke and T. Wrigstad. External uniqueness is unique enough. In *European Conference for Object-Oriented Programming (ECOOP)*, July 2003.
15. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
16. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI)*, June 2002.
17. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
18. A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Principles of Programming Languages (POPL)*, January 1997.
19. M. Viroli. Parametric polymorphism in Java: An efficient implementation for parametric methods. In *Symposium on Applied Computing (SAC)*, March 2001.
20. M. Viroli and A. Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.

Cheaper Reasoning with Ownership Types – Work in Progress –

Matthew Smith and Sophia Drossopoulou
{mjs198,sd}@doc.ic.ac.uk

Imperial College London

Abstract. We use ownership types to facilitate program verification. Namely, an assertion that has been established for a part of the heap which is unaffected by some execution will still hold after this execution. We use ownership and effects, and extend them to assertions to be able to make the judgement as to which executions do not affect which assertions. We describe the ideas in terms of an example, and outline the formal system.

1 Introduction

We outline ways in which ownership types can aid program verification in the context of a Hoare logic for a language with a type and effects ownership system. Previous work [4] describes Joe_1, a Java-like language with such a type system, where *disjoint* types characterise disjoint regions of the heap. Importantly, heap disjointness can be deduced statically.

In this paper we make use of these results to enhance a system for program reasoning: We assume a (sound) underlying Hoare logic for a language like Joe_1. We then describe "read effects" for assertions in terms of the effects from [4]. We can then extend the Hoare logic with a further rule, which states that assertions are preserved by executions whose write effect is disjoint from the assertions' read effect. We argue that the extended Hoare logic is still sound. In this paper we are assuming an underlying Hoare logic with non-standard assertions which allow predicate definitions. We are making that assumption because we believe that this allows for a natural way of reasoning about programs, and also because it allows us to express a compelling example. However our approach is orthogonal to the choice of underlying Hoare logic.

Section 2 discusses alternative approaches to dealing with aliases in program verification and the logics of separation. Section 3 introduces our example. Section 4 introduces Joe_1 through extension of our example with ownership types and discusses the features and properties of Joe_1 essential to our work. Section 5 discusses our contribution in terms of additions to a Hoare Logic system. Section 6 revisits our example in terms of the formal system. Section 7 discusses further directions for this work and concludes.

2 Related Work

Aliasing is a fundamental part of the OO paradigm but it makes reasoning about programs more cumbersome [9]. In a setting with aliasing, changing one object may affect properties of another. Several existing works on reasoning in the presence of aliasing, for OO languages at least, focus on reasoning safely in a language with pointers. [3] describes a system whereby one can safely reason about programs with pointers. The Hoare systems of [14] and [13] deal with a Java subset and so naturally deal with aliasing.

Our work differs from these in that we do not attempt to develop a Hoare logic for reasoning in the presence of aliasing, rather we *extend* such a Hoare logic. We use knowledge provided from static typing about the areas affected by execution, and the areas affecting assertions, in order to be able to deduce *preservation* of assertions in a simple way.

Attacking similar problems from a different direction (and at a different level of abstraction) is work on separation logics [15, 11]. Separation logics introduce logical primitives (* for example) which state that two areas of a heap are separate (disjoint). Assertions about one area of the heap remain unchanged when a separate area is modified. The frame rule of separation logics states that a Hoare sentence {P*R}C{Q*R} is valid if the sentence {P}C{Q} is valid and the computation C does not modify any of the free variables of R. The rule we introduce in Sec. 5.5 bears close resemblance to this. We achieve these goals through static typing and at a higher level.

3 Example

Figure 1 shows a simple project management example, written in a Java-like language . An `Employee` has a `Timetable`, a linked-list of `Project`/`Duration` pairs. We propose two properties for the `Timetable`s of an `Employee`, `e`:

`nonO(e)` The entries in `e.t` are non-overlapping (with respect to their start and finish dates)

`durInP(e)` The duration of each entry in `e.t` is contained within the duration of the project for that entry

We allow `Timetable`s to change and consider a simple mutation whereby all an employee's `Timetable` entries are delayed by some number of days i.e. the method `delay` in class `Employee`.

Suppose we wish to prove that the properties `nonO` and `durInP`, of an object `e`, are preserved when executing the `delay` method of `Employee` on `e`.

The possible side effects of performing `delay` on `e` make such assertions difficult to prove. `Employee`s might share `Timetable` objects and thus performing delay on `e` might affect the properties of any number of other `Employee`s. To ensure the properties hold through execution of `delay` we would need to directly reason about employees which might alias `Timetable`s of `e`. For example, assume two `Employee`s, `alice` and `bob`, which are guaranteed not to be

```
class Duration{
    int start;
    int end;

    void shunt(int period){
        start += period;
        end += period;
    }
}

class Employee{
    Timetable t;
    void delay(int period){
        // invokes delay on t
    }
}
```

```
class Timetable{
    Project p;
    Duration d;
    Timetable next;

    void delay(int period){
        // shunts d by period
        // and call on next
    }
}

class Project{
    Duration d;
}
```

Fig. 1. Project Example in almost-Java

aliases. Assume also, that for some program point, `p1` we have established that `nonO(alice.t)`, `nonO(bobt.t)`, `durInP(alice.t)` and `durInP(bob.t)`. Then we execute `alice.delay(23)`. Because of the potential for aliasing, all these assertions need to be re-established not just those pertaining to `alice`. However, if we knew that `bob`'s `Timetable` was not accessible through `alice`, we would argue that `alice.delay(23)` cannot affect `bob`'s timetable and expect that the assertions `nonO(bob.t)` and `durInP(bob.t)` should be preserved through execution of `alice.delay(23)`.

Ownership types [6, 4] give the machinery to restrict, through the type system, which objects may be accessible from which other objects. Joe_1, the extension of ownership to effects and disjointness ([4]), gives the machinery to deduce when certain expressions will not affect certain areas of the heap. In this paper we use the machinery from [4] to extend a Hoare logic.

4 Joe_1

Joe_1 is a Java-like language with ownership types and effects [4]. We give a brief outline here, which we hope allows an intuitive understanding of our current work.

In Joe_1 types are parameterised with ownership *contexts*(p,q etc.). Classes are defined using context variables which are bound to objects or the global context, `world`, when types are instantiated.

In Fig. 2 we add ownership types (and effects) to the example of Fig. 1. Figure 3 shows a possible instantiation of classes from Fig. 2. The rounded boxes represent objects in a UML-like notation. For example `alice:Employee` represents the object `alice` of class `Employee`. The square boxes in the diagram represent the ownership hierarchy, the box on the border being the owner of those immediately inside. Arrows are references i.e. fields. Note that as one expects for

ownership systems, arrows can break out of boxes but not into them[1]. All the employees and projects are owned by some higher object, the company or department perhaps. Each `Employee` owns their `Timetable` objects, the `Timetable`s in turn own the underlying duration objects.

```
class Duration<owner>{
    int start;
    int end;

    void shunt(int period)
            wr this {
        start += period;
        end += period;
    }
}

class Employee<owner, proj>{
    Timetable<this, proj> t;
    void delay(int period)
            wr under(this) {
        // invokes delay on t
    }
}
```

```
class Timetable<owner, proj>{
    Project<proj> p;
    Duration<this> d;
    Timetable<owner,proj> next;

    void delay(int period)
            wr under(this){
        // shunts d by period
        // and call on next
    }
}

class Project<owner>{
    Duration<this> d;
}
```

Fig. 2. Project Example in with ownership types in Joe_1

Joe_1 differs from the original ownership types systems [6] as it allows (in controlled circumstances) access to objects across ownership boundaries i.e. breaking into boxes. `let` statements in Joe_1 allow assignment of any reachable statement to a local variable. The variable name can then be used as an ownership context and replaces `this` in types of objects from inside an ownership box. This is safe as the variable is a "short lived dynamic alias"–to borrow the terminology of [4]. The variable is visible only inside the `let` statement and the type system will not allow the referenced object to 'leak' out of the let through methods calls etc. We shall also be making use of these `let` statements in our assertions and type system though at some times we shall omit them in either context for brevity.

Two contexts are *disjoint* when they refer to different objects or one is `world` and the other isn't. Disjointness can be reasoned from type/ownership information stored in the typing environment. Disjointness of types is based on the class hierarchy and the disjointness of contexts. Types are disjoint when i) one is not a subclass of the other, or ii) any of their ownership parameters are disjoint, or iii) one is a subtype of some type which is disjoint from the other.

Since we know that each Employee owns its `Timetable` and each `Timetable` owns its durations we also know that no `Timetable` object can be shared between two `Employee`s (they have disjoint types). Thus, changes which only affect

[1] Therefore `alice` may not point to `t3` nor to `d1`.

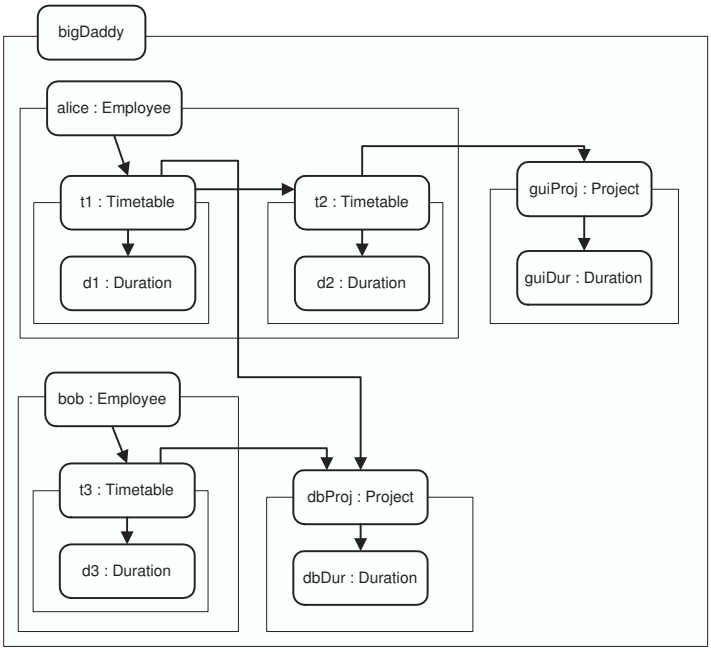

Fig. 3. Possible instantiation of code of Fig. 2

alice's Timetable leave bob's Timetable unaffected. Hence the properties nonO and durInP would be preserved for bob.

Types in $\texttt{Joe}_1$ come paired with an *effect* which describes where, in the heap, the typed expression will read or write. The read/write components are described by *shapes* (ϕ, ϕ' etc.) which characterise the affected portions of the heap. Thus effects have the form $\texttt{rd}\,\phi\ \texttt{wr}\,\phi'$ ranged over by ψ. The typing judgement of $\texttt{Joe}_1$ has the form $E\ \vdash\ e : t\ !\psi$ meaning that with respect to some type environment, E, the expression e has type t and produces at most the effect ψ when executed. For example:

$\texttt{alice:Employee}{<}x,x{>},x \prec^* x\ \vdash\ \texttt{alice.delay(23):void}\ !\texttt{wr}\ \texttt{under(alice)}$

for some x. Type environments store information of two kinds. The first is the conventional binding of program variables (including this) to types. The second kind of information is in the form of ordering of ownership contexts. $\texttt{Joe}_1$ puts certain restrictions on ownership parameters, specifically that the owner of an object should be inside all the other contexts. Along with additional rules this information enables us to reason properties such as disjointness.

Shapes describe sections of the heap in terms of contexts and are based on the ownership hierarchy. A shape could be just the object bound to a context parameter p, say. The shape $\texttt{under}(p)$ refers to the union of all the objects directly or indirectly owned by p. $\phi \cup \phi'$ is the shape derived from set union of ϕ and ϕ'.

It is possible to reason not only disjointness of types but also of shapes based on some type environment E. Two shapes are disjoint just when they have no objects in common. Read $E \vdash \phi\ \#\ \phi'$ as ϕ and ϕ' are disjoint. One can deduce some disjointness of context from the type environment and further disjointness of shapes follows intuition from set theory. For example, if two contexts, p, q, are disjoint and have a common owner then the shapes $\texttt{under}(p)$ and $\texttt{under}(q)$ are also disjoint.

Soundness of the shapes of expressions is an important result of [4]. Lemma 6.1 of that paper states that if an expression e having type t and effect $\texttt{rd}\,\phi\ \texttt{wr}\,\phi'$ is executed in the context of some heap H and stack S, then the areas of the heap not contained in ϕ' are unchanged in execution[2]. This result is important for this work as we apply the effects of $\texttt{Joe}_1$ to assertions.

5 Hoare Logics with Ownership

5.1 Assertions

We assume a logic with assertions, based on first order logic with predicates. Π, a predicate declaration, consists of a name, p, a list of arguments with their types, a shape ϕ, and a predicate body, B. ϕ is the declared shape of a predicate, in terms of the ownership parameters of the arguments and the arguments

[2] [4] includes a method of projecting an effect from its abstract contexts onto its underlying heap locations

themselves. Assertions and predicates are as follows:

$$
\begin{array}{lll}
\Pi & ::= & p\ (arg^*)\phi\ (B) \\
\Pi s & ::= & \Pi^* \\
arg & ::= & t\ z \\
z & ::= & \texttt{this} \mid x \\
B & ::= & \texttt{let}\ x\ = a\ \texttt{in}\ B \mid A \\
A & ::= & A_0\ \wedge A_1 \mid \neg(A) \mid \texttt{ff} \mid a_0 = a_1 \mid p\ (z^*) \\
a & ::= & z \mid z.f \mid \texttt{null}
\end{array}
$$

where `this` is the current receiver, x ranges over local programme variables and predicate parameters and f ranges over field identifiers. Assertion expressions, a, are a restricted form of program expressions without side-effects.

$a_0 = a_1$ states that the two expressions are equal i.e. they refer to the same heap location. Conjunction and negation are as one expects. p ranges over predicate names and $p\ (\overline{z})$ is the application of a predicate.

We also employ the `let` statements of Joe_1 to allow predicates to look inside ownership boundaries. This is safe not only by virtue of safety of Joe_1 but intuitively as assertions (particularly equality) have no computational effect, as no reassignment is occurring no ownership properties can be compromised.

We shall adopt the $\overline{over scored}$ notation observed in [10] meaning an indexed sequence, for example $\overline{t\ x}$ stands for $t_1\ x_1, \ldots, t_n\ x_n$ and $\overline{a}$ stands for $a_1, \ldots, a_m$.

Note that we allow a very powerful language for predicates - much more powerful than [14, 11] as we allow predicate definitions. We need such predicates in order to express the example of Sec. 3. We omit quantification since it is not useful for our example. As we said in the Introduction, the main thrust of this work is orthogonal to the choice of Hoare logic. If it should prove difficult to find such a logic we are confident that we shall be able to apply our work to a more usual one.

Example The following predicate definition, declared with respect to the code of Fig. 2 captures `nonO` from Sec. 3. It exploits recursion to traverse the list of `Timetable`. For convenience we assumed that the entries in `t` are ordered. We omit the details of the shape ϕ for now. In this example we write out `let` statements in full, in future we shall omit them for brevity's sake.

```
nonO(Timetable<o, po> t)φ (
   let dur = t.d in
      let next = t.next in
         let ndur = next.dur in
            let start = ndur.start in
               end≤start ∧ nonO(next)
)
```

5.2 Shapes of Assertions

The judgement $E \vdash B{:}\phi$ states that the assertion body, B, depends on the part of the heap characterised by ϕ, with respect to the type environment E. By depends we mean that all the variables, fields etc. referenced are in a part of the heap covered by ϕ. We take the program `P` and the predicate declarations Πs as implicit. The judgement is defined as follows:

$$\frac{E \vdash A_0 : \phi_0 \qquad E \vdash A_1 : \phi_1}{E \vdash A_0 \wedge A_1 : \phi_0 \cup \phi_1}\ \text{ASS-CONJ} \qquad \frac{E \vdash A : \phi}{E \vdash \neg(A) : \phi}\ \text{ASS-NEG}$$

$$\frac{\begin{array}{l} p(\overline{t\ x})\phi(A) \in \Pi s \\ E \vdash \overline{z} : \sigma(\overline{t})!\overline{\psi} \end{array}}{E \vdash p(\overline{z}) : \sigma[\overline{x} \mapsto \overline{z}](\phi)}\ \text{ASS-PRED} \qquad \frac{\begin{array}{l} E \vdash a_0 : t_0 !\mathtt{rd}\ \phi_0 \\ E \vdash a_1 : t_1 !\mathtt{rd}\ \phi_1 \end{array}}{E \vdash a_0 = a_1 : \phi_0 \cup \phi_1}\ \text{ASS-EQ}$$

$$\frac{\begin{array}{l} E\ \vdash a : t\ !\mathtt{rd}\ \phi_0\ \ \mathtt{wr}\ \phi_1 \\ E, x : t \vdash B : \phi' \\ E, x : t \vdash \phi' \sqsubseteq \phi \\ E \vdash \phi \end{array}}{E \vdash \mathtt{let}\ x = a\ \mathtt{in}\ B : \phi}\ \text{ASS-LET} \qquad \frac{}{E \vdash \mathtt{ff} : \emptyset}\ \text{ASS-FF}$$

The rules for evaluating the shape of a judgement are similar to those of Joe_1. In rules ASS-EQ, ASS-RED and ASS-LET we use the typing judgement of Joe_1 (as described in Sec. 4) to type the assertion expressions, a, since these are valid Joe_1 syntax. Note that we do not include the shape of the parameters (in rule ASS-PRED) or a (in rule ASS-LET) since we are only interested in the shape of the assertions; we do not accumulate computational effect as in Joe_1. The shape of a predicate is declared in terms of the declared contexts of its parameters and the parameters themselves. The shape of a predicate application is the declared shape under the substitution which maps the ownership variables of the predicate declaration to those of the supplied arguments and the formal arguments to the actual parameters.

An assertion, A, and the predicate definitions, Πs, are well-formed with respect to a type environment (and also implicitly the program and the predicate declarations) if all assertion expressions a, are well-formed, and types of arguments to predicates match the declared types. Using the type system of Joe_1 to find the shape of a_0 and a_1 has the added benefit that assertions which have a shape are 'well-formed' i.e. expressions of the form $z.f$ are type correct and thus we needn't check elsewhere that field, f exists.

Well-formedness of predicate definitions must be checked separately to ensure the declared shape is valid for the body of the predicate.

$$\frac{\overline{t\ x}, constr(\overline{t\ x}) \vdash B : \phi' \qquad \overline{t\ x}, contr(\overline{t\ x}) \vdash \phi' \sqsubseteq \phi}{\vdash p(\overline{t\ x})\phi(B)}$$

A predicate declaration is well-formed if i) the type environment generated from the declared types of its parameters is valid, ii) the shape of its body, in the context of the constructed environment is well-formed with shape ϕ', and iii) the calculated shape of the body is a subshape of the declare shape (which also requires that ϕ is well-formed w.r.t. the environment. The type environment pairs the formal parameters to their declared types and has ownership constraints (generated by *constr*) as required by $\texttt{Joe}_1$ (we omit details).

Note that the body of the predicate, A, is typed in the environment $\overline{t\,x}$, created by the formal parameters of the predicate. Therefore the identifier `this` may not appear in A (nor indeed any variable names other than the formal parameters), thus reflecting the fact that the predicates are not declared within classes.

Example The shape `under(o)` would be a valid shape for the predicate `nonO` defined previously[3].

5.3 Satisfaction of Assertions

We express satisfaction of an assertion with respect to a heap, H, and a stack, S, by the *partial* function $\mathcal{S}$ mapping assertions, heaps and stacks (and implicit predicate declarations) to $\{\texttt{tt}, \texttt{ff}, \bot\}$[4]

$\mathcal{S}$ is defined as follows:

$$
\begin{array}{ll}
\mathcal{S}(H,S,\texttt{let}\ x = a\ \texttt{in}\ B) & = \mathcal{S}(H, S[x \mapsto (H,S)(a)], B) \\
\mathcal{S}(H,S,A_0 \wedge A_1) & = \mathcal{S}(H,S,A_0) \wedge \mathcal{S}(H,S,A_1) \\
 & \quad \text{if } \mathcal{S}(H,S,A_i) \neq \bot \quad i \in \{0,1\} \\
\mathcal{S}(H,S,\neg(A)) & = \neg(\mathcal{S}(H,S,A)) \text{ if } \mathcal{S}(H,S,A) \neq \bot \\
\mathcal{S}(H,S,\texttt{ff}) & = \texttt{ff} \\
\mathcal{S}(H,S,a_0 = a_1) & = (H,S)(a_0) = (H,S)(a_1) \text{ if } (H,S)(a_i) \neq \bot \quad i \in \{0,1\} \\
\mathcal{S}(H,S,p(\overline{z})) & = \mathcal{S}(H,S,A[\overline{x}/\overline{z}]) \text{ where } p(\overline{t\,x})\phi(B) \in \Pi s \\
\mathcal{S}(H,S,A) & = \bot \text{ otherwise}
\end{array}
$$

The auxiliary function $(H,S)(\dots)$ maps expressions to the appropriate heap location.

[3] Since `int`s are passed by value in Java we assume they are `under` the object referencing them

[4] $\mathcal{S}$ has to be a *partial* function since we can define inconsistent predicates in our system, for example:

$$p\ (\text{c}{<}\dots{>}\ x)\emptyset\ (\neg(p\ (x)))$$

Alternatively we could have defined satisfaction through the interpretation of predicates from Πs into H,S as e.g. in [1]. The function $\mathcal{S}$ is undecidable in the general case however this does not affect the applicability of our result.

$$
\begin{array}{ll}
(H,S)(x) & = S(x) \\
(H,S)(a.f) & = \begin{cases} H((H,S)(a))(f) \text{ if } (H,S)(a) \neq \bot, \texttt{null} \\ \bot \text{ otherwise} \end{cases} \\
(H,S)(\texttt{this}) & = S(\texttt{this}) \\
(H,S)(\texttt{null}) & = \texttt{null}
\end{array}
$$

5.4 The Assumed Hoare Logic

Suppose there exists a Hoare Logic for the language with sentences of the form:

$$E \vdash A_0 \ \{e\ \} A_1$$

where E is a typing environment, binding `this` and local variables to types. The program P, and predicate definitions Πs are taken to be implicit. Soundness of the Hoare logic is defined as follows:

Definition 1. *The Hoare logic is sound if:*
$\forall E, A_0, A_1, e, H, H'\ S, v$*:*

$$\left.\begin{array}{l} E \vdash H \\ E\ \vdash e : t\ !\psi \\ < H; S; e > \xrightarrow{\psi'} < H'; v > \\ \mathcal{S}(H,S,A_0) = \texttt{tt} \\ E \vdash A_0\ \{e\ \} A_1 \end{array}\right\} \Rightarrow \mathcal{S}(H',S,A_1) = \texttt{tt}$$

5.5 Hoare Logic Extension

Assuming a system as described previously, we define an *ownership extended* Hoare logic as follows:

$$\frac{E \vdash A_0\ \{e\ \} A_1 \qquad E \vdash A : \phi \qquad E\ \vdash e : t\ !\texttt{rd}\ \phi_0\ \ \texttt{wr}\ \phi_1 \qquad E \vdash \phi\ \#\ \phi_1}{E \vdash_O A_0\ \wedge A\ \{e\ \} A_1\ \wedge A}$$

The first judgement, $E \vdash A_0\ \{e\ \} A_1$, is a derivation of the assumed Hoare logic. It does not utilise the properties of ownership types. The second judgement, $E \vdash A{:}\phi$, is the calculation of the shape of the assertion, A, as described in Sec. 5.2. $E\ \vdash e : t\ !\texttt{rd}\ \phi_0\ \ \texttt{wr}\ \phi_1$ is the typing judgement of $\texttt{Joe}_1$ as described in Sec. 4. $E \vdash \phi\ \#\ \phi_1$ is the disjointness judgement described in Sec. 4. Thus, the Hoare logic rule we propose states that any assertion which is true before the execution of an expression and which inhabits a part of the heap which is not written to by the expression, is true after execution of the expression. Note that the subscript on $\vdash$ makes the distinction between judgements of the original and the "ownership-aware" Hoare logic.

The following conjecture states that the extension we suggest preserves soundness of the Hoare logic.

Conjecture 1. If a Hoare logic $\vdash$ is sound, then the ownership aware extension, $\vdash_O$, is also sound.

We prove that conjecture using the following conjecture which states that execution of an expression e with disjoint shape from assertion A preserves the satisfaction of that assertion.

Conjecture 2.

$$\left.\begin{array}{l} E \ \vdash e :t \ !\mathtt{rd}\,\phi_0 \ \ \mathtt{wr}\,\phi_1 \\ E \vdash A : \phi \\ E \vdash \phi \ \# \ \phi_1 \\ E \vdash H \\ < H; S; e > \xrightarrow{\psi} < H'; v > \\ \mathcal{S}(H, S, A) = \mathtt{tt} \end{array}\right\} \Rightarrow \mathcal{S}(H', S, A) = \mathtt{tt}$$

We expect this to follow as a corollary of Lemma 6.1 in [4] which states that on execution of some instruction the portion of the heap which is not in the shape written by the execution is unchanged.

6 Once More with Formality

We shall now roughly show how one might apply our system in the context of the example of Fig. 2.

Consider the following set of predicate definitions, intended to capture the properties discussed in Sec. 3:

Πs =

```
within(Duration<o1> d1, Duration<o2> d2) under(o1) ∪ under(o2)
    ( d1.start ≥ d2.start ∧ d1.end ≤ d2.end )

nonO(Timetable<o, po> t) under(o)
    ( t.next ≠ null⇒
        t.d.end ≤ t.next.d.start ∧ nonO(t.next) )

durInP(Timetable<o,op> t) under(op)
    ( t.next ≠ null⇒ within(t.d,t.p.d) ∧ durInP(t.next) )


nonO(Employee<o,op> e) under(() o)
    ( nonO(e.t) )

durInP(Employee<o,op> e) under(op)
    ( durInP(e.t) )
```

Suppose:

$$E = \texttt{alice:Employee<bigDaddy, bigDaddy>},\ \texttt{bob:Employee<bidDaddy, bigDaddy>}$$

and we also know that $E \vdash \texttt{alice\#bob}$ (which we have if we know `alice` and `bob` are not aliases).

By application of type rules of [4] we obtain that:

$$E \vdash \texttt{alice.delay(23)} : \texttt{void!wr under(alice)}$$

By further application of the rules of [4] and those of Sec. 5.2 we obtain

$$E \vdash \texttt{nonO(alice)} : \texttt{under(alice)}$$
$$E \vdash \texttt{durInP(alice)} : \texttt{under(bigDaddy)}$$
$$E \vdash \texttt{nonO(bob)} : \texttt{under(bob)}$$
$$E \vdash \texttt{durInP(bob)} : \texttt{under(bigDaddy)}$$

Therefore, by application of the Hoare logic extension rule, we obtain that:

$$E \vdash_O \texttt{nonO(bob)}\ \{\ \texttt{alice.delay(23)}\ \}\ \texttt{nonO(bob)}$$

Furthermore, if we were able to derive (the hard way) that:

$$E \vdash \texttt{nonO(alice)}\ \{\ \texttt{alice.delay(23)}\ \}\ \texttt{nonO(alice)}$$

then we could also obtain (the cheap way) through the extension rule that:

$$E \vdash_O \texttt{nonO(bob)} \wedge \texttt{nonO(alice)}\ \{\ \texttt{alice.delay(23)}\ \}\ \texttt{nonO(bob)} \wedge \texttt{nonO(alice)}$$

So, we were able to obtain the preservation of the assertion `nonO(bob)` after the execution of `alice.delay(23)` through mechanisms of the type system.

Note, however, that we are not able to deduce that:

$$E \vdash_O \texttt{durInP(bob)}\{\texttt{alice.delay(23)}\}\texttt{durInP(bob)}$$

even though we know `alice.delay(23)` only shifts the timetable of `alice` and thus does not affect the validity of `durInP(bob)`. This is so because in our current system the effects/shapes are too coarse. In further work we aim to refine shapes through e.g. data groups [7, 12].

7 Conclusions and Further Work

We have shown how to extend a Hoare logic with ownership awareness and how this will support locality of reasoning, thus allowing the re-establishing of properties after execution of some code in a cheap way.

Our approach is "parametric" in the choice of the underlying Hoare logic. Any logic which supports assertions of the kind described in Sec 5.1 can be extended and soundness will be preserved. Thus we should be able to "plug" and experiment with various such logics.

Our approach should also be extensible to be made "parametric" with respect to the containment system. We have used ownership types in order to be able to isolate the shape of expressions and assertions. We should be able to generalise to any approach which supports the expression of shapes of expressions and notions of disjointness, e.g. datagroups, balloons etc.

The current extension of the Hoare logic is limited in that we may only apply our extension at the last step. The full extension of the the Hoare logic would allow the application of the new rule at any part of the derivation. We have not considered this here as we were not sure how to prove preservation of soundness. It has been recently pointed out to us [5] that if all rules of the underlying system preserve soundness then the extension also preserves soundness. We shall describe the full extension in future work.

As well as the above, further work includes:

- investigation of suitable Hoare logics for application of these results
- details and proofs of the presented work
- further examples which will demonstrate the benefits or point to further extensions
- an extension of Joe_1 to support datagroups [7, 12, 2] so that we can better localise the shapes and effects
- incorporation of further ideas from ownership e.g. uniqueness [5]

8 Acknowledgements

We are grateful to Dave Clarke, John Potter and our colleagues from SLURP for useful discussions. We are particularly grateful to Mariangiola Dezani and Joe Wells for their help and suggestions for the full extensions of Hoare logic. We would also like to thank the anonymous referees for important and valuable comments.

References

[1] Krzysztof R. Apt. Ten years of hoare's logic: A survey.part i. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

[2] G.M. Bierman and M.J. Parkinson. Effects and effect inference for a core java calculus. In *WOOD 2003*. ENTCS, 2003.

[3] Richard Bornat. Proving pointer programs in hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[5] Private communication with Joe Wells and Mariangiola Dezani

[4] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 292–310. ACM Press, 2002.

[5] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP 2003*, 2003.

[6] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 48–64. ACM Press, 1998.

[7] Aaron Greenhouse and John Boyland. An object-oriented effects system. *Lecture Notes in Computer Science*, 1628:205–230, 1999.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[9] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[10] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. *Lecture Notes in Computer Science*, 1850:129–154, 2000.

[11] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[12] K. Rustan M. Leino. Data groups: specifying the modification of extended state. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 144–153. ACM Press, 1998.

[13] Cees Pierik and Frank S. de Boer. A syntax-directed hoare logic for object-oriented programming concepts. http://www.cs.uu.nl/research/techreps/UU-CS-2003-010.html, 2003.

[14] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symosium un Programming (ESOP '99)*, volume 1576, pages 162–176. Springer-Verlag, 1999.

[15] J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002.

Formalising Eiffel References and Expanded Types in PVS

Richard Paige[1], Jonathan Ostroff[2], and Phillip Brooke[3]

[1] Department of Computer Science, University of York, UK. paige@cs.york.ac.uk
[2] Department of Computer Science, York University, Canada. jonathan@cs.yorku.ca
[3] School of Computing, University of Plymouth, UK. philb@soc.plym.ac.uk

Abstract. Ongoing work is described in which a theory of Eiffel reference and expanded (composite) types is formalised. The theory is expressed in the PVS specification language, thus enabling the use of the PVS theorem prover and model checker to semi-automatically prove properties about Eiffel structures and programs. The theory is being used as the basis for automated support for the Eiffel Refinement Calculus.

1 Introduction

There is a definite need to be able to formally reason about *models* of object-oriented (OO) systems, e.g., as written in UML, in order to be able to analyse requirements and verify and validate design alternatives. Such techniques can let us discover errors, omissions, and ambiguities early in the system development process. Equally, it is vitally important to be able to reason formally about object-oriented *programs*, written in industrial-strength OO programming languages such as Java, C++, Eiffel, and Smalltalk. This is challenging for a number of reasons:

- Industrial-strength OO languages are typically very large, with numerous features that are not easily formalised, and for which formal reasoning can be difficult, such as pointers, deep loop exits, dynamic dispatch, and aggregate types.
- Industrial-strength OO languages are invariably complex, with sophisticated type systems that enable efficient execution and programming, but which are not necessarily designed for formal reasoning.
- Industrial-strength OO programs are usually very large, thus presenting a need for monotonic and compositional reasoning.

Despite these obvious difficulties, the need for reasoning about industrial-strength OO programs and programming languages remains: if engineers are to adopt reasoning techniques and tools, these mechanisms must support the languages and techniques that are applied in practice.

There is ongoing work on supporting reasoning about OO programs. Some of this focuses on defining new precise OO specification languages that can be translated into more widely used programming languages such as Java and C++. Perfect Developer [Es00] is an example of this, supporting formal OO specification, theorem proving, and code generation; it requires learning a new specification language, but thereafter

can generate code in a number of widely used languages, such as C++. The work of Cavalcanti et al [CN00] has produced a wp semantics for a subset of typical OO programming features; this semantics can thereafter be used to define a refinement calculus for transforming OO specifications into programs.

Other work has tackled the complexities of reasoning about OO programming languages head on. The LOOP project [BJ01] has focused on producing theories for the PVS system [CO95] that allow reasoning about Java programs; this should be contrasted with our work which is intended to support reasoning about specifications and transforming specifications into executable programs. The Extended Static Checker [LN00] provides a lightweight approach to verifying Java programs via source code annotation and a `lint`-like tool interface that detects many typical OO programming bugs at compile time, e.g., null reference access.

This paper reports on ongoing work on formalising a theory of reference and expanded types in the Eiffel programming language [Mey92]. A set of axioms expressing this theory was proposed in [PO03]. Meyer also has ongoing work on formalising parts of the Eiffel language using Hoare triples [Mey03]. This paper extends [PO03] by showing how the axioms can be formalised in the PVS language; the PVS theorem prover, model checker, and ground evaluator can thereafter be used to semi-automatically reason about and simulate specifications of Eiffel programs. Immediate applications of the PVS theory would be to support reasoning about aliasing in Eiffel, and about particularly complicated interactions between Eiffel reference types and expanded types. The PVS theories produced form the basis for automated support for the Eiffel Refinement Calculus [PO03].

We commence with a brief introduction to Eiffel and PVS, and attempt to provide some justification for combining the use of these two technologies in the calculus.

1.1 Eiffel

Eiffel is an object-oriented programming language and method [Mey97]; it provides constructs typical of the object-oriented paradigm, including classes, objects, inheritance, associations, composite ("expanded") types, generic types, polymorphism and dynamic binding, and automatic memory management.

However, Eiffel is not just a programming language — it also includes the notion of a *contract* to specify the duties of clients and suppliers. A valid Eiffel program may consist only of contracts – that is, it may possess no program code whatsoever – or it may be a combination of contract and code. or code only. It is in part because of its support for contracts that we have chosen Eiffel as the target language for the refinement calculus referenced in [PO03], and as the specification language to be partially formalised in PVS.

A short example of an Eiffel class is shown in Fig. 1. The class $CITIZEN$ inherits from $PERSON$ (thus defining a subtyping relationship). It provides several attributes, e.g., $spouse, children$ which are of reference type (in other words, $spouse$ refers to an object of type $CITIZEN$); these features are publicly accessible (i.e., are exported to ANY client). Attributes are by default of reference type; a reference attribute either points at an object on the heap, or is $Void$. The class provides one expanded attribute, $blood\_type$. Expanded attributes are also known as composite attributes; they are not

references, and memory is allocated for expanded attributes when memory is allocated for the enclosing object.

The remaining features of the class are routines, i.e., functions (like $single$, which returns $true$ iff the citizen has no spouse) and procedures (like $divorce$, which changes the state of the object). These routines may have preconditions (**require** clauses) and postconditions (**ensure** clauses), but no implementations. Routines may also have a **modifies** clause, which specifies those entities that may be changed by the routine – i.e., the *frame* of the routine. The purpose of the Eiffel Refinement Calculus is to transform such contracts into immediately executable Eiffel code, with a guarantee that the code is consistent with the contract. Finally, the class has an invariant, specifying properties that must be true of all objects of the class at stable points in time, i.e., before any valid client call on the object. While we have used predicate logic in specifying the invariant of $CITIZEN$, it should be observed that Eiffel does not support this exact syntax. It does possess a notion of *agent* that can be used to simulate quantifiers like the ones used in the example.

class $CITIZEN$ **inherit** $PERSON$
feature $\{ANY\}$
    $spouse : CITIZEN$
    $children, parents : SET[CITIZEN]$
    $blood\_type : \mathbf{expanded}\ BLOOD\_TYPE$
    $single : BOOLEAN$ **is**
        ensure $Result = (spouse = Void)$
feature $\{BIG\_GOVERNMENT\}$
    $marry \cdots$
    $have\_child \cdots$
    $divorce$ **is**
        modifies $single, spouse$
        require $\neg\ single$
        ensure $single \wedge (\mathbf{old}\ spouse).single$
invariant
    $single \vee spouse.spouse = Current;$
    $parents.count \leq 2;$
    $\forall\, c \in children \bullet \exists\, p \in c.parents \bullet p = Current$
end

Fig. 1. Eiffel Class Interface

Other facilities offered by Eiffel, but not demonstrated here, include generic (parameterised) types, dynamic dispatch, multiple inheritance, static typing, and agents (function objects). We refer the reader to [Mey92] for full details.

1.1.1 Reference and Expanded Types in Eiffel Eiffel is a novel language in that it supports both user-defined reference variables and *expanded* variables. A reference variable may be attached to an object at run-time; thus, it is effectively a (safe) pointer

to a memory location. The pointer is "safe" in the sense that its address cannot be taken (easily), and pointer arithmetic cannot be applied to it. To use a reference variable requires two steps: declaration of the variable, and allocation/attachment of an object to the variable. This mechanism is somewhat cumbersome and inefficient for basic datatypes, such as integers. Thus, a shortcut is provided wherein a declaration of a variable i of type $INTEGER$ associates i directly with a storage location. This is roughly equivalent to stack allocation of memory in languages such as C and Pascal. Basic types, such as integers, are by default allocated in this way, but Eiffel provides a generalised mechanism by which any variable can be directly associated with a storage location; such variables are declared to be of expanded type. This allows programmers to avoid using reference mechanisms where they deem it appropriate; this should be contrasted with Java, which does not support a similar user-defined notion of expanded type. Interesting compatibility issues arise when using reference and expanded variables together in expressions and, particularly, assignment statements.

1.2 PVS

The PVS system [CO95] combines an expressive specification language with an interactive theorem prover and proof checker. The specification language is founded on classical typed higher-order logic with typical base types `bool`, `nat`, `int`, `real`, etc. It also provides function constructors of the form `[A->B]`. The type system of PVS is augmented with dependent types, abstract data types, and predicate subtypes, the last of which is a distinguishing characteristic of the specification language. The subtype `{x:A|P(x)}` consists of those elements of type `A` that satisfy the predicate `P`. The presence of predicate subtypes means that type checking of PVS specifications is undecidable in general, and thus requires use of the PVS theorem prover. Thus, the type checker generates proof obligations (called *type correctness conditions (TCCs)* in those cases where type conflicts cannot be resolved automatically. Frequently, large numbers of TCCs are discharged automatically by special stategies.

The PVS system has been used successfully in industrial projects, particularly for protocol and microprocessor verification. It is widely considered to be one of the most powerful theorem provers in use today. It is because of the power of the theorem prover, the expressiveness of its specification language, and the wealth of PVS libraries and expertise available in the research community, that we have targeted it as the means for providing automated support for the refinement calculus.

2 Overview of the Theory of Eiffel Reference Types

The paper [PO03] proposes a refinement calculus for Eiffel, which allows specifications – written in a pre- and postcondition style, using Eiffel's built-in support for such facilities – to be refined directly to Eiffel programs. The calculus is built atop Hehner's predicative programming theory [Heh03]. The calculus was produced by formalising the precisely stated semantics of Eiffel statements given in [Mey92] using Hoare logic. At the basis of this formalisation is a theory of Eiffel reference types, based on so-called

entity groups, which are the equivalence classes induced by the reference equality operator in Eiffel. This theory is used in formalising assignment statements, object **create** statements, and method calls. The theory, and its axioms, are based on careful analysis of the partial formalisation of Eiffel's semantics given in [Mey92].

We summarise elements of the theory here. Eiffel programs are made up of a number of classes, each of which possess *routines*; a routine is either a query (which returns a result without side effects) or a command (which changes the state of an invoking object). Routines may declare entities (variables) as local variables; entities may also be introduced as attributes of a class. Classes themselves are connected via associations and inheritance relationships.

Given routine r of class C, we let $r.\rho$ denote the set of reference entities that could appear in the routine body (including attributes of C, arguments and local variables of r, and syntactically legal dots and multi-dots expressions). If routine r is a query then $Result \in r.\rho$ because $Result$ is a predefined local variable of the query. Each entity in $r.\rho$ is potentially the subject of a creation instruction either directly in the body of r itself, or in a routine called by r.

Associated with each entity $e \in r.\rho$ there is a corresponding *entity group* which we denote by $\underline{e}$. Before the first creation statement in the body of r, the group is an empty set. After a $\mathbf{create}\ e$ instruction, $\underline{e} = \{e\}$, and this entity group is used to keep track of all reference entities in $r.\rho$ that point to the same object as e. We let $r.\pi$ denote all entity groups of routine r, and we require that these groups be disjoint (this equivalence relation will be formalized in the sequel).

The boolean expression $equal(e1, e2)$ is used for object equality. It can be defined recursively, provided that its occurrence in a program is syntactically legal; the formalisation is in [PO03]. An axiom is also needed that asserts that reference equality is at least as strong as object equality:

$$e1 \stackrel{r}{=} e2 \rightarrow equal(e1, e2) \tag{1}$$

The remaining axioms defining entity groups are as follows. Consider a routine r with bunch of entity groups $r.\pi$:

$$\forall\, \underline{ei}, \underline{ej} \in r.\pi \ \bullet\ \ \underline{ei} = \underline{ej} \ \vee\ \underline{ei} \cap \underline{ej} = \varnothing \tag{2}$$

Axiom (2) states that two entity groups are either disjoint or identical. Only reference entities are subject to aliasing, and hence only reference entities have associated groups. Thus if $e1$ and $e2$ point to the same object, then they are in the same group.

$$\forall\, \underline{e} \in r.\pi \bullet (\forall\, e1, e2 \in \underline{e} \bullet e1 \stackrel{r}{=} e2) \tag{3}$$

Axiom (3) states that if two entities are in the same entity group, they refer to the same object.

$$\forall\, \underline{e} \in r.\pi \bullet \forall\, e1, e2 \in \underline{e} \bullet \underline{e1} = \underline{e2} \tag{4}$$

(4) asserts that if two entities are in the same group, then the group can be referred to in expressions by either name.

$$\forall\, \underline{e} \in r.\pi \bullet \forall\, e \in \underline{e} \bullet e \neq \mathit{Void} \tag{5}$$

$$\forall\, e \in r.\rho \bullet (\forall\, \underline{e}' \in r.\pi \bullet e \notin \underline{e}') \equiv e = \mathit{Void} \tag{6}$$

$$\forall\, e \in r.\rho \ \bullet \ (e = \mathit{Void}) \ \equiv \ (\underline{e} = \varnothing) \tag{7}$$

Axiom (5) states that any entity in an group is attached to an object, and thus cannot be Void. Axiom (6) states that if an entity e is in no group, then it is Void. This is the state an entity is in after declaration, or after an assignment $e := \mathit{Void}$. Axiom (7) equates a Void reference with an empty entity group. It is perhaps not clear why entity groups have been used to formalise reference types in Eiffel, as opposed to obvious approaches, e.g., formalising a memory model with memory locations, pointers, and objects. One issue with the latter approach is that it can lead to long expressions (typically stating what values are stored in what memory locations) appearing in refinement steps. We desire to make the refinement calculus useful for carrying out refinement steps by hand and with automated assistance. Long expressions do not arise with entity groups, as they effectively let developers express only those entities that are of interest in a refinement step. Certainly, when using PVS it would be reasonable to specify an Eiffel memory model, and to define entity groups in terms of that memory model. In formulating the Eiffel memory model, we would envision using an approach similar to that for the Extended Static Checker for Java [LN00]. We are considering this approach in revisions to our theories, but for now are simply formalising the axioms directly so as to make it easier to validate and check our PVS formulation of the Eiffel refinement calculus.

The notion of entity groups can be used to define the semantics of instructions that use reference types in Eiffel such as the **create** statement, assignment and feature call. The $\mathbf{create}\ e$ instruction creates a new object and attaches it to entity e; any previous reference or attachment via entity e is lost; however, any other entities that referred to the object originally attached to e remain. This continued attachment is established by the previous axioms, specifically (4) and (5). The semantics of $\mathbf{create}\ e$ is given in Definition 1.

Definition 1. *[Entity creation semantics]* Given a reference entity e, the instruction $\mathbf{create}\ e$ is defined as

$$\begin{array}{l} \mathbf{modifies}\ e, t \\ \mathbf{ensure}\ \underline{e} = \{e\} \wedge \mathit{default}(e) \end{array}$$

where t represents the global clock (it is used in refining specifications to loops or recursive programs). The $\mathit{default}(e)$ clause asserts that each attribute $e.a$ is set to its default value on creation as described in [Mey97] (e.g., if a is a $\mathit{BOOLEAN}$ it is set to false, if it's a reference it is set to Void etc.). If the class $e.\mathit{type}$ has a creation routine r (whose purpose is to establish the class invariant), then we must execute this routine after the creation statement, i.e., $\mathbf{create}\ e;\ e.r$.

The type-compatible reference assignment statement $e1 := e2$ changes entity $e1$ to refer to the same object as entity $e2$. Definition 2 provides the semantics for the reference assignment (i.e., assigning references to references), leaving assignments involving both references and expanded types for the sequel.

Definition 2. *[Reference assignment semantics]* Suppose $e1$ and $e2$ are references and their declared types are compatible according to [Mey97], so that $e2$ can be assigned to $e1$. Then $e1 := e2$ is defined as

$$\begin{aligned} &\textbf{modifies } e1, t \\ &\textbf{ensure } e1 = e2 \end{aligned}$$

We now present the meaning of procedure calls, to illustrate how the theory can be applied to feature calls; query calls are formalised in [PO03]. Definition 3 provides the meaning of a targeted command call $e1.c(e2)$, where $e1$ and $e2$ are reference entities and c is a command of class $SUPPLIER$. The meaning of this call is supplied by the precondition and postcondition of c, targeted to the entities $e1$ and $e2$ (in the definition, $\backslash$ applied to any expression refers to the value of the expression evaluated in the prestate).

Definition 3. *[Targeted command call]* The call $e1.c(e2)$ for command $c(x1 : TYPE)$, which changes entities contained in $c.modifies$, and which is in a class having attribute a, means

$$\begin{aligned} &\textbf{modifies } c.modifies[a := e1.a, x1.a := e2.a] \\ &\textbf{require } e1 \neq Void; \\ &\qquad c.pre[x1 := e2, a := e1.a, Current := e1] \\ &\textbf{ensure } c.post[\backslash a := \backslash e1.a, \backslash Current := \backslash e1, \\ &\qquad\qquad x1 := e2, a := e1.a, Current := e1]; \\ &\qquad e2 = \backslash e2 \end{aligned}$$

Local variables can be introduced to deal with more complicated, and potentially multiple, arguments.

Such statements, e.g., assignment statements and command calls, can be introduced during refinement, and thus it is possible to check that routine implementations are consistent with the pre- and postconditions specified in class interfaces.

2.1 Expanded types

We have not yet discussed the effect of assigning an expanded object to a reference, and vice versa. The interplay between expanded (composite) types and reference types is somewhat complicated by the fact that Eiffel allows user-defined expanded types, and thus the semantics of the assignment statement needs to be generalised in order to allow interactions between the two kinds of variables. The table in Fig. 2 suggests

$e1$ expanded and $e2$ reference $e1 := e2$ equivalent to $e1.copy(e2)$	$e1$ reference and $e2$ expanded $e1 := e2$ equivalent to $e1 := clone(e2)$
modifies $t, e1$ **require** $e2 \neq Void$ **ensure** $equal(e1, e2)$	**modifies** $t, e1$ **require** $true$ **ensure** $\underline{e1} = \{e1\} \wedge equal(e1, e2)$

Fig. 2. Hybrid assignments

how to extend our approach to handle them, leaving a full treatment (e.g., expanded parameters) for later work.

Thus, full formalisation of expanded types will require formalisation of Eiffel's $copy$ and $clone$ statements. This can be carried out using the notion of entity groups described previously.

3 PVS Formulation

The axiomatisation of Eiffel reference types presented in the previous section is sufficient and useful for manual reasoning; examples in [PO03] demonstrate its efficacy. We are currently formalising the axioms and definitions in the PVS specification language, so that we can use particularly the PVS prover to reason about references automatically. In this section, we briefly outline progress that has been made on the formalisation.

The PVS theory includes four main sections:

- declarations of basic types for entities, entity groups, and primitive types;
- declaration of primitive functions for entity comparison, identifying different kinds of entities (i.e., basic types versus reference types), as well as conversions of Eiffel basic types into PVS types;
- the axioms from Section 2;
- Definitions of **create**, reference assignment, feature calls, and hybrid assignments involving expanded and reference types.

To use the theory, programmers `import` it in a theory that defines PVS translations of programmer-defined classes. The PVS translations define new types, functions, and axioms that constrain the programmer classes. This is discussed further in the sequel.

3.1 Declarations of basic types and functions

Fundamental types must be declared in PVS for Eiffel entities (i.e., variables), entity groups, and entity groups associated with routines.

```
ENTITY: TYPE+
ENTITY_GROUP: TYPE+ = set[ENTITY]
SET_GROUP: TYPE+ = set[ENTITY_GROUP]
ROUTINE: TYPE+
QUERY, COMMAND: TYPE+ FROM ROUTINE
```

Each new class introduced in an Eiffel program (not including basic/primitive types) will introduce a new PVS type holding attributes. These PVS types all are subtypes of pre-declared PVS type `OBJECT`. This type is declared primarily for generality: to be able to define functions that apply to all classes. Note that basic/primitive Eiffel types are subtypes of `OBJECT`.

```
OBJECT: TYPE+
EIFFEL_TYPE: TYPE+ FROM OBJECT
EIF_INT, EIF_REAL, EIF_CHAR: TYPE+ FROM EIFFEL_TYPE
```

Useful functions can now be declared. In particular, we will need to know whether an entity is attached to an object, whether two entities have the same declared/static type, whether two entities are reference (or object) equal, etc. Finally, we need to define functions to implement the entity groups from the previous section, i.e., $r.\rho$ and $r.\pi$.

```
isvoid: [ ENTITY -> bool ]
deref: [ ENTITY -> OBJECT ]
sametype: [ ENTITY, ENTITY -> bool ]
ref_equal: [ ENTITY, ENTITY -> bool ]
obj_equal: [ ENTITY, ENTITY -> bool ]
routine_entities: [ ROUTINE -> set[ENTITY] ]
pi: [ ROUTINE -> SET_GROUP ]
ep_from_entity: [ ENTITY -> ENTITY_GROUP ]
pre: [ ROUTINE, ENTITY, ENTITY -> bool ]
post: [ ROUTINE, ENTITY, ENTITY, ENTITY -> bool ]
frame: [ ROUTINE -> bool ]
```

We will also need to provide conversions from Eiffel built-in types, such as integers and characters, to their PVS equivalents. In this sense, we need to implement a PVS embedding of Eiffel primitives. Since PVS provides a conversion facility, this is relatively straightforward.

```
int_c: [ EIF_INT -> int ]
real_c : [ EIF_REAL -> real ]
char_c : [ EIF_CHAR -> char ]

CONVERSION int_c, real_c, char_c
```

3.2 PVS specification of entity group axioms

The axioms from Section 2 can now be expressed in PVS, based on the functions and basic types previously declared. We present several axioms to illustrate the process. First, the axiom that states that reference equality is stronger than object equality. This is a direct transliteration of the ERC axiom.

```
ax2: AXIOM
(FORALL (em, en: ENTITY):
  ref_equal(em,en) IMPLIES obj_equal(em,en))
```

More complex is the axiom that states that entity groups associated with routines are either equal or they do not intersect.

```
ax3: AXIOM
(FORALL (r:ROUTINE):
  (FORALL (ei,ej:ENTITY_GROUP):
    member(ei, pi(r)) AND member(ej, pi(r)) IMPLIES
      ((ei=ej) OR (empty?(intersection(ei,ej))) ) ))
```

Two final examples demonstrate PVS specifications of: the axiom that states that entities in the same entity group refer to the same entity; and, an entity in an entity group must be non-$Void$.

```
ax5: AXIOM
(FORALL (r:ROUTINE): (FORALL (ei:ENTITY_GROUP):
  (FORALL (em,en: ENTITY):
  (member(ei,pi(r)) AND member(em,ei) AND member(en,ei)) IMPLIES
  ep_from_entity(em)=ep_from_entity(en) )))

ax6: AXIOM
(FORALL (r:ROUTINE): (FORALL (em:ENTITY):
  (member(ei,pi(r)) AND member(em,ei)) IMPLIES NOT isvoid(em) )))
```

Based on these axioms, and the functions declared in the previous subsection, reference and object equality can be defined as well.

3.3 PVS definitions of operations

We can now specify some of the fundamental Eiffel instructions in PVS, particularly those that manipulate reference types. As two examples, we show how to specify Eiffel's **create** statement, and reference assignment.

Eiffel's creation and initialisation statement has the form **create** e, where e is an entity that may or may not be attached to an object. It is formulated in PVS as follows.

```
create: [ ENTITY -> bool ] =
(LAMBDA (em: ENTITY): ep_from_entity(em)= singleton(em) AND default(em))
```

`default` is a function that, given an entity, returns $true$ iff the entity has been initialised to its default value. The default value depends on the type of the object attached to the entity (but typically it is made up of default values for the attributes of the object - e.g., 0 for integers, $true$ for booleans, $Void$ for references).

Reference assignment of the form $e := e1$ is formalised as follows. `ref_assign` returns $true$ iff the two argument entities are reference equal. In other words, $e := e1$ is represented in PVS as the function call $ref\_equal(em, en)$ - we are representing the *effect* of the assignment in PVS.

```
ref_assign: [ENTITY, ENTITY -> bool ] = (LAMBDA (em,en:ENTITY): ref_equal(em,en))
```

3.4 Feature calls

The fundamental construct in any object-oriented program is the targetted feature call, which has the form $o.f(a)$, where o is an entity/variable – the target – that is attached to an object, f is a function or procedure, and a is a set of arguments. Targetted procedure calls are statements, and can thus be sequentially composed with other typically programming constructs, such as assignments, loops, and selections. Targetted function calls return values, and as such can be used as r-values in assignment statements, in guards, as arguments, or in pre- and postconditions. We show how to formalise Eiffel targetted function and procedure calls in PVS, starting with the function call $e2.q(e3)$. This call might appear in an assertion, and as such it is important to formalise it separately from any statement in which it can be used. Its PVS formulation is as follows. To start with, we declare a PVS function `eval` which returns an entity when applied to a function, target, and set of arguments; the return type should be constrained to be that of the Eiffel function. This PVS function is then *implicitly* defined in the axiom `eval_ax`, which defines what it means to call q with target $e2$ and argument $e3$. The axiom assumes that functions to obtain the precondition and postcondition of a routine are available, respectively, as `pre` and `post`.

```
eval:[ ROUTINE, ENTITY, ENTITY -> ENTITY ]

eval_ax: AXIOM
(FORALL (em,en:ENTITY): FORALL (r:ROUTINE):
  (pre(r,em,en) AND NOT isvoid(em) IMPLIES post(r,em,en,eval(em,en,r))))
```

To use the above axiom, one makes use of the function `eval`. For example, to define the meaning of the assigned query call $e1 := e2.q(e3)$, one would make use of the following PVS function.

```
acq:[QUERY, ENTITY, ENTITY, ENTITY -> bool ] =
(LAMBDA (q:QUERY,e1:ENTITY,e2:ENTITY,e3:ENTITY):
  (NOT isvoid(e2) and reference(e1) AND
   reference(e2) and compatible(e1,result_type(q))) IMPLIES e1 = eval(e2,e3,q))
```

Finally, we formalise the targetted procedure call $e1.c(e2)$ in PVS. We first assume that the frame of each procedure can and has been specified by the programmer, and is available via the function `frame`. The call can then be formalised as follows.

```
tcq:[ COMMAND, ENTITY, ENTITY, ENTITY, ENTITY -> bool ] =
(LAMBDA (c:COMMAND,e1:ENTITY, e2:ENTITY,
         old_e1:ENTITY, old_e2:ENTITY):
  (NOT isvoid(e1) AND pre(c,old_e1,e2)) IMPLIES
  (post(c,e1,e2,old_e1) AND e2=old_e2 AND frame(c))))
```

Notice that the call introduces both old and new variables, i.e., pre- and poststate.

3.5 Expanded types

Based on the axioms and functions defined previously, it is now possible to formalise hybrid assignment statements, involving both expanded and reference types. For example, consider the assignment statement $e1 := e2$, where $e1$ is an expanded type and $e2$ is a reference type. In Eiffel, the meaning of this statement is identical to the instruction $e1.copy(e2)$. In other words, an attribute-by-attribute copy is made of the object attached to $e2$, stored in $e1$. We can thus formalise this instruction by partly formalising the Eiffel feature $copy$, as follows[1].

```
copy: [ ENTITY, ENTITY -> bool ] =
(LAMBDA (e1, e2: ENTITY):
  (reference(e2) AND NOT reference(e1) AND
   NOT isvoid(e2)) IMPLIES (obj_equal(e1,deref(e2)))
```

3.6 Using the theory

The above suite of functions, declarations, definitions, and axioms is part of the PVS theory of Eiffel reference types. It is intended to be used in PVS translations of Eiffel specifications[2]. Thus, an Eiffel program will be translated into a set of PVS theories – one per class – where each theory imports the above PVS theory of reference types. Each Eiffel class is translated into a PVS record, declaring the class's attributes and types. For example, a class C with attributes $x : INTEGER$ and $c : C$ (i.e., one expanded attribute and one reference) will be mapped to the PVS type

```
C: TYPE+ FROM OBJECT =
  [# x: EIF_INT, c: ENTITY #]

C_ax: AXIOM
(FORALL (varc:C): EXISTS (oc:C):
  NOT isvoid(c(varc)) IMPLIES deref(c(varc)) = oc )
```

The axiom states that whenever c is attached to an object, it must be attached to an object of type C.

Translations of routines (as PVS functions) and their contracts can be carried out in much the same way, though details remain to be worked out. We have concentrated so far on fixing the details of the theory of reference types in PVS, since the formalisation of classes and contracts depends on it.

A challenge with using the theory will be in proving that sequential compositions of routine calls that appear in method bodies satisfy a contract. This is challenging because intermediate state needs to be introduced. However, the formalisation of sequential composition in [Heh03] deals with this problem by treating sequential composition as relational composition, thus hiding the intermediate state using an existential quantifier. We expect that we can use this approach in PVS.

[1] *copy* is not fully formalised on its domain; it is defined only on domains where the l-value is expanded and the r-value reference.

[2] An Eiffel specification includes contracts, but not implementations, of classes – i.e., no routine bodies.

4 Discussion and Conclusions

The PVS theory defined above is accepted by the PVS system, and typechecks. Work is continuing on using the theory to carry out examples of reasoning, particularly for refinement. Current work is focusing on completing the aliasing example in [PO03] in full detail in PVS. So far, use of PVS has helped us detect omissions – particularly in terms of the types used in expressions – in our semi-formal axiomatisation of entity groups: PVS forces us to be explicit about types, and this helped to reveal errors.

Additional work is considering alternative PVS formalisations of the refinement calculus (particularly, using an explicit model of Eiffel memory, with entity groups being a derived notion). We are also automating the generation of PVS theories from Eiffel. A basic PVS theory, capturing entity groups, will be imported by any generated theory; the generated theory will define records to capture the attributes and pre- and postconditions associated with routines of classes.

References

[BJ01] J. van den Berg and B. Jacobs.: The LOOP compiler for Java and JML. In *Proc. TACAS 2001*, Lecture Notes in Computer Science 2031, Springer-Verlag, 2001.

[CN00] A. Cavalcanti and D. Naumann.: A weakest-precondition semantics for refinement object-oriented programs. *IEEE Trans. Software Engineering* **26**(8), 2000.

[CO95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas.: A Tutorial Introduction to PVS, in *Proc. WIFT '95,* Springer-Verlag, 1995.

[DL98] D.L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe.: Extended Static Checking. SRC Research Report 159, December 1998.

[DL01] K. Dhara and G. Leavens.: Mutation, Aliasing, Viewpoints, Modular Reasoning, and Weak Behavioral Subtyping. Technical Report #01-02, Department of Computer Science, Iowa State University, March 2001.

[Es00] Escher Technologies, Inc.: Getting Started With Perfect, available from `www.eschertech.com`, 2000.

[Heh03] E.C.R. Hehner.: *A Practical Theory of Programming (Second Edition)*, Springer-Verlag, 2003.

[LN00] K.R.M. Leino, G. Nelson, and J.B. Saxe.: ESC/Java User's Manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[Mey92] B. Meyer.: *Eiffel: the Language*, Prentice-Hall, 1992.

[Mey97] B. Meyer.: *Object-Oriented Software Construction* (Second Edition), Prentice-Hall, 1997.

[Mey00] B. Meyer.: Agents, iterators, and introspection. ISE Inc. Technical Paper, 2000.

[Mey03] B. Meyer.: Proving program pointer properties, draft last revised February 2003.

[PO03] R. Paige and J. Ostroff.: *ERC: an Object-Oriented Refinement Calculus for Eiffel*, under review, 2003. Draft available at www.cs.yorku.ca/techreports/2001.

Connecting Effects and Uniqueness with Adoption

John Tang Boyland

University of Wisconsin—Milwaukee

Abstract. In a previous paper, we discussed how the concepts of uniqueness and effects are interdependent. In this paper, we show how "Adoption and Focus," a proposal for handling linear pointers in shared variables can be extended to connect the two concepts. Our innovations include the ability to define adoption relations between individual fields rather than whole objects, and the ability to "focus" on more than one adoptee at a time. The resulting system uses recursive alias types, "permission closures" and "conditional permissions." Then we show how previously proposed effect and uniqueness annotations can be represented in the type system.

1 Introduction

In a previous paper [1], we discussed how the concepts of uniqueness and effects are interdependent. If one wishes to check uniqueness, it can be best done when considering effects, because read effects on a unique variable cannot be permitted while it is temporarily aliased. Checking effects on the other hand may require uniqueness, because an effect on a unique object can be transferred to the object that currently has the only unique reference. In retrospect, this interdependence could be expected because the better-known problems of data-dependence determination and aliasing are similarly related. Uniqueness involves an aliasing property and effects can be used to determine data dependencies.

1.1 Background on Uniqueness

Unique references that may occur in shared structures can be modeled soundly using "destructive reads" in which the stored pointer variable (often a field of an object) is nullified atomically with the read of that variable. This solution goes back at least to Hogg's Islands [2] and has been variously used by Baker [3], Minsky's Eiffel* [4], Aldrich et al's AliasJava [5] (in their proofs), and Clarke and Wrigstad's External Uniqueness [6].

However, destructive reads have several problems: (1) they make it difficult to query information about a unique object without losing it; (2) they make it impossible to have sound invariants about the non-nullness of unique variables; (3) they are inappropriate for use in "const" methods, which are supposed to treat their object as read only; (4) they require a language change. Thus several

researchers have independently proposed the use of what we call "borrowing" reads, which do not nullify the field. Unfortunately borrowing reads greatly impact the benefits of uniqueness unless it can be shown that a unique field is not read (or at least not considered unique) during the lifetime of the borrowing. For example, AliasJava permits borrowing without checking for possible reads during the lifetime of the borrow and thus can only guarantee that a "unique" pointer stored in a field will not be also available in another field. In particular, a class cannot prevent "outsiders" from modifying the contents of supposedly unique sub-objects of instances of this class.

Similarly, Leino, Nelson and Stata [7, 8] proposed a pointer property, "virginity," related to uniqueness that is required of all initialization of "pivot" fields (fields that refer to wholly-owned subsidiary objects or "sub-objects"). The rules have a similar weakness to that of AliasJava: they are (intentionally) not strong enough to prevent a client of an object from having access to its sub-objects.

Leino and others' recent work [9] and "strong" ownership type systems [10] avoid these aliasing problems by not permitting clients to create objects that will be internal to a container. In essence, the system forbids objects from being transferred from one container to another. External uniqueness extends ownership to cover transferable uniqueness, but does so through destructive reads.

In our earlier work on "alias burying" [11], we proposed using effects annotations to prevent borrowing reads from weakening the semantics of uniqueness. With alias burying, no destructive reads are needed. The paper suggested annotating every method with the list of fields of this or any other object that is read during the dynamic extent of the method call. Clearly, this not only exposes too much of object's internal structure, but is overly conservative since it does not distinguish different objects. We intended to use our object-oriented effects system [12] instead, but no one has been able to come up with a satisfactory system to combine these two areas. This paper contributes such a type system.

1.2 Contribution

We can check that program accesses meet declared effects by using a *permission* system, in which the context used to check program elements indicates which parts of the state we are allowed to access. ("Fractional permissions" [13] can be used to distinguish reads from writes, but this paper will not discuss this (orthogonal) extension.) Effects are in terms of *state* in the program: variables, and fields of objects on the heap. For encapsulation purposes, we aggregate fields into "data groups" [12, 14]. This is modeled by having the permission to access the field "nested" within the permission to access the data group.

In a permission system, there is only one permission for each state. Thus permissions can also be used to model uniqueness: a unique pointer is one which is packaged along with the permission to access the state pointed to. This conception of uniqueness is weaker than the usual sense of uniqueness in which there are no other pointers to the state in the store. However, in our system, any other potential pointers to the same state come without permission to access it, and thus even the limited sense of uniqueness provided by permissions is sufficient

for analysis purposes. The problem is not the *existence* of aliasing pointers, but rather the *access* of the state through these aliasing pointers.

Of course not all pointers are unique. A *shared* pointer is modeled by having the permission to access the pointed-to state "nested" in a globally accessible permission. Now, we want to make sure that it is impossible for two separate parts of the code to grab the permissions to the same shared state. This could be done using dynamic checks, but in our system we use the type system to prevent the same permission from being removed twice without being replaced in between.

Languages often permit a "null" pointer to be used in the place of a pointer to actual state. Our system makes this detail explicit: the permissions associated with a possibly-null pointer are conditional on the boolean formula that the pointer is not null.

Thus the type system described in this paper has the following properties:

- It uses permissions to model both effects and uniqueness.
- It uses the concept of permission "nesting" (adoption) to model shared state and aggregation of state in "data groups."
- Conditional permissions make explicit the use of "null" or other terminating pointers.

In the following section, we describe this type system, and then in Section 3 we showed how one can take some of our earlier effects annotations together with the uniqueness annotations of our work on Alias Burying and interpret them as types in system proposed in this paper. Section 4 compares our system to closely related work.

2 The System

We define our type system over a simple low-level imperative language. Section 3 then adapts the types to a Java-like language.

2.1 Operational Semantics

The source language has four kinds of values: the unit value, booleans, integers and pointer values (that is, object references):

$$\text{(value)} \quad v ::= \texttt{()} \mid \texttt{true} \mid \texttt{false} \mid n \mid o$$

Here o refers to an object reference, an absolute memory address. The only absolute memory address that occurs in unevaluated programs is \$0—the value of null pointers.

In this simple language, the only kind of variable is the field of an object (global variables are handled by treating them as fields of the null object). Array elements could be handled with dependent types [15] in a similar manner.

Each field f has a declared default value and type chosen from the following limited set:

v_f	τ_f
`()`	unit
`false`	bool
0	int
\$0	ptr(\$0)

Expressions include values, simple arithmetic (represented by addition), comparisons, allocation, field reads and writes, sequential composition, conditionals, procedure calls and "nesting" (adoption):

$$s ::= \begin{array}{l} v \mid e\texttt{+}e \mid e\texttt{=}e \mid \texttt{new\{}f\texttt{,...,}f\texttt{\}} \mid e\texttt{.}f \mid \\ e\texttt{.}f\texttt{:=}e \mid s\texttt{;}s \mid \texttt{if } b \texttt{ then } s \texttt{ else } s \mid \texttt{call } p \mid \texttt{nest } e\texttt{.}f \texttt{ in } e\texttt{.}f \end{array}$$

A program consists of a mapping of procedure names to procedure expressions. A memory consists of a binding of fields to their values (a function). We also record "adoption" (the nesting relation).

$$\begin{array}{rl} \text{(program)} & g ::= \{p \to e, \ldots\} \\ \text{(location)} & loc ::= o\texttt{.}f \\ \text{(memory)} & \mu ::= \{loc \to v, \ldots\} \\ \text{(adoption)} & a ::= \{loc \prec loc, \ldots\} \end{array}$$

There is no requirement that adoption is a functional relation.

Figure 1 defines a small-step semantics for evaluating this simple language. The two interesting parts are (1) the evaluation of `new` expressions in which an address is found unused in the memory or adoption information, and (2) nesting which adds an adoption relation fact. Adoption can never be undone.

2.2 Permission Types

The environment $E = (\Delta; \Pi)$ in which code is checked has two parts: a type context Δ that lists address variables ρ; and a "set" of permissions Π that indicates what state we are permitted to access and the type that the state will have. We treat Π in a somewhat linear fashion, in that permissions cannot be duplicated. However, making use of permissions does not consume them. (The use of "..." here means there are other forms for permissions.)

$$\begin{array}{rl} \text{(type context)} & \Delta ::= \cdot \mid \rho \mid \Delta, \Delta \\ \text{(permissions)} & \Pi ::= \cdot \mid \Pi, \Pi \mid \ldots \end{array}$$

The most important kind of permission is a key k. Now, since any key may have other keys nested in it, and some of those keys may be active, the permission enumerates those nested keys that are currently "carved out." As mentioned in the introduction, unlike earlier proposals is which a key applies to a whole

$$\frac{(\mu; a; e_1) \to_g (\mu'; a'; e_1')}{(\mu; a; e_1 \texttt{+} e_2) \to_g (\mu'; a'; e_1' \texttt{+} e_2)} \qquad \frac{(\mu; a; e_2) \to_g (\mu'; a'; e_2')}{(\mu; a; v \texttt{+} e_2) \to_g (\mu'; a'; v \texttt{+} e_2')}$$

$$(\mu; a; n_1 \texttt{+} n_2) \to_g (\mu; a; n_1 + n_2) \qquad \frac{(\mu; a; e_1) \to_g (\mu'; a'; e_1')}{(\mu; a; e_1 \texttt{=} e_2) \to_g (\mu'; a'; e_1' \texttt{=} e_2)}$$

$$\frac{(\mu; a; e_2) \to_g (\mu'; a'; e_2')}{(\mu; a; v \texttt{=} e_2) \to_g (\mu'; a'; v \texttt{=} e_2')} \qquad (\mu; a; v_1 \texttt{=} v_2) \to_g (\mu; a; v_1 = v_2)$$

$$\frac{o \notin \mathrm{Rng}(\mu) \qquad \forall_{f \in F}\, o.f \notin \mathrm{Dom}(\mu) \cup \mathrm{Dom}(a) \cup \mathrm{Rng}(a)}{(\mu; a; \texttt{new\{}f_1\texttt{,} \ldots \texttt{,}f_n\texttt{\}}) \to_g (\mu[o.f_i \to v_{f_i} \mid 1 \le i \le n]; a; o)}$$

$$\frac{(\mu; a; e) \to_g (\mu'; a'; e')}{(\mu; a; e\,\texttt{.}\,f) \to_g (\mu'; a'; e'\,\texttt{.}\,f)} \qquad \frac{\mu(o.f) = v}{(\mu; a; o\,\texttt{.}\,f) \to_g (\mu; a; v)}$$

$$\frac{(\mu; a; e_1) \to_g (\mu'; a'; e_1')}{(\mu; a; e_1\,\texttt{.}\,f\,\texttt{:=}\,e_2) \to_g (\mu'; a'; e_1'\,\texttt{.}\,f\,\texttt{:=}\,e_2)} \qquad \frac{(\mu; a; e_2) \to_g (\mu'; a'; e_2')}{(\mu; a; v_1\,\texttt{.}\,f\,\texttt{:=}\,e_2) \to_g (\mu'; a'; v_1\,\texttt{.}\,f\,\texttt{:=}\,e_2')}$$

$$\frac{v_2 \sim v_f \qquad \mu' = \mu[o.f \to v_2]}{(\mu; a; o\,\texttt{.}\,f\,\texttt{:=}\,v_2) \to_g (\mu'; a; ())} \qquad \frac{(\mu; a; e_1) \to_g (\mu'; a'; e_1')}{(\mu; a; e_1\,\texttt{;}\,e_2) \to_g (\mu'; a'; e_1'\,\texttt{;}\,e_2)}$$

$$(\mu; a; \texttt{();}\,e_2) \to_g (\mu; a; e_2)$$

$$\frac{(\mu; a; e_1) \to_g (\mu'; a'; e_1')}{(\mu; a; \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \to_g (\mu'; a'; \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3)}$$

$$(\mu; a; \texttt{if true then}\ e_2\ \texttt{else}\ e_3) \to_g (\mu; a; e_2)$$

$$(\mu; a; \texttt{if false then}\ e_2\ \texttt{else}\ e_3) \to_g (\mu; a; e_3) \qquad (\mu; a; \texttt{call}\ p) \to_g (\mu; a; g\,p)$$

$$\frac{(\mu; a; e_1) \to_g (\mu'; a'; e_1')}{(\mu; a; \texttt{nest}\ e_1\texttt{.}f_1\ \texttt{in}\ e_2\texttt{.}f_2) \to_g (\mu'; a'; \texttt{nest}\ e_1'\texttt{.}f_1\ \texttt{in}\ e_2\texttt{.}f_2)}$$

$$\frac{(\mu; a; e_2) \to_g (\mu'; a'; e_2')}{(\mu; a; \texttt{nest}\ v_1\texttt{.}f_1\ \texttt{in}\ e_2\texttt{.}f_2) \to_g (\mu'; a'; \texttt{nest}\ v_1\texttt{.}f_1\ \texttt{in}\ e_2'\texttt{.}f_2)}$$

$$\frac{a' = a \cup \{(o_1.f_1) \prec (o_2.f_2)\}}{(\mu; a; \texttt{nest}\ o_1\texttt{.}f_1\ \texttt{in}\ o_2\texttt{.}f_2) \to_g (\mu; a'; ())}$$

Fig. 1. Operational Semantics

object, a key here applies to a *field* of an object. The permission to access a field includes the type stored there. We have four atomic types:

$$\begin{aligned} \text{(type)} \quad & \tau ::= \text{unit} \mid \text{bool} \mid \text{int} \mid \text{ptr}(l) \mid \dots \\ \text{(object reference)} \quad & l ::= o \mid \rho \\ \text{(key)} \quad & k ::= l.f \\ \text{(permissions)} \quad & \Pi ::= \dots \mid k : \tau \backslash \{k_1 : \tau_1, \dots, k_n : \tau_n\} \mid \dots \end{aligned}$$

If there are no keys carved out of a key (the permission is of the form $k : \tau \backslash \{\}$, the permission may be abbreviated as $k : \tau$.

Two types τ and τ' are "storage compatible" (written $\tau \sim \tau'$) if space storing a value of type τ can be reused to store a τ'. Of course $\tau \sim \tau$ for all types τ, but we also assume all pointer types are interchangeable: $\text{ptr}(l) \sim \text{ptr}(l')$.

Field of unit type do not store any information, but instead are used to model "data groups" (we reuse the terminology of Leino and others [14, 9]). Aggregation of fields into data groups is accomplished by nesting, for instance $\rho.\text{x} : \text{int} \prec \rho.\text{loc}$, where the "x" field of the object is part of the "loc" data group.

The type system described here uses a simple logic that can be represented by boolean formulae over key equalities $k = k'$ and nesting information $k : \tau \prec k'$. In a slight abuse of notation, we conflate syntactic boolean operators (such as $\wedge, \vee, \neg$) with the underlying semantic boolean operations. The syntax also provides a construct for named recursive formulae t over object references.

$$\text{(formula)} \quad \Gamma ::= \text{true} \mid k = k \mid k : \tau \prec k \mid \Gamma \wedge \Gamma \mid \neg\Gamma \mid t(l_1, \dots, l_n)$$

(We write $k \neq k'$ as shorthand for $\neg(k = k')$.)

Boolean formulae are used in two ways in permissions. First, a formula may be paired with a permissions "set." Second, we have conditional permissions in which a set of permissions is guarded by a condition. In another slight abuse of notation, we borrow the linear implication " $\multimap$ " operator to remind the reader that conditional permissions are consumed when applied:

$$\text{(permissions)} \quad \Pi ::= \dots \mid (\Pi; \Gamma) \mid \Gamma \multimap \Pi$$

For example, suppose `x` is a global variable (represented by a field of the null object) that may be null, but if it is not null, we have permission to access all of the fields of the pointed-to object. This situation is represented by the following set of permissions:

$$\$0.\texttt{x} : \text{ptr}(\rho), \rho \neq \$0 \multimap \rho.\text{All}$$

Here "All" is a data group in which all of the fields of the object are nested, perhaps indirectly. Now, we often do not know what the actual pointer value of a variable is and thus need to use some form of quantified types. For this reason a variable may have existential type, for example $\texttt{x} : \exists\rho.\text{ptr}(\rho)$, but this form is not sufficient, because we need to be able to express permissions about the existentially bound address variable. Thus we have forms of compound types, a pairing of a set of permissions with a type, and an existential:

$$\text{(type)} \quad \tau ::= \dots \mid \tau \text{ with } \Pi \mid \exists\Delta.\tau$$

Thus, a variable with an unknown pointer value, but for whose object (if not null) we have permission to access the fields, would be declared as follows:

$$\$0.\mathtt{x} : \exists\rho.\mathrm{ptr}(\rho) \text{ with } \rho \neq \$0 \multimap \rho.\mathrm{All} : \mathrm{unit}$$

This indeed is how we express the concept of a unique pointer. While other places in the system may have the same pointer value, no one else has the permission to access the pointed-to state, since permissions are unique.

A procedure may be polymorphic over some location variables Δ. It accepts a "set" of permissions and returns a (possibly new) "set" of permissions, using perhaps some new variables. A program type is a type for each procedure. Substitutions map address variables to other variables or to absolute addresses.

$$\begin{aligned} \text{(procedure type)} \quad & \alpha ::= \forall\Delta.\,(\Pi \to \exists\Delta.\Pi) \\ \text{(program type)} \quad & \omega ::= \{p : \alpha, \ldots\} \\ \text{(substitution)} \quad & \sigma ::= \{\rho \to l, \ldots\} \end{aligned}$$

Parameters and results can be passed in global variables or in specially created activation objects. Substitutions are used to transform the permissions into a form accepted by a procedure and to convert the resulting permissions back. Substitutions are typed by their domain and range:

$$\frac{\mathrm{Dom}(\sigma) = \Delta \qquad \mathrm{Rng}(\sigma) \subseteq \Delta' \cup O}{\sigma : \Delta \to \Delta'}$$

Here O is the set of all absolute (object) addresses. Substitutions are lifted to apply to permissions in the normal way, and in particular σ acts as the identity function on address variables outside of its explicit domain.

The type rules corresponding to the syntax are given in Figure 2. Each expression produces a possibly new environment after being checked and thus our relation is $E \vdash_\omega e : \tau \dashv E'$ where ω is the program typing.

The If rules bear some explanation: as well as a default rule, we have a special case rule for the comparing of pointers. The type system makes the equality or inequality being tested available in the corresponding branches. At the end of an `if`, we need to merge the two environments, which makes use of substitutions:

$$\frac{\begin{array}{c}\Delta_1, \Pi_1 = \sigma_1\Delta'; \sigma_1\Pi' \quad \Delta_2, \Pi_2 = \sigma_2\Delta'; \sigma_2\Pi' \\ \Delta = \Delta_1 \cap \Delta_2 \qquad \Delta' - \Delta \text{ fresh} \qquad \rho \in \Delta \Rightarrow \sigma_i\rho = \rho\end{array}}{(\Delta'; \Pi') = (\Delta_1, \Pi_1) \vee (\Delta_2, \Pi_2)}$$

The IfTrue and IfFalse rules are needed for type preservation.

The rule for `nest` expressions requires that the key being "adopted" be fully available (without anything carved out) and not equal to any key currently carved out of the "adopter." This rule prevents a key from being twice adopted by the same adopter, and also prevents self or cyclic adoption. These situations do not cause consistency problems, but are probably bugs in the code.

$$\textsc{Unit}\quad E \vdash_\omega () : \text{unit} \dashv E \qquad \textsc{Num}\quad E \vdash_\omega n : \text{int} \dashv E \qquad \textsc{True}\quad E \vdash_\omega \texttt{true} : \text{bool} \dashv E$$

$$\textsc{False}\quad E \vdash_\omega \texttt{false} : \text{bool} \dashv E \qquad \textsc{Address}\quad E \vdash_\omega o : \text{ptr}(o) \dashv E$$

$$\textsc{Plus}\ \frac{E \vdash_\omega e_1 : \text{int} \dashv E' \vdash_\omega e_2 : \text{int} \dashv E''}{E \vdash_\omega e_1 \texttt{+} e_2 : \text{int} \dashv E''} \qquad \textsc{Equal}\ \frac{E \vdash_\omega e_1 : \tau \dashv E' \vdash_\omega e_2 : \tau \dashv' E'' \qquad \tau \sim \tau'}{E \vdash_\omega e_1 \texttt{=} e_2 : \text{bool} \dashv E''}$$

$$\textsc{RField}\ \frac{E \vdash_\omega e : \text{ptr}(l) \dashv \Delta'; \Pi' \qquad \Pi' = l.f : \tau \backslash \{\ldots\}, \Pi_1}{E \vdash_\omega e\,.\,f : \tau \dashv \Delta'; \Pi'}$$

$$\textsc{New}\ \frac{\rho \text{ fresh}}{\Delta; \Pi \vdash_\omega \texttt{new\{}f_i \mid 1 \le i \le n\texttt{\}} : \text{ptr}(\rho) \dashv \rho, \Delta; \rho.f_i : \tau_{f_i} \mid 1 \le i \le n, \Pi_1}$$

$$\textsc{WField}\ \frac{E \vdash_\omega e_1 : \text{ptr}(l) \dashv E' \vdash_\omega e_2 : \tau \dashv \Delta''; \Pi'' \qquad \Pi'' = l.f : \tau' \backslash \{\ldots\}, \Pi_1 \qquad \tau \sim \tau'}{E \vdash_\omega e_1\,.\,f \texttt{:=} e_2 : \text{unit} \dashv \Delta''; l.f : \tau \backslash \{\ldots\}, \Pi_1}$$

$$\textsc{Seq}\ \frac{E \vdash_\omega s : \text{unit} \dashv E' \vdash_\omega s' : \tau \dashv E''}{E \vdash_\omega s\texttt{;}s' : \tau \dashv E''} \qquad \textsc{If}\ \frac{\Delta, \Pi \vdash_\omega e_0 : \text{bool} \dashv E' \quad E' \vdash_\omega e_1 : \text{unit} \dashv E_1 \quad E' \vdash_\omega e_2 : \text{unit} \dashv E_2 \quad E'' = E_1 \vee E_2}{E \vdash_\omega \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \text{unit} \dashv E''}$$

$$\textsc{IfEqual}\ \frac{\begin{array}{c}\Delta; \Pi \vdash_\omega e : \text{ptr}(l) \dashv E' \\ E' \vdash_\omega e' : \text{ptr}(l') \dashv \Delta; \Pi \qquad \Delta; (\Pi; l.\text{All} = l'.\text{All}) \vdash_\omega e_1 : \text{unit} \dashv E_1 \\ \Delta; (\Pi; l.\text{All} \neq l'.\text{All}) \vdash_\omega e_2 : \text{unit} \dashv E_2 \qquad E'' = E_1 \vee E_2\end{array}}{E \vdash_\omega \texttt{if}\ e\texttt{=}e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \text{unit} \dashv E''}$$

$$\textsc{IfTrue}\ \frac{E \vdash_\omega e_1 : \text{unit} \dashv E'}{E \vdash_\omega \texttt{if true then}\, e_1\, \texttt{else}\, e_2 : \text{unit} \dashv E'} \qquad \textsc{IfFalse}\ \frac{E \vdash_\omega e_2 : \text{unit} \dashv E'}{E \vdash_\omega \texttt{if false then}\, e_1\, \texttt{else}\, e_2 : \text{unit} \dashv E'}$$

$$\textsc{Call}\ \frac{\omega(p) = \forall \Delta_1.\Pi_1 \to \exists \Delta_2.\sigma_2\Pi_2 \qquad \sigma_1 : \Delta_1 \to \Delta \qquad \Delta' \text{ fresh} \qquad \sigma_2 : \Delta' \to \Delta_2}{\Delta; \sigma_1\Pi_1, \Pi_3 \vdash_\omega \texttt{call}\ p : \text{unit} \dashv \Delta \cup \Delta'; \sigma_1\Pi_2, \Pi_3}$$

$$\textsc{Nest}\ \frac{\begin{array}{c}E \vdash_\omega e : \text{ptr}(l) \dashv E' \vdash_\omega e' : \text{ptr}(l') \dashv \Delta''; \Pi'' \qquad k = l.f \quad k' = l'.f' \\ \Pi'' = (k : \tau \backslash \{\}; k \neq k_1 \wedge \ldots \wedge k \neq k_n), k' : \tau' \backslash \{k_1 : \tau_1, \ldots, k_n : \tau_n\}, \Pi_1\end{array}}{E \vdash_\omega \texttt{nest}\ e.f\ \texttt{in}\ e'.f' : \text{unit} \dashv \Delta''; (k' : \tau' \backslash \{k_1 : \tau_1, \ldots, k_n : \tau_n\}; k : \tau \prec k'), \Pi_1}$$

$$\textsc{Proc}\ \frac{\Delta_1; \Pi_1 \vdash_\omega s : \text{unit} \dashv \Delta_1'; \sigma\,\Pi_2 \qquad \Delta_1' \cap \Delta_2 = \emptyset \qquad \sigma : \Delta_2 \to \Delta_1'}{\vdash_\omega s : \forall \Delta_1.\Pi_1 \to \exists \Delta_2.\Pi_2}$$

Fig. 2. Syntactic Type rules

2.3 Consistency

Types can ensure that programs don't go wrong, but only if the memory is in a state matched by the types, or permissions in our case. Part of the "matching" that must be done is to give the absolute addresses corresponding to each address variable ρ:

$$\text{(mapping)} \quad \psi ::= \{\rho \to o, \ldots\}$$

The consistency relation is written

$$\mu; a; \psi \vdash \Pi \text{ consistent}$$

We have not worked out all the details of consistency at the point of writing, and space constraints would preclude a full description in any case. The basic concepts that need to be verified are that there never be more than one permission for each key (field), and that the value stored in a field be of the type indicated in the permission. The boolean formulae paired with the permissions must also be matched against the actual memory and adoption facts.

Part of the information in any permission are the equality, inequality and adoption facts in it. For instance, if we see that one key has two other keys carved out of it ($k : \tau \backslash \{k_1 : \tau_1, k_2 : \tau_2\}$), then those keys must be distinct and furthermore must each be nested in k. We write $\Pi \models \Gamma$ to mean that the boolean formula Γ is implied by the set of permissions Π. The condition that lets us determine the most information from a set of permissions is to find anything that is consistent with every situation in which the permissions are consistent:

$$\frac{\forall_{\mu;a;\psi}\ (\mu; a; \psi \vdash \Pi \text{ consistent}) \Rightarrow (\mu; a; \psi \vdash (\cdot; \Gamma) \text{ consistent})}{\Pi \models \Gamma}$$

This definition is not constructive, but it is easy to define special cases such as

$$(\cdot; \Gamma) \models \Gamma \qquad\qquad (f \neq f') \Rightarrow (\cdot \models l.f \neq l'.f')$$

$$k : \tau \backslash \{k_1 : \tau_1, \ldots, k_2 : \tau_2, \ldots\} \models k_1 \neq k_2 \qquad (\Pi_1 \models \Gamma) \Rightarrow ((\Pi_1, \Pi_2) \models \Gamma)$$

$$(\Pi \models \Gamma_1) \wedge (\Pi \models \Gamma_2) \Rightarrow (\Pi \models \Gamma_1 \wedge \Gamma_2)$$

Depending on how consistency is finally formalized, it may be possible to give a fixed number of such structural rules that are complete for the characterization of the $\models$ relation.

We overload the operator to apply to pairs of permissions too:

$$\frac{\forall_{\mu;a;\psi}\ (\mu; a; \psi \vdash \Pi_1 \text{ consistent}) \Rightarrow (\mu; a; \psi \vdash \Pi_2 \text{ consistent})}{\Pi_1 \models \Pi_2}$$

and then define

$$\frac{\Pi_1 \models \Pi_2 \qquad \Pi_2 \models \Pi_1}{\Pi_1 \equiv \Pi_2}$$

As before one can define any number of special cases:

$$\cdot, \Pi \equiv \Pi \qquad\qquad \Pi_1, (\Pi_2, \Pi_3) \equiv (\Pi_1, \Pi_2), \Pi_3$$

$$(\Pi_1; \Gamma_1), (\Pi_2; \Gamma_2) \equiv (\Pi_1, \Pi_2; \Gamma \wedge \Gamma_2)$$

$$k : \tau \backslash \{k_1 : \tau_1, k_2 : \tau_2, \ldots k_n : \tau_n\} \equiv k : \tau \backslash \{k_2 : \tau_2, \ldots, k_n : \tau_n, k_1 : \tau_1\}$$

$$(\Pi \models \Gamma) \Rightarrow (\Pi \equiv (\Pi; \Gamma)) \qquad\qquad (\Gamma \multimap \Pi; \Gamma) \equiv (\Pi; \Gamma)$$

$$k' : \tau' \backslash \{k_1 : \tau_1, \ldots k_n : \tau_n\}; k : \tau \prec k' \wedge k \neq k_1 \wedge \ldots \wedge k \neq k_n \equiv k' : \tau \backslash \{k : \tau, k_1 : \tau_1, \ldots k_n : \tau_n\}, k : \tau$$

The last sample rule shows how one can carve out a key from another key, or replace it.

One may substitute a smaller set of permissions before or after any typing:

$$\textsc{Transform}\ \frac{\Pi \models \Pi_1 \qquad \Delta; \Pi_1 \vdash_\omega s \Rightarrow \Delta'; \Pi_1' \qquad \Pi_1' \models \Pi'}{\Delta; \Pi \vdash_\omega s \Rightarrow \Delta'; \Pi'}$$

Dropped keys represent places where work is created for the garbage collector.

2.4 Summary

The novel parts of this type system (as compared to previous work) are:

- The permission keys are fields rather than whole objects, enabling fine-grained protection;
- Multiple (distinct) permissions may be "carved out" of a single nesting location at a time;
- Arbitrary permissions may be packed into existential types.

The work described here however is partial and ongoing. Further work that needs to be addressed includes:

- Defining memory consistency.
- Proving that well-typed programs don't go wrong.
- Proving that effects are respected.
- Coming up with a usable approximation (if not a complete characterization) of the $\models$ relation. In particular, we need a way to use this relation algorithmically during type checking.
- Making the type rules algorithmic. Assuming we have a (probably conservative) way to apply the $\models$ relation, and thus the Transform rule, we only need to address the nondeterminism of the If rules (especially the rule for $E_1 \vee E_2$).

What this paper *does* provide is an interesting way to unify the concepts of effects and uniqueness in a new type system (inspired by Fähndrich and DeLine's adoption and focus). How these concepts are realized is described next.

3 Realizing Effects and Uniqueness Annotations

As is clear from what precedes this discussion, the types used in this proposal are complex and verbose. One cannot expect programmers to want to use such a system. In this section, we explain how commonly proposed, higher-level annotations can be expressed in the full type system. Of course, these annotations do not give one the full expressive power of the complete type system, but the full system can be provided in unusual situations.

One question that needs to be answered is why one should propose a type system that is too complex to use and then hide it underneath a simpler type system. Why not simply use the simpler type system and be done with it? The problem is that no one has yet come up with a type system that successfully unifies effects and uniqueness annotations. It appears that solving the problem correctly requires more formal machinery than expected. A similar situation applies to object-oriented languages: typing "self" for a language with imperative state and overriding requires a complex combination of existential types, bounded polymorphism and recursive types, even though these features are used in stylized ways [17]. This analogy suggests that there may be a way to type-check uses of the annotations described here without requiring a type checker for the complete type system. The algorithm could be proved sound against the complete type system.

The following annotations can be realized with our extension of adoption:

data groups A class may declare data groups. A field is tagged with its parent data group; a data group may also have a parent. Unlike our earlier work [12] (where data groups were called "regions"), a field may be nested in two data groups, but the type system will require such a field to be always carved out of all but one of its data groups at any point in the program.

reference annotations A field, parameter, receiver or return value is tagged with one of the annotations: `unique`, `shared` or `borrowed`, where the latter annotation is legal only for parameters and receivers. Ownership can also be handled as a generalization of `shared`.

effects Any method may be annotated with effects. Since this paper does not use fractional permissions, we do not distinguish between read and write effects. The state accessed is expressed using one of the following forms: `this.`f where f may be a data group; p`.`f where p is a formal parameter; `other` which means anything accessible from \$0.All. We are looking at ways to extend the system to represent effects on state such as `any.`f which represents all f fields (or data groups) of any object accessible from \$0.All.

3.1 Class and Field Annotations

We represent class types by named, potentially recursive, fact-creating functions with one parameter giving the location for the object. The facts include one adoption fact for every field and data group. For simplicity, we assume the existence of a single root data group "All" in which all fields are nested (perhaps

indirectly). The fact for a field includes its type which, for pointers, is an existential whose body is a macro-call that takes the location, the fact-creating function for the type and whether the pointer is "good," usually ($\rho \neq \$0$).[1] The macro function to use is named by the reference annotation and is expanded as part of the realization process:

$$\begin{aligned}
\texttt{\textit{unique}}(\rho, t, c) &= \text{ptr}(\rho) \text{ with } c \multimap (\rho.\text{All} : \text{unit}; t(\rho)) \\
\texttt{\textit{shared}}(\rho, t, c) &= \text{ptr}(\rho) \text{ with } (\cdot; c \Rightarrow (\rho.\text{All} : \text{unit} \prec \$0.\text{All} \wedge t(\rho)) \\
\texttt{\textit{borrowed}}(\rho, t, c) &= \text{ptr}(\rho) \text{ with } (\cdot; c \Rightarrow t(\rho))
\end{aligned}$$

The "borrowed" annotation is legal only for method parameters (and receivers), and indicates that the parameter does not come with permissions on its own; any permissions are provided in the method effects.

For a first example, consider the following Java-like class and its fact-creating function:

```
class Point {
  group Loc;
  int x in Loc;
  int y in Loc;
}
```

$$\begin{aligned}
&\texttt{Point}(\rho) = \\
&\quad \rho.\texttt{Loc} : \text{unit} \prec \rho.\text{All} \wedge \\
&\quad \rho.\texttt{x} : \text{int} \prec \rho.\texttt{Loc} \wedge \\
&\quad \rho.\texttt{y} : \text{int} \prec \rho.\texttt{Loc})
\end{aligned}$$

Next, a rectangle class that has two unique points and a shared string name:

```
class Rectangle {
  group Looks;
  group Dims in Looks;
  unique Point tl in Dims;
  unique Point br in Dims;
  shared String n in Looks;
}
```

$$\begin{aligned}
&\texttt{Rectangle}(\rho) = \\
&\quad \rho.\text{Looks} : \text{unit} \prec \rho.\text{All} \wedge \\
&\quad \rho.\text{Dims} : \text{unit} \prec \rho.\text{Looks} \wedge \\
&\quad \rho.\texttt{tl} : \exists \rho.\texttt{\textit{unique}}(\rho, \texttt{Point}, \rho \neq \$0) \prec \rho.\text{Dims} \wedge \\
&\quad \rho.\texttt{br} : \exists \rho.\texttt{\textit{unique}}(\rho, \texttt{Point}, \rho \neq \$0) \prec \rho.\text{Dims} \wedge \\
&\quad \rho.\texttt{n} : \exists \rho.\texttt{\textit{shared}}(\rho, \texttt{String}, \rho \neq \$0) \prec \rho.\text{Looks}
\end{aligned}$$

3.2 Parameters and Methods Effects

The parameters and return value are packed into an "argument" object pointed to by a global `ap`. We also give general access to a number of global "temporaries." Permissions to access all these globals, the parameters and the return value are passed to and returned from the procedure. When a procedure is called, the return value is uninitialized (if a pointer, has a pointer type with an address variable for which we have no information). At the end, the parameters are uninitialized. The procedure type is polymorphic using a variable for the argument object, the receiver, each pointer parameter and the result (if a pointer value). The permissions for each parameter are typed on procedure entry using the same macros for the pointer annotations, except this time the macro-calls

[1] If we added non-null annotations (as suggested by Fähndrich and DeLine [18]), the condition can be strengthened to "true" for non-null pointers.

are not wrapped in existentials. The return value is typed using the annotation macro-call upon exit.

The effects of the method are realized by permissions that are passed to the procedure and then returned. A data group of a receiver or parameter is represented by a data group of the corresponding location variable. The state `other` is represented by \$0.All. If some of the effects' state are known to be shared (directly or indirectly adopted into \$0.All), the permission will be carved out in advance.

For a simple example, consider a method of `Rectangle` that changes the name of a rectangle:

```
void setName(shared String n) borrowed accesses this.Look;
```

Its realized type is

$$\forall \rho_f, \rho_t, \rho_s. \left(\begin{array}{c} (\$0.\texttt{ap} : \mathrm{ptr}(\rho_f), \mathit{temps}, \rho_f.\texttt{this} : \mathit{borrowed}(\rho_t, \texttt{Rectangle}, \mathrm{true}), \\ \rho_f.\texttt{n} : \mathit{shared}(\rho_s, \texttt{String}, \neg(\rho_s = \$0), \rho_t.\mathrm{Looks} : \mathrm{unit}) \rightarrow \\ \exists \rho_s'.(\$0.\texttt{ap} : \mathrm{ptr}(\rho_f), \mathit{temps}, \rho_f.\texttt{this} : \mathrm{ptr}(\rho_t), \\ \rho_f.\texttt{n} : \mathrm{ptr}(\rho_s'), \rho_t.\mathrm{Looks} : \mathrm{unit}) \end{array} \right)$$

3.3 Discussion

This realization links parameter annotations and effects more strongly than in our previous work: it does not permit parameters to be aliased; it does not permit shared state to be passed borrowed to a method that affects `other`. Intersection types "solve" this problem at the cost of a number of variants, exponential in the number of parameters. On the other hand, the stricter definition is probably a good default; a parameter whose state could be accessed through another parameter or through a global variable should be declared as such. The stricter rule corresponds closely to the new ANSI C `restrict` qualifier as formalized by Foster and others [19].

The realization described here does not fully handle inheritance and virtual overriding because there is no way to express downcasts. Neither does it handle the problem of partially constructed objects. We hope to use Fähndrich and Leino's "monotonic heap states" [16] to describe how the "facts" for an object grow from the facts provided by the superclass to the additional ones provided by the subclass.

4 Related Work

This work was conceived as an extension to the work of Manuel Fähndrich and Robert DeLine called "adoption and focus" [20]. This type system permits a linear pointer to be irrevocably "adopted" by another pointer. Then the pointer can be duplicated (i.e. copied nonlinearly) with a "guarded type" $g \triangleright \tau$ where g is the adopter. Should some piece of the code need to use the pointer and it has access to the adopter, it can "focus" on the adoptee and gain the linear pointer

while temporarily giving up the rights on the adopter. When there is no more need to access the linear pointer (and its linearity and type have been restored), the linear pointer can "disappear" back into the adopter which is then restored. Our system extends/changes the ideas of adoption and focus in the following ways:

1. Reads and writes can be distinguished using "fractional permissions" [13]. For reasons of brevity, this issue is not discussed in this paper.
2. Unlike adoption and focus, "carving out" permissions for a nested key ("focusing" in their terminology) does not make the nesting key (the "adopter") inaccessible. It only forbids the carving out of the *same* nested key again.
3. In our system, adoption (and protection in general) is performed at the level of fields of an object rather than whole objects. This allows finer-grained permissions.

Additionally, "adoption and focus" was described just as a type system, without anything to prove correct. The system described here has made more progress in the direction of defining what correctness means, without reaching the goal. We intend to continue this progress.

Adoption and focus, as well as this work use Alias Types [21], a technique to represent aliasing in the type system. Alias types can be used to precisely describe the shape of recursive data structures [22]. Adoption allows alias types to described uniqueness as well, and our extensions of adoption allow alias types to describe effects.

Effects systems have been defined for functional languages in order to safely deallocate "regions" of data that are no longer being accessed [23]. A region encapsulates a (possibly heterogeneous) set of objects. This work was extended by Walker and others [24] in a capability calculus that has one permission key for each region. These insights were later used in the "adoption and focus" work.

Boyapati and others [25, 26] have incorporated uniqueness, effects and regions in an ownership type system for Java-like languages. They use a permission system over whole objects (not fields as in this work) to prevent data-races. Uniqueness is supported in a few special situations, such as tree-like structures. For the most part, however, the ownership type system prevents object transfer. It appears that some kinds of transfer could be added to their system using "external uniqueness" [6].

One of the attractive features of external uniqueness is that uniqueness is defined for *external* pointers and does not prevent aliasing internal to the object. We incorporated this idea into our system so that if a structure with internal aliasing is described using recursive types, a unique reference could point to it. Using our permission system for effects obviates the need for destructive reads.

5 Conclusions

In this paper, we describe how we can use the ideas of "adoption and focus" to design a type system that takes into account effects and uniqueness in a unified

manner. Conditional permissions permit one to express null pointers without tagged unions. Field adoption allows us to express data groups. We show how high-level annotations (method effects and pointer annotations) can be expressed in this system. Although the work is not fully formalized, especially memory consistency, and we doubt that complete algorithmic type inference is possible, we expect that a practical algorithmic inference system can be defined that will check stylized uses of the type system, such as that used in our realization of annotations.

Acknowledgments

I thank Aaron Greenhouse, Bill Retert and Tim Halloran for their useful comments and corrections on this paper. I also thank Manuel Fähndrich, Dave Clarke and Jan Vitek for conversations that led to clarifying my ideas.

References

1. Boyland, J.: The interdependence of effects and uniqueness. Paper from Workshop on Formal Techniques for Java Programs, 2001 (2001)
2. Hogg, J.: Islands: Aliasing protection in object-oriented languages. In: OOPSLA'91 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications, New York, ACM Press (1991) 271–285
3. Baker, H.G.: 'Use-once' variables and linear objects—storage management, reflection and multi-threading. ACM SIGPLAN Notices **30** (1995) 45–52
4. Minsky, N.: Towards alias-free pointers. In Cointe, P., ed.: ECOOP'96 — Object-Oriented Programming, 10th European Conference. Volume 1098 of Lecture Notes in Computer Science., Berlin, Heidelberg, New York, Springer (1996) 189–209
5. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications, New York, ACM Press (2002) 311–330
6. Clarke, D., Wrigstad, T.: External uniqueness. In Pierce, B.C., ed.: Informal Proceedings of International Workshop on Foundations of Object-Oriented Languages 2003 (FOOL 10). (2003)
7. Leino, K.R.M., Stata, R.: Virginity: A contribution to the specification of object-oriented software. Information Processing Letters **70** (1999) 99–105
8. Leino, K.R.M., Nelson, G.: Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center, Palo Alto, California, USA (2000)
9. Leino, K.R.M., Poetzsch-Heffter, A., Zhou, Y.: Using data groups to specify and check side effects. In: Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation, New York, ACM Press (2002) 246–257
10. Clarke, D.: Object Ownership and Containment. PhD thesis, University of New South Wales, Sydney, Australia (2001)
11. Boyland, J.: Alias burying: Unique variables without destructive reads. Software Practice and Experience **31** (2001) 533–553
12. Greenhouse, A., Boyland, J.: An object-oriented effects system. In Guerraoui, R., ed.: ECOOP'99 — Object-Oriented Programming, 13th European Conference. Volume 1628 of Lecture Notes in Computer Science., Berlin, Heidelberg, New York, Springer (1999) 205–229

13. Boyland, J.: Checking interference with fractional permissions. In Cousot, R., ed.: Static Analysis: 10th International Symposium. Volume 2694 of Lecture Notes in Computer Science., Berlin, Heidelberg, New York, Springer (2003) 55–72
14. Leino, K.R.M.: Data groups: Specifying the modification of extended state. In: OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications, New York, ACM Press (1998) 144–153
15. Xi, H.: Dependent Types in Practical Programming. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA (1998)
16. Fähndrich, M., Leino, K.R.M.: Heap monotonic typestates. In: Informal Proceedings of "International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)", Utrecht University, Netherlands (2003)
17. Bruce, K.C., Cardelli, L., , Pierce, B.C.: Comparing object encodings. In: Theoretical Aspects of Computer Software. Volume 1281 of Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, New York (1997) 415–438
18. Fähndrich, M., DeLine, R.: Declaring and checking non-null types in an object-oriented language. Submitted to OOPSLA 2003 (2003)
19. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation, New York, ACM Press (2002) 1–12
20. Fähndrich, M., DeLine, R.: Adoption and focus: Practial linear types for imperative programming. In: Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation, New York, ACM Press (2002) 13–24
21. Smith, F., Walker, D., Morrisett, J.G.: Alias types. In Smolka, G., ed.: ESOP'00 — Programming Languages and Systems, 9th European Symposium on Programming. Volume 1782 of Lecture Notes in Computer Science., Berlin, Heidelberg, New York, Springer (2000) 366–381
22. Walker, D., Morrisett, G.: Alias types for recursive data structures. In: Types in Compilation: Third International Workshop, TIC 2000. Volume 2071 of Lecture Notes in Computer Science., Berlin, Heidelberg, New York, Springer (2001) 177–206
23. Tofte, M., Talpin, J.P.: Implementation of the typed call-by-value λ-calculus using a stack of regions. In: Conference Record of the Twenty-first Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, New York, ACM Press (1994) 188–201
24. Walker, D., Crary, K., Morrisett, G.: Typed memory management via static capabilities. ACM Transactions on Programming Languages and Systems **22** (2000) 701–771
25. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA'02 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications, New York, ACM Press (2002) 211–230
26. Boyapati, C., Salcianu, A., Beebee, W., Rinard, M.: Ownership types for safe region-based memory management in real-time java. In: Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation, New York, ACM Press (2003) 324–337

Heap Monotonic Typestates (Extended Abstract)

Manuel Fähndrich and K. Rustan M. Leino

Microsoft Research
{maf,rustan}@microsoft.com

Abstract. The paper defines the class of *heap monotonic typestates*. The monotonicity of such typestates enables sound checking algorithms without the need for non-aliasing regimes of pointers. The basic idea is that data structures evolve over time in a manner that only makes their representation invariants grow stronger, never weaker. This assumption guarantees that existing object references with particular typestates remain valid in all program futures, while still allowing objects to attain new stronger typestates. The system is powerful enough to establish properties of circular data structures.

1 Introduction

Types are the main mechanism by which programmers specify properties about data structures that are mechanically checked by today's compilers. Types, however, are a very limited specification tool, in particular in imperative programming languages, where objects evolve over time. As objects evolve, they acquire more properties, and stronger invariants get established. But such new properties cannot be captured in the form of types, since types in mainstream languages capture only properties that hold uniformly from the birth to the demise of an object.

This paper presents a statically checkable *typestate* system. Typestates [9] specify extra properties of objects beyond the usual programming language types. As the name implies, typestates capture aspects of the state of an object. When an object evolves, its typestate may evolve as well. Typestates can be used to restrict valid parameters, return values, or field values, and in doing so provide extra guarantees on internal object invariants.

The main contribution of this paper is that it identifies a class of *heap monotonic typestates*, along with a statically checkable condition on field updates. Under that condition, it can be proven that all object states evolve monotonically: statically observable object invariants only become stronger as objects evolve. At first, the idea may seem restrictive, but it results in a surprisingly liberal programming methodology: our static typestate discipline captures gradual initialization of entire object graphs, and can even prove properties of cyclic structures. A surprising result is that only typestate annotations are needed: there is no need for non-aliasing annotations or assumptions, nor is there a need

to declare read or write effects of methods. These properties put our approach at a minimal distance from ordinary type checking and distinguish it from previous related work on proving stronger program invariants [4, 3, 8, 1]. We believe this system is practical, because it puts no restrictions on the shape of object graphs. As a result, we expect our approach easily to combine with existing approaches for structuring the heap or managing resources, such as ownership types [2] or alias types [10]. Moreover, our typestate system is formulated in such a way that it can take advantage of non-aliasing information, if present.

The rest of the paper is organized as follows: Section 2 introduces heap monotonic typestates by means of an example. Section 3 formalizes typestates in the presence of inheritance and gives sufficient field update conditions to maintain monotonicity. Section 4 discusses further ramifications of monotonic typestate and future extensions. The remaining sections discuss related work and conclude.

2 Motivating example

To illustrate evolving objects, the example in Fig. 1 contains code fragments of a typical compiler front-end. The main data structure is an abstract syntax tree (AST) consisting of AstNode objects. The parsing, name resolution, type checking, and back-end phases are represented. Parsing produces an abstact syntax tree. This tree is then modified by first doing name resolution (ast.ResolveNames(...)), followed by doing type checking (ast.TypeCheck(...)). After type checking, the AST is passed to the back-end (ast.Emit(...)) .

An AST cannot be passed to the back-end without first performing type checking. Similarly, type checking cannot be performed without name resolution. The state of the AST after parsing therefore differs from its states after name resolution and after type checking. Mainstream programming languages do not allow programmers to express such state properties, let alone statically check them. Typestate annotations and typestate checking fill this gap.

We distinguish three states of the AST: "Naked" after parsing, "Bound" after name resolution, and "Typed" after type checking. Figure 1 lists typestate annotated signatures of the methods representing the various front-end phases. Annotation [return:Post("Naked")] states that method Parse returns an AST satisfying state "Naked". The methods performing name resolution, type checking, and code emission are instance methods of AstNode. They have Pre and/or Post annotations to express the typestate expected on entry, and the typestate guaranteed on exit. For instance, the annotation on method ResolveNames specifies that on entry, the receiver (this object) must satisfy state "Naked"; whereas on exit, it will satisfy state "Bound". The annotation on Emit simply requires the receiver to satisfy state "Typed".

So far, we have only seen typestates in their abstract form, that is, as adjectives modifying a type. In that form, typestate annotations may constrain when types are compatible. The other major purpose of typestates is to capture extra data invariants. Figure 2 shows class fragments of subclasses of AstNode with annotations expressing such invariants. In our example, the typestate "Typed"

```
void Main(...) {
  AstNode ast= Parse(filename);
  ast.ResolveNames(emptyEnvironment);
  ast.TypeCheck(emptyTypeEnvironment);
  ast.Emit (...);
}

[return:Post("Naked")] AstNode Parse(string file );

abstract class AstNode {

  [Pre("Naked"),Post("Bound")]
  abstract void ResolveNames(Env env);

  [Pre("Bound"),Post("Typed")]
  abstract void TypeCheck(TypeEnv typeEnv);

  [Pre("Typed")]
  abstract void Emit (...);

  ...
}
```

Fig. 1. Front-end of a compiler

captures the fact that the type field of Expression objects has been initialized to a non-null pointer to a Type object. This invariant is expressed by the annotation [NotNull(WhenEnclosingState="Typed")] on field type of class Expression. Similarly, the typestate "Bound" captures the fact that the binding field of Identifier objects in the AST has been initialized to a non-null pointer to the binding node.

These two annotations describe the relation between the typestate of an object and the *atomic properties* of its fields (in this case non-nullity).

Furthermore, we may relate the typestate of an object to the typestates of objects pointed to in its fields. For example, the UnaryExpr class needs to specify the state of the operand sub-expression in the AST. This state is dependent on the state of the unary expression node itself. In our example, the relation is simple: if the unary expression object satisfies state "Naked" (resp. "Bound", "Typed"), then so does the sub-expression. The annotation [InState("Naked", WhenEnclosingState="Naked")] expresses the first of these three dependencies.

The advantage of these data invariants should now be evident. Method TypeCheck can rely on the fact that the binding field of Identifier objects is non-null. Similarly, the back-end can rely on the fact that the type field of expression objects is non-null.

2.1 The aliasing problem

A technical reason that has kept typestate out of mainstream languages is the problem of maintaining correct typestate information in the presence of aliasing. To appreciate this problem, consider a method StripTypes:

```
// Set all type fields of expressions to null.
void StripTypes(AstNode ast);
```

```
class Expression : AstNode {
  [NotNull(WhenEnclosingState="Typed")]
  Type type;
  ...
}

class Identifier : Expression {
  string name;

  [NotNull(WhenEnclosingState="Bound")]
  AstNode binding;
  ...
}

class UnaryExpr : Expression {
  Operator oper;

  [NotNull(WhenEnclosingState="Naked,Bound,Typed")]
  [InState("Naked", WhenEnclosingState="Naked")]
  [InState("Bound", WhenEnclosingState="Bound")]
  [InState("Typed", WhenEnclosingState="Typed")]
  Expression operand;

  ...
}
```

Fig. 2. Typestate invariants of some front-end classes

This method sets all type fields of Expression objects back to null. The problem with this method is that the typestate of abstract syntax trees is weaker on exit than it is on entry. Thus, other pointers to nodes of the same abstract syntax tree may need to have their typestate weakened. Suppose for example that after type checking we build control-flow graphs (CFGs) that internally keep references to AstNodes satisfying typestate "Typed".

```
ast.TypeCheck(emptyTypeEnvironment);
CFG cfg= BuildCFG(ast);
StripTypes(ast);
WorkOnCFG(cfg);
```

The above code sequence is problematic, since the AstNode references in the cfg object are annotated to satisfy typestate "Typed", but after the call to StripTypes, these references point to objects that do not satisfy the typestate "Typed". In order to correctly track such non-local and non-monotonic typestate changes, strict non-aliasing regimes must be followed. The earliest attempts at typestate checking appear in a language called Nil that completely rules out aliasing in pointer structures [9]. Vault is a more recent programming language that permits typestate checking and strong aliasing control [3]. However, once an object's aliases are no longer statically known in Vault, its typestate needs to be *frozen*, that is, it can no longer change, except temporarily [5].

Although some typestate protocols will always require aliasing control (for example, open/close protocols), we identify in this paper a class of *heap monotonic* typestates that do not require aliasing control. The idea behind heap monotonic typestates is that, once a certain typestate is reached, no future changes to the

object will ever invalidate that typestate. Monotonic typestate checking makes it possible to capture object references in arbitrary typestates without the need to invalidate such references on future object updates. In our proposal, the method StripTypes cannot be typestate checked, since the update of the type field to null violates the monotonicity of our typestates.

3 Typestate formalization

We start with a number of definitions. Let Σ be a set of identifiers used to represent local variables and fields of objects. We use $\sigma \in \Sigma^+$ to represent access paths. Let V be the domain of values, including locations L used during program execution. A heap H is a map $L \times \Sigma \rightarrow V$, mapping location-field pairs to values. The local variables are accessed via a distinguished location containing all locals ℓ_{locals}. For convenience, define $H(\sigma) = H(\ell_{\text{locals}}, \sigma)$ and $H(\ell, x.\sigma') = H(H(\ell, x), \sigma')$.

Definition 1 (Heap monotonic predicate). *A predicate P on values and heaps is* heap monotonic*, if $P(v, H) \Rightarrow P(v, H')$ for every value v and heaps H and H', where H' is obtained from H by updates allowed by the static program semantics.*

Examples of heap monotonic predicates in most programming languages are notnull(v), null(v), dynamictype(v) $\leq T$, $v \geq 5$, etc.[1] Let $\mathcal{A}$ be a fixed set of heap monotonic predicates.

In a non-object-oriented setting, a typestate for an object o of type T is simply a named predicate over the fields of o. We write A_T for such a typestate predicate, where A is the name of the state. In this paper, we are interested in heap monotonic typestates. A typestate is by definition heap monotonic if it only depends on heap monotonic predicates. If field updates are restricted to preserve monotonicity (Sect. 3.4), then each heap monotonic typestate is itself a heap monotonic predicate. Thus the typestate of an object can depend on the typestates of objects stored in its fields.

3.1 Heap monotonic typestate in the presence of inheritance

In an object-oriented setting with single implementation inheritance, an object can be viewed as a list of *frames*, one per class in the inheritance path from the root class Object to its dynamic type.

We specify typestates of objects by giving a typestate per class frame. Thus, an object with dynamic type AstNode has a typestate for the Object frame and a separate typestate for the AstNode frame. A typestate A_T only describes fields declared in T, none in super or sub-classes of T.

We further need a technical device to abstract the typestate of sub-classes of an object, since the dynamic type of an object is rarely known statically. We thus

[1] These examples are trivially heap-monotonic, since they are independent of the heap.

introduce typestate predicates of the form $A_{\leq T}$. The meaning of this predicate is that A_S holds for every subclass S of T. Formally, if $\mathsf{supertype}(Q) = T$

$$A_{\leq T} \iff A_T \wedge A_{\leq Q}$$

Thus, the typestate annotations in our examples so far are interpreted as $A_{\leq \mathsf{Object}}$.

Let $\mathcal{MP}$ be the set of heap monotonic predicates consisting of $\mathcal{A}$ and all typestate predicates A_T. The interpretation of a typestate A_T is a map $[\![A_T]\!]\colon \Sigma \to 2^{\mathcal{MP}}$, mapping fields of T to the heap monotonic predicates that are true for the value contained in the field, when the enclosing object frame T satisfies state A. We interpret such sets of predicates as conjunctions.

Note that our typestates are not mutually exclusive. It is perfectly fine to have an $\mathsf{Expression}$ object in typestates $\mathsf{Bound}_{\mathsf{Expression}} \wedge \mathsf{Typed}_{\mathsf{Expression}}$. In fact, there is nothing in our typestate definitions that explicitly orders typestates. Since states are monotonic, "transitioning" an object from A_T "to" B_T results in an object with $A_T \wedge B_T$.

3.2 Language

We work with a small core object-oriented language consisting of classes, fields, and methods. Without loss of generality, we assume that each field uniquely identifies its declaring class. We describe statements modifying or accessing the heap and method calls, but omit other details. Typestate annotations are assumed to be given in the form of typestate predicates A_T discussed above, but we do not provide formal syntax and interpretation for such annotations here. The syntax used in the examples is one possible approach. We assume the language is statically typed, similar to C# or Java, and focus only on typestates.

method		$T.m(x_1, \ldots, x_n)$ returns z $\{\iota\}$
instruction sequence	$\iota ::=$	$\cdot \mid s; \iota$
instruction	$s ::=$	$x := y \mid x.f := y \mid y := x.f \mid y := \mathsf{null} \mid y := \mathsf{new}\, T()$
		$z := y_0.m(y_1, \ldots, y_n) \mid$
		$z := y_0.T.m(y_1, \ldots, y_n) \mid \cdots$

Instructions of interest consist of variable copy, field assignment, field read, null assignment, object construction, and virtual and direct method call.

3.3 Dynamic semantics

The semantics of the language is a standard small-step semantics of the form $(H, \iota) \to (H, \iota)$, relating machine states consisting of a heap H and instruction sequence ι. We omit the details for space reasons.

Let $[\![H]\!] : \Sigma^+ \to 2^{\mathcal{MP}}$ be the largest predicate assignment for heap H consistent with the rules in Fig. 3. We use the largest, rather than the inductively defined set in order to allow more properties of circular heap structures. Rule [H-atom] provides atomic heap-monotonic predicates. For example, for any location $\ell \neq \mathsf{null}$, $\mathsf{notnull}(\ell, H)$ holds. Rule [H-ts] relates predicates of fields $\sigma.f$ with

$\boxed{H \vdash \sigma : M}$

$$\frac{H(\sigma) = \ell \wedge p(\ell, H) \qquad p \in \mathcal{A}}{H \vdash \sigma : \{p\}} \text{ [H-atom]} \quad \frac{\forall f \in T. H \vdash \sigma.f : [\![A_T]\!](f)}{H \vdash \sigma : \{A_T\}} \text{ [H-ts]}$$

$$\frac{H \vdash \sigma : M_1 \wedge H \vdash \sigma : M_2}{H \vdash \sigma : M_1 \cup M_2} \text{ [H-and]} \qquad \frac{H(\sigma) = \mathsf{null}}{H \vdash \sigma : \{A_T\}} \text{ [H-ts-null]}$$

Fig. 3. Rules for deriving heap properties

the typestate of σ. Rule [H-ts-null] is a special case for null, which we consider to have every typestate.

$[\![H]\!]$ maps each access path σ to the observable heap monotonic predicates that hold for the value accessed through σ. Note that $[\![H]\!]$ is total, mapping access paths not in H to the empty set.

3.4 Static semantics

In this section, we formalize parts of the static semantics. We give type rules for the statements in our language. Of particular interest are the rules for interpreting and proving certain typestates, as well as the field update rule. We end by showing how to typestate check a small example.

Typestate checking cannot simply use a typestate environment (mapping identifiers to typestates) akin to a type environment, because we want to prove new typestates for a particular pointer stored at path σ, once sufficiently strong properties of its fields $\sigma.f$ are known. For this purpose, it is necessary to keep associations (akin to must-aliasing) that remember that a particular variable y holds the same value as $x.f$.

The static semantics thus uses two auxiliary structures, a heap abstraction $S\colon R \times \Sigma \to R$ and a predicate map $E\colon R \to 2^{\mathcal{MP}}$, where R is a finite set of symbolic pointers ρ. The heap abstraction S maps a symbolic pointer and a field to the symbolic pointer contained in that field. The predicate map E maps each symbolic pointer ρ to a set of predicates known to hold for ρ. Local variables are looked up as fields of a distinguished pointer ρ_{locals}. We use the short-hands $S(\sigma) = S(\rho_{\text{locals}}, \sigma)$ and $S(\rho, x.\sigma) = S(S(\rho, x), \sigma)$.

A symbolic pointer ρ abstracts an actual heap pointer in the following way. Each symbolic pointer corresponds to exactly one heap pointer. Multiple distinct symbolic pointers may correspond to the same heap pointer. For locals, the information is conservative must-alias information, that is, if $S(x) = S(y)$, then at runtime, $H(x) = H(y)$. The information for locals can be kept in synch with the actual execution, because in our language, locals are only assigned directly. For fields, the abstraction is more subtle. It doesn't correspond to must-aliasing exactly because we allow the abstraction to be outdated. For example, if $S(y) = S(x.f)$, then either $H(y) = H(x.f)$ for the current heap H or there is a past heap where the object referred to by x contains the current value of y in field

f, that is, a past $H' \leq H$, such that $H'(H(x), f) = H(y)$ This information is crucial, since it states that at some point in the past (heap H'), the must-aliasing information was correct. This allows us to deduce that if $H(x.f) \neq H(y)$, then there was an assignment to field f between heap H' and the current heap.

We now proceed to the typestate rules for each statement kind and observe how this static information is maintained and used. The static well-typestate relation for statements has the form $S, E \vdash \iota : S', E'$, where S and E are the static structures prior to the execution of statements ι, and S' and E' are the static structures after execution of ι.

Variable copy

$$\frac{S[(\rho_{\text{locals}}, x) \mapsto S(y)], E \vdash \iota : S', E'}{S, E \vdash x := y; \iota : S', E'}$$

The rule states that the remainder of the instructions ι are checked under the assumption $S(x) = S(y)$.

Field access

$$\frac{\begin{array}{l} S, E \vdash x.f : M \\ \rho \text{ fresh} \quad S(x) = \rho_x \\ S[(\rho_{\text{locals}}, y) \mapsto \rho][(\rho_x, f) \mapsto \rho], E[\rho \mapsto M] \vdash \iota : S', E' \end{array}}{S, E \vdash y := x.f; \iota : S', E'}$$

The result of reading field f is recorded under a fresh symbolic pointer ρ corresponding to the current pointer in $x.f$. We use a fresh pointer here, since the current static knowledge $S(x.f)$ may be outdated. But after the statement, we record the equality $S(x.f) = S(y)$, since it is definitely true at this point in the execution. The predicates M known to hold for $x.f$ prior to the statement are recorded in E for the fresh pointer ρ. The auxiliary judgment $S, E \vdash \sigma : M$ is used to deduce predicates for particular access paths. The rules are shown in Fig. 4.

For field accesses, the rules in Fig. 4 can prove properties in two ways: either by rule [TS-elim], applying knowledge of the typestate of x to the field $x.f$. Alternatively, by rule [Loc], where we use knowledge of the properties of the symbolic pointer $\rho = S(x.f)$ directly. It is not immediately obvious why this second way is sound, since we know that $x.f$ could be pointing to a new object, not corresponding to ρ. Fortunately, our rule for field update (shown later) enforces strong enough properties, that rule [Loc] can be proven sound. It is however necessary to restrict the knowledge of $E(\rho)$ to the set of predicates $\bigcup \mathcal{O}_{x.f}$ observable through path $x.f$. To see why, consider the following code snippet:

```
y := x.f;
y.EstablishStateA (); // establishes typestate A for y
// does x.f have typestate A?
```

$\boxed{S, E \vdash \sigma : M}$

$$\frac{E(S(x)) \supseteq M}{S, E \vdash x : M}\ [\text{Var}] \qquad \frac{\begin{array}{c} S(x.f) = \rho \\ E(\rho) \cap \bigcup \mathcal{O}_{x.f} \supseteq M \end{array}}{S, E \vdash x.f : M}\ [\text{Loc}]$$

$$\frac{S, E \vdash \sigma : M_1 \qquad S, E \vdash \sigma : M_2}{S, E \vdash \sigma : M_1 \cup M_2}\ [\text{Union}] \qquad \frac{}{S, E \vdash \sigma : \emptyset}\ [\text{Empty}]$$

$$\frac{S, E \vdash \sigma : \{A_T\} \qquad [\![A_T]\!](f) \supseteq M}{S, E \vdash \sigma.f : M}\ [\text{TS-elim}] \qquad \frac{\forall f \in T\,.\, S, E \vdash \sigma.f : [\![A_T]\!](f)}{S, E \vdash \sigma : \{A_T\}}\ [\text{TS-intro}]$$

Fig. 4. Rules for proving predicates of access paths

$\boxed{\mathcal{O}_\sigma}$

$$\begin{array}{l} \mathcal{O}_x = \{\mathcal{MP}\} \\ \mathcal{O}_{\sigma.f} = \{[\![A_T]\!](f) \mid A_T \in \bigcup \mathcal{O}_\sigma\} \qquad f \in T \end{array}$$

Fig. 5. Observable predicates for a given access path

We have to consider that $x.f$, after the call to EstablishStateA, may differ from y. (We make no assumptions about what is or is not modified.) There are two cases to consider. If $\bigcup \mathcal{O}_{x.f} \ni A_T$, then there is some typestate B_U of class U declaring field f such that $[\![B_U]\!](f) \ni A_T$ and we can conclude that $x.f$ satisfies typestate A, since—as we will see below—any update to $x.f$ during the call to EstablishStateA must have updated the field with an object satisfying state A_T. Otherwise ($\bigcup \mathcal{O}_{x.f} \not\ni A_T$), we cannot conclude $x.f$ has typestate A, since an update could have stored an object in $x.f$ not satisfying state A. As an example, consider field UnaryExpr.operand where $\mathcal{O}_{x.\mathsf{operand}} = \{\{\mathsf{notnull}, \mathsf{Naked}_{\leq \mathsf{Object}}\}, \{\mathsf{notnull}, \mathsf{Bound}_{\leq \mathsf{Object}}\}, \{\mathsf{notnull}, \mathsf{Typed}_{\leq \mathsf{Object}}\}\}$, therefore

$$\bigcup \mathcal{O}_{x.\mathsf{operand}} = \{\mathsf{notnull}, \mathsf{Naked}_{\leq \mathsf{Object}}, \mathsf{Bound}_{\leq \mathsf{Object}}, \mathsf{Typed}_{\leq \mathsf{Object}}\}$$

Field update

$$\frac{\begin{array}{l} S, E \vdash y : M \qquad M \supseteq \mathcal{U}_f(x) \\ S(x) = \rho_x \\ S' = S[(\rho_x, f) \mapsto S(y)] \\ S', E \vdash \iota : S'', E'' \end{array}}{S, E \vdash x.f := y; \iota : S'', E''}$$

The field update rule is the most crucial piece in our approach. We must first find an upper bound $\mathcal{U}_f(x)$ on the observable predicates of field $x.f$. This upper bound must account for all predicates of field f that any other access path to $x.f$

may already know or may be establishing. We will return below to our actual definition of $\mathcal{U}_f(x)$.

The field update is safe, if the condition $M \supseteq \mathcal{U}_f(x)$ is satisfied. This condition ensures that the update does not invalidate the typestate assumptions of any other access path. The static heap approximation is updated to reflect that at this point, $S'(x.f) = S'(y)$.

Let us now define $\mathcal{U}_f(x)$ to be the set

$$\bigcup\{M \mid \exists\sigma.\sigma \neq x \wedge H(\sigma) = H(x) \wedge (M \in \mathcal{O}_{\sigma.f}) \wedge ([\![H(x.f)]\!] \cup M \text{ consistent})\}$$

Note that we formulate $\mathcal{U}_f(x)$ in terms of knowledge of the dynamic heap H that in general won't be statically known. In the absence of any aliasing information on x or knowledge about the value $x.f$, the bound goes up to $\bigcup \mathcal{O}_{z.f}$. In this case, field updates require that the written value satisfy all predicates that could ever be observed of field f. This worst estimate allows each field to go only from the null-initialized state (as established on entry to the constructor), to the fully initialized state.

Another extreme case is when we know that x is the only pointer to the object whose field is updated (depending on the programming language, for example during or right after construction). In that case, $\mathcal{U}_f(x) = \emptyset$, and any update is valid (even a non-monotonic one).

Another possible case is when we know something about the current value of $x.f$. In general, we need to include in $\mathcal{U}_f$ only predicate sets M that are consistent with the current properties of the field $[\![H(x.f)]\!]$. For example, if $x.f$ is null, we can compute $\mathcal{U}_f(x)$ to be the union of all observable predicates for f that are consistent with null. Consider field operand of class UnaryExpr in Fig. 2. If we know that $x.f$ is null at the moment of the update, we can conclude that no pointer to the unary expression object can assume it in any of the typestates Naked,Bound,Typed, since they all include predicate notnull, which is inconsistent with the current value of $x.f$.

Methods A virtual method signature consists of pre and post predicates for each method parameter and result, including the receiver this. We refer to these predicates by $\mathsf{pre}(m.x)$ and $\mathsf{post}(m.x)$, where m is the name of a declared method, and x is the parameter name, this, or return. Annotations [Pre(A)] on x translate to $\mathsf{pre}(m.x) = A_{\leq \mathsf{Object}}$. Annotations [Post(B)] on x translate into $\mathsf{post}(m.x) = B_{\leq \mathsf{Object}}$.

A particular implementation of method m in a class T is referred to as $T.m$. The signature of $T.m$ differs with respect to the virtual method signature only in the treatment of the receiver post condition. If the post condition for the receiver in a declared virtual method is $A_{\leq \mathsf{Object}}$, then the post-condition for the receiver in a particular implementation method $T.m$ is weaker, namely

$$\mathsf{post}(T.m.\mathsf{this}) = \bigwedge_{S \geq T} A_S$$

that is, the conjunction of all typestates A_S for class frames S at or above T. The frames of strict subclasses of T obtain no stronger properties.

It should intuitively be clear why this is so. A method implementation $T.m$ can only directly affect the state of the object at or above class frame T. To produce a deep post condition of the form $A_{\leq \mathsf{Object}}$, a virtual dispatch is needed.

We now give the conditions for typestate checking of a method body. Assume $x_0 = \mathsf{this}$.

$$\frac{\begin{array}{ll} S = [(\rho_{\text{locals}}, x_i) \mapsto \rho_i] & i = 0..n, \rho_i \text{ fresh} \\ E = [\rho_i \mapsto \mathsf{pre}(T.m.x_i)] & i = 0..n \\ S, E \vdash \iota : S', E' & \\ S', E' \vdash x_i : \mathsf{post}(T.m.x_i) & i = 0..n \\ S', E' \vdash y : \mathsf{post}(T.m.\mathsf{return}) & \end{array}}{\vdash T.m(x_1, \ldots, x_n) \text{ returns } y \ \{\iota\}}$$

The first two lines describe the initial environment S, E in which the method body is typed. We assume a distinct symbolic pointer ρ_i for each parameter, and populate E with the respective preconditions. Note that we need not have any knowledge about possible aliasing of the parameters. The third line checks the body ι, resulting in the static structures S' and E' describing the typestates at exit of the method. Finally, the last two lines check the post-conditions of the parameters, this, and the result.

We assume that each class T implements each virtual method of any parent class. If no implementation is explicitly given, the implementation

$$\{\ \textbf{return this.base}.\mathsf{m}(x_1,\ldots,x_n);\ \}$$

is assumed, where base is the immediate supertype of T. This requirement is necessary to check the correctness of any specified typestate changes on the receiver this.

Method calls We allow both virtual calls and non-virtual (direct) method calls. The two differ only in whether the virtual method signature or a particular implementation signature is used. We give the rule for virtual calls. Direct calls look identical, but every occurrence of m is replaced with $T.m$. Assume $x_0 = \mathsf{this}$.

$$\frac{\begin{array}{ll} S, E \vdash y_i : \mathsf{pre}(m.x_i) & i = 0..n \\ E' = E[S(y_i) \mapsto E(S(y_i)) \cup \mathsf{post}(m.x_i)] & i = 0..n \\ S' = S[(\rho_{\text{locals}}, z) \mapsto \rho] & \rho \text{ fresh} \\ E'' = E'[\rho \mapsto \mathsf{post}(m.\mathsf{return})] & \\ S', E'' \vdash \iota : S''', E''' & \end{array}}{S, E \vdash z := y_0.m(y_1, \ldots, y_n); \iota : S''', E'''}$$

The first line ensures that the arguments in the calling context satisfy the preconditions of the virtual method parameters. The second line joins the post typestate of each parameter to the current knowledge in the predicate map. The third line adds a fresh symbolic pointer ρ for z to the store abstraction. The fourth line adds the result typestate for ρ to the predicate map. The last line ensures that the static environment right after the method call is sufficient to prove typestate safety of the remaining instruction sequence ι.

3.5 Example revisited

We now return to our motivating example and show the TypeCheck method for unary expressions.

```
[Pre("Bound"),Post("Typed")]
UnaryExpr.TypeCheck()
{
  y := this.operand;
  y.TypeCheck();
  this.type := new Type(...);
}
```

The initial environment is as follows: $S(\mathsf{this}) = \rho_0, E(\rho_0) = \{\mathsf{notnull}, \mathsf{Bound}_{\leq \mathsf{Object}}\}$. After the assignment to y, we also have $S(y) = \rho_1, S(\rho_0, \mathsf{operand}) = \rho_1, E(\rho_1) = [\![\mathsf{Bound}_{\mathsf{UnaryExp}}]\!](\mathsf{operand}) = \{\mathsf{notnull}, \mathsf{Bound}_{\leq \mathsf{Object}}\}$. Our static information satisfies the precondition to invoke TypeCheck on y. On return from the call, the environment is updated to $E(\rho_1) = \{\mathsf{notnull}, \mathsf{Bound}_{\leq \mathsf{Object}}, \mathsf{Typed}_{\leq \mathsf{Object}}\}$. After the assignment to this.type, the environment is updated to $S(\rho_0, \mathsf{type}) = \rho_2, E(\rho_2) = \{\mathsf{notnull}\}$. Call this environment S', E'. At this point, we can prove the post condition on the receiver: $\mathsf{Typed}_{\mathsf{Object}} \wedge \mathsf{Typed}_{\mathsf{AstNode}} \wedge \mathsf{Typed}_{\mathsf{Expression}} \wedge \mathsf{Typed}_{\mathsf{UnaryExpr}}$. The first two predicates in the conjunct are trivial, since their typestate mapping is empty. So for each field f of Object and AstNode, we can prove $S', E' \vdash \mathsf{this}.f : \emptyset$, and then conclude via rule [TS-intro] (twice) and [TS-union] that $S', E' \vdash \mathsf{this} : \{\mathsf{Typed}_{\mathsf{Object}}, \mathsf{Typed}_{\mathsf{AstNode}}\}$. For $\mathsf{Typed}_{\mathsf{Expression}}$, we need to prove $S', E' \vdash \mathsf{this.type} : \{\mathsf{notnull}\}$, which we do via rule [Loc]. Similarly, we prove $S', E' \vdash \mathsf{this.operand} : \{\mathsf{notnull}, \mathsf{Typed}_{\leq \mathsf{Object}}\}$ via [Loc].

3.6 Soundness

The soundness of the system is subtle because it relies on the field update guarantees and the use of out of date field information. We have proven soundness of the hard cases (field update and method call) via a standard subject reduction approach [11].

4 Discussion

This section discusses some additional properties of heap monotonic typestates and considers possible extensions.

4.1 DAGs and circular structures

Our typestate proposal works for arbitrary graph structures, not just trees or DAGs. Establishing arbitary typestate relations among DAG nodes is done by traversing the DAG as if it were a tree. To avoid the duplicate traversals, dynamic typestate tests (see next subsection) can be used.

Surprisingly, the static proof technique described here can prove typestate properties of circular structures as well. Consider a general graph, where each node satisfies typestate A_N, if some internal field f is non-null, and all successor nodes are in A_N. The general way to establish that each node in an arbitrary graph satisfies A_N is to first build a corresponding DAG, where the missing back pointers point to a dummy object for which it is trivial to establish typestate A_N. It is then possible to prove inductively that each node satisfies A_N. After that, do one more traversal, where all pointers going to the dummy node are updated to point to their intended node. Since their intended target is already in typestate A_N, the update is okay.

Since the approach requires updating of back pointers, it can only be used to establish the ultimate typestate of each graph node. It seems not possible to gradually transition the entire circular graph to better typestates as in our AST example.

4.2 Dynamic typestate tests

Our approach can prove properties of a DAG inductively, simply by traversing it as a tree. However, shared subtrees have to be traversed multiple times. It would be desirable to dynamically record knowledge of an established typestate in a field, and allow a dynamic test of such a field to infer the entire typestate of the object.

Let's call such a field a typestate designator and mark these fields specially with an annotation [TypeState]. Further assume, that such fields must be of type string, and that we have the global invariant that for every object o, if $o.f$ is a typestate designator, and $o.f$ is equal to "A", then o satisfies $A_{\leq \mathsf{Object}}$.

Writing to a typestate designator requires a compile-time known string "A", and a proof that the enclosing object satisfies $A_{\leq \mathsf{Object}}$. Furthermore, in the true branch of a conditional test of the form if (o.f == "A"), the typestate of o can be assumed to be $A_{\leq \mathsf{Object}}$.

Using this device, DAG traversals can establish typestate properties without visiting nodes more than once.

4.3 Relational atomic predicates

So far, all our heap monotonic atomic predicates are unary, that is, they involve exactly one value of a single field. An obvious generalization is to allow relational predicates, such as x≤y, where x and y are both fields of the same class frame. Updates to such fields are slightly more tricky, since to maintain the relational property, two updates may be required, and the property may not hold after the first update. As long as one can statically prove that both updates occur atomically, such extensions can be handled.

The requirement that both fields are of the same class frame is to maintain modular soundness. Without this requirement, modifications to a field of class T, may invalidate the invariant of a subclass S of T, but the code in class T has no knowledge of such an invariant.

4.4 Concurrency

Our typestate checking remains sound in the presence of concurrency. That static rules indeed assume that after each instruction, every field could be updated by another thread (provided the update satisfies our field update rule).

5 Related work

The extended static checker (ESC) project uses theorem proving to enforce method-level specifications and object invariants [4]. The enforcement of object invariants however is sound only under strict non-aliasing conditions. The work on Vault [3] and Roles [8] soundly enforce object invariants and allow state changes. However, both systems require non-aliasing assumptions. Earlier work on alias types also allows incrementally establishing data structure invariants, but only under non-aliasing assumptions [10].

The present system has the advantage that it is useful even without any non-aliasing assumptions, but it can exploit non-aliasing assumptions to allow non-monotonic typestate changes.

The work on type checking Java byte code for safety properties by Freund and Mitchell considers only whether the constructor of a newly allocated object is called before other methods [7]. It does not enforce field initializations or other object invariants.

The motivation for the present work stems in part from our prior work on guaranteeing initialization of object fields to non-null pointers [6], and also from typestate-like code comments we found in a front-end written by Dave Hanson.

6 Conclusion

Initialization of objects often happens gradually over the lifetime of an object, rather than during execution of the constructor alone. We believe that monotonically evolving typestate provides a good match for capturing such evolving object invariants. It has the advantage over prior work that it requires no non-aliasing guarantees, but can exploit them if they are present. The resulting programming model seems flexible.

References

1. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230, November 2002.
2. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings*, pages 48–64, October 1998.

3. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.
4. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.
5. Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, June 2002.
6. Manuel Fähndrich and K. Rustan M. Leino. Non-null types in an object-oriented language, 2002. Presented at the 2002 Workshop on Formal Techniques for Java-like Languages.
7. Stephen N. Freund and John C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.
8. Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages*, 2002.
9. Robert. E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, 12(1):157–171, January 1986.
10. David Walker and Greg Morrisett. Alias types for recursive data structures. In *Proceedings of the 4th Workshop on Types in Compilation*, September 2000.
11. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

Towards a Model of Encapsulation

James Noble[1], Robert Biddle[1], Ewan Tempero[2], Alex Potanin[1], and Dave Clarke[3]

[1] School of Mathematical and Computing Sciences, Victoria University of Wellington
[2] Department of Computer Science, The University of Auckland
[3] Institute of Information and Computing Sciences, Utrecht University

Abstract. Encapsulation is a founding principle of object-oriented programming: to this end, there have been a number of recent of proposals to increase programming languages' support for encapsulation. While many of these proposals are similar in concept, it is often difficult to describe their effects in practice, or to evaluate clearly how related proposals differ from each other. We are developing a general topological model of encapsulation for object-oriented languages, based on a program's object graph. Using this model, we can characterise a range of confinement, ownership, and alias protection schemes in terms of their underlying encapsulation function. This analysis should help programmers understand the encapsulation provided by programming languages, assist students to better compare and contrast the features of different languages, and help language designers to craft the encapsulation schemes of forthcoming programming languages.

1 Introduction

We are developing a simple topological model of encapsulation in object-oriented systems, programming languages, and more general computer systems. This model has three parts — an *access graph* describing the topology of the system under consideration; an *encapsulation function* describing how particular parts of the system are encapsulated; and an *encapsulation constraint* that, when true, captures the fact that the encapsulation function partitions the access graph so that so part of it is actually encapsulated.

For this position paper, we take a statement of Dan Ingalls as the defining quality of encapsulation[1]:

> *No component in a complex system should depend on the internal details of any other component.* [14]

As far as we are aware, there is as yet no agreed formalisation of encapsulation: our model is intended to be a step in that direction.

[1] Glossing over the fact that Ingalls was defining *modularity* rather than *encapsulation*.

1.1 Access Graph

The first part of our model is an *access graph*. An access graph models the topology of a system: nodes ($\mathcal{N}$) in the graph represent individual objects in the model, and edges ($\mathcal{E}$) between nodes represent accesses between objects.

Nodes and edges may (or may not) be labelled: node labels are typically given by boolean predicates (as functions from $\mathcal{N}$ to $\mathbb{B}$); or functions to (sets of) nodes, components, or model-specific sets; edges may be labeled similarly (e.g. i, r, w). Our formulation of encapsulation does not depend upon whether the graph is directed: in an undirected graph we take an edge (a, b) to mean that a accesses b and b accesses a.

Some notation: we write $\mathcal{S}$ for all the objects in the system, that is, the set of all nodes in the graph; $\{a\}\triangleright$ to be the set of all other nodes that have edges beginning from a and $\triangleright\{a\}$ to be the set of all other nodes that have edges ending at a; $\{a\}\triangleright\!\!\triangleright$ and $\triangleright\!\!\triangleright\{a\}$ for their transitive closures (all other nodes which can be reached from a and which can reach a respectively); $a \longrightarrow b$ to mean that there is an edge from a to b; *paths*(a, b) for the set of all paths (a set of sequences of nodes) from node a to node b; $a \sqsubseteq b$ to mean that a is a dominator for b, that is, every path to b from some nominated entry point leads through a. If edges are labelled, we may subscript graph operations to restrict to matching edges (i.e. $a\triangleright\!\!\triangleright_i$ is all other nodes reachable from a along edges labelled i).

Our access graph is unremarkable in its suburban monotony, being at heart a directed graph: our whole model of encapsulation is similarly simple-minded. The reason for this naïve formulation is that the access graph is itself an abstraction: our model of encapsulation will work over *any* directed graph model that meets these criteria. In this position paper, we will consider only *object graphs*, with objects as nodes and their variables (references between objects) as edges. We may build this graph informally based on some kind of object identifiers, use Zeller's Memory Graphs [19] or Hoare and Jifeng's trace model [12], or some other formulation of an object-oriented program that also is based on a directed graph [10]. We can decorate the graph as necessary to distinguish between static or dynamic object accesses, the classes, packages, modules, block structure, or files to which objects belong. We expect the ubiquity of the access graph will allow us to address some other kinds of encapsulation in object-oriented systems and encapsulation in non-object-oriented (and non-informatic) systems.

1.2 Encapsulation Function

The second part of the model is the encapsulation function. The encapsulation function partitions the system ($\mathcal{S}$) by taking an identifier of an *encapsulated component* ($\mathcal{C}$) to a three-tuple of sets of access graph nodes that respectively describe the encapsulation *boundary* ($\mathcal{B}$), the *inside* ($\mathcal{I}$), and the *external* references ($\mathcal{R}$) of the component. We will use e.g. $\mathcal{B}$ to represent $\mathcal{B}_c$ for some given $c \in \mathcal{C}$.

$$\begin{array}{l} \mathcal{C} : \mathrm{id} \\ \mathcal{S}, \mathcal{B}, \mathcal{I}, \mathcal{R}, \mathcal{O} : \mathbb{P}\mathcal{N} \\ \mathbf{e} : \mathcal{C} \rightarrow (\mathcal{B}, \mathcal{I}, \mathcal{O}) \end{array}$$

The boundary of a component represents the interface that components presents to the rest of system, while the inside is that part of the component which is encapsulated, and may itself be a network of interconnected objects. The inside of a component can only be accessed via the component's interface, that is, by crossing its encapsulation boundary. The external references of a component are nodes which the boundary or inside accesses directly, but which are not contained within boundary or inside of the component. Finally, the *outside* ($\mathcal{O}$) of a component are all nodes which are neither inside the component nor on its boundary — including the component's external references.

The idea that encapsulation or confinement is fundamentally a partition of some software space is not a new one: a range of recent encapsulation schemes have been (formally) described in such terms [3, 4, 18]. Here, we argue that this is not accidental: rather, encapsulation is essentially defined by such a partition.

$$\mathcal{O} = \mathcal{S} - (\mathcal{B} \cup \mathcal{I})$$
$$\mathcal{R} \subseteq \mathcal{O}$$

This is illustrated in figure 1 below. All the nodes in the graph $\mathcal{S}$ represent the whole system: those nodes outside any subset boundary are in the outside $\mathcal{O}$ of the component shown in the diagram.. The boundary $\mathcal{B}$ (the left subset) provide the interface to the encapsulated component: they can be accessed from anywhere — outside the component, inside it, from other boundary nodes, or from the component's external references. The centre subset in the diagram represent the inside $\mathcal{I}$ of the component: nodes here may only be accessed by each other or by the boundary nodes. The right-most subset are the outside nodes referred to by the inside and boundary of the component — its external references $\mathcal{R}$.

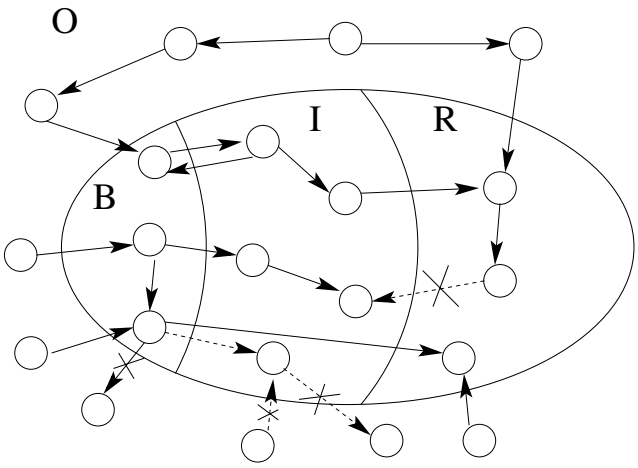

Fig. 1. An encapsulated component

Note that we make a distinction between components ($\mathcal{C}$) and nodes ($\mathcal{N}$): components are not nodes: rather a component *encapsulates* several nodes (via its encapsulation function $\mathbf{e}$) by identifying an encapsulation boundary, the nodes inside that boundary, and any external references leaving that boundary. In an object graph, objects are nodes, however components may be based upon a range of structures, including objects but also packages, classes, and universes, as we will see below.

The argument to the encapsulation function — a component identifier — effectively chooses a particular partition of the system. For that component, the function selects those nodes that will be on the boundary, inside, external, and outside. The interpretation of the component identifier depends upon the particular scheme being modelled, and is typically linked to the access graph proper via node labels. For example, for package-based encapsulation (such as confined types, see section 2.6 below) the unit of encapsulation is a package, so component identifiers model packages, and nodes need a labelling function (say $\mathit{package} : \mathcal{N} \rightarrow \mathcal{C}$) to identify the package to which they belong. An encapsulation function for such a scheme takes an identifier (such as "`java.lang.util`") as its argument to select the package that is encapsulated: the function itself will then choose objects based on their package (e.g. grouping objects of classes in `java.lang.util`). By contrast, in object-based encapsulation schemes such as islands or balloons, each node forms an encapsulated component, so we let $\mathcal{C} = \mathcal{N}$.

1.3 Encapsulation Constraints

So far, we have an access graph as a (directed) graph; and an encapsulation function which selects nodes from that graph. The third part of our model is the *encapsulation constraints* that link the access graph and encapsulation function. These constraints ensure that the encapsulation function does actually describe encapsulation: that inside nodes are only accessed via the boundary, and that outside nodes are only accessed via the external references.

More formally, the encapsulation constraints are as follows: first, a component's boundary, inside, and externals must be disjoint (this implies that the externals must be outside the component).

$$\mathcal{B} \cap \mathcal{I} = \mathcal{B} \cap \mathcal{R} = \mathcal{I} \cap \mathcal{R} = \{\,\}$$

The second constraint is the most important one: the boundary must actually be a boundary for the inside. To be a valid encapsuation the nodes inside an encapsulated component can only be accessed via the component's boundary: that is, any edges ending at inside nodes can only come from other inside nodes or boundary nodes:

$$\triangleright\mathcal{I} \subseteq (\mathcal{B} \cup \mathcal{I})$$

Alternatively we could state that all paths from the outside of a component into the inside must pass through the boundary.

$$\forall o \in \mathcal{O} : \forall i \in \mathcal{I} : \forall p \in \mathit{paths}(o, i) : \exists b \in p \text{ s.t. } b \in \mathcal{B}$$

where *paths*(i, o) is all paths (as a set of sequences of nodes) from i to o.

Finally we require that every outside node directly accessed by an inside node is recorded in the component's externals:

$$(\mathcal{B} \cup \mathcal{I})\triangleright \subseteq (\mathcal{B} \cup \mathcal{I} \cup \mathcal{R})$$

Note that this sets a lower bound on the extent of the external references — a component may include outside objects in its external references even if it does not refer to them. This is useful because it allows us to set $\mathcal{R} = \mathcal{O}$ to indicate that an object's external references are unconstrained (note that the outside ($\mathcal{O}$) is all nodes in the system ($\mathcal{S}$) except the boundary and interface — $\mathcal{O} = \mathcal{S} - (\mathcal{B} \cup \mathcal{I})$). In any event, one can always define an alternative encapsulation function with a tighter external reference set.

This formal definition is designed to capture the intent of Dan Ingalls' statement above:

> *No component in a complex system should depend on the internal details of any other component.* [14]

In our model, the "*external aspects*" of an encapsulated component lie on its boundary, while the *"internal details"* are its inside. The encapsulation constraints ensure that these internal details of a component are hidden from every external component in the system.

1.4 Nested Encapsulation

These definitions imply that encapsulated components may be nested inside each other. We say that c_1 contains c_2 ($c_1 \leq c_2$) or c_2 is inside c_1 ($c_2 \geq c_1$) if all c_2's boundary and inside are part of the boundary and inside of c_1 (see figure 2). That is $\leq$ is defined as:

$$c_1 \leq c_2 \textbf{ iff } \mathcal{B}_{c_2} \subseteq (\mathcal{B}_{c_1} \cup \mathcal{I}_{c_1}) \wedge \mathcal{I}_{c_2} \subseteq (\mathcal{I}_{c_1})$$

One consequence of this definition is that the external references of the inner component must be contained within the inside, boundary, or external references of the outer component:

$$\mathcal{R}_{c_2} \subseteq (\mathcal{B}_{c_1} \cup \mathcal{I}_{c_1} \cup \mathcal{R}_{c_1})$$

2 Encapsulation Schemes over Object Graphs

Having outlined our model of encapsulation, in this section we apply that model to a range of encapsulation schemes over object graphs, beginning with simple implicit models and progressing to more complex schemes. We describe each scheme by its characteristic encapsulation function.

An object graph is part of a program's operational state: the graph will evolve as the program runs, as objects are created and deleted and as assignments change references between nodes. This is the reason that we characterise encapsulation *schemes*

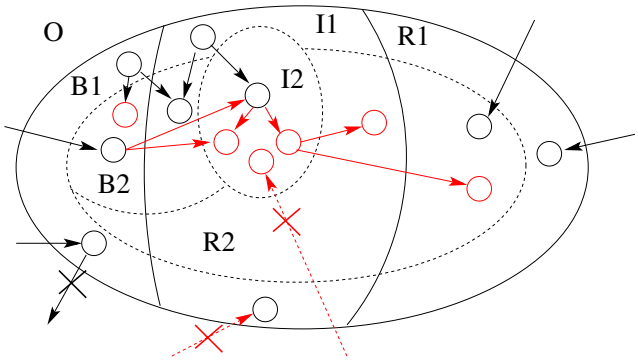

Fig. 2. Nested Components

by encapsulation *functions*: these functions are implicitly parameterised by the object graphs to which they are applied. Typically, an encapsulation scheme will mandate that the encapsulation constraints hold at all times in all runs of all programs to which the scheme applies. Different schemes will use a wide range of *mechanisms* to achieve this, from dynamic checks at assignment or invocation, through static annotations, extended type systems, abstract interpretation, theorem proving, or even providing no enforcement at all and simply relying on programmers' adherence to conventions. One key advantage of our model of encapsulation functions (and access graphs) is that they can abstract away considerations of mechanism and provide succinct characterisations of the encapsulation policies supported by each scheme — in particular, the topology of encapsulation that each scheme provides.

2.1 Islands: Full Encapsulation

Hogg's Islands [13] was the first alias protection scheme advocated for an object-oriented language. The key notion of an Island is that some classes in the program are distinguished as *bridge* classes: instances of those classes are subject to a series of syntactic restrictions, so that all objects reachable from the bridge — i.e. the objects making up the island — are *only* reachable from the bridge (see figure 3).

In terms of an encapsulation function, an object o that has been identified as an bridge (using the label $bridge : \mathcal{N} \rightarrow \mathbb{B}$) has itself as a boundary, all objects reachable via that boundary as its inside, and no external references:

$$\mathbf{e}_{island}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \\ \mathcal{I} = \{o\}\triangleright\!\!\triangleright \\ \mathcal{R} = \{\,\} \end{pmatrix} \quad \text{where } bridge(o)$$

Islands are one example of an alias protection scheme with a *full encapsulation* topology: the encapsulation functions of such schemes are characterised by having empty

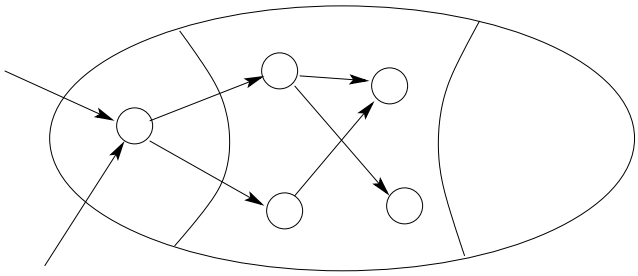

Fig. 3. Full Encapsulation

external references. Almeida's Balloon Types [2] also mandate full encapsulation, as does Banerjee and Naumann's original work on heap confinement [3].

Note that Island's encapsulation function must also be interpreted to cover only static references (from objects' fields and global variables) but not dynamic references (such as local variables or method arguments on the stack). Islands provides no protection for dynamic references.

2.2 Uniqueness

Islands also supports another common type of encapsulation, uniqueness. A classically unique object has only one incoming reference ($|\rhd u| = 1$) and so a unique object is always encapsulated by the sole object that refers to it. This condition produces a range of encapsulation function.

Most generally, a unique object may refer to any other object in the system (see figure 4):

$$\mathsf{e}_{\mathit{unique}}(u) = \begin{pmatrix} \mathcal{B} = \rhd\{u\} \\ \mathcal{I} = \{u\} \\ \mathcal{R} = \mathcal{O} \end{pmatrix} \quad \text{where } |\rhd\{u\}| = 1$$

where we say that the external references are the outside ($\mathcal{O}$) of the unique object to mean that the references are unconstrained (not that the object necessarily refers to every outside object!). Note also that this definitions works by placing the unique object into the inside of the encapsulation: the encapsulation boundary is the (sole) object which refers to this object.

Clarke and Wrigstrad have recently demonstrated that a weaker formulation of uniqueness can provide all the benefits of full uniqueness but is significantly more flexible. An *externally unique* object [7, 8] may have only one reference from outside, but may have any number of other references from inside:

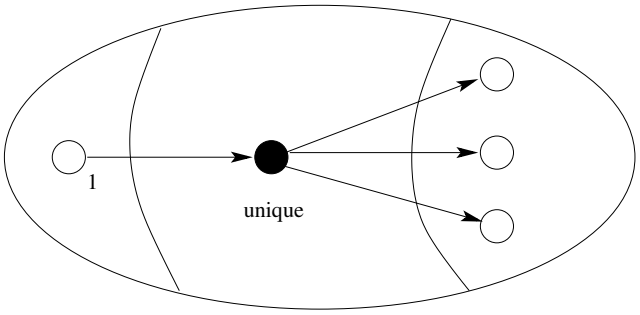

Fig. 4. A Unique Object

$$\mathbf{e}_{\textit{external-unique}}(u) = \begin{pmatrix} \mathcal{B} = \{v\} \\ \mathcal{I} = \{u\} \cup \{o | u \sqsubseteq o\} \\ \mathcal{R} = \mathcal{O} \end{pmatrix}$$
$$\text{where } v = (\triangleright\{u\} - \mathcal{I}) \text{ and } |v| = 1$$

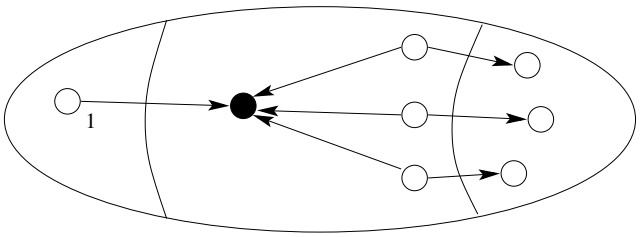

Fig. 5. External Uniqueness

Again, the encapsulation boundary is the object that refers to the externally unique object. Here, however, an unique object contains a number of other objects but any incoming references from these objects do *not* count against the uniqueness of the externally unique object — external uniqueness counts only those references from the boundary in to the unique object, not references from the inside of the object.

2.3 Balloon Types

Alemida's Balloon Types [2] provide full encapsulation, however, they are also constrained to be unique. Using a label ($\textit{balloon} : \mathcal{N} \rightarrow \mathbb{B}$) for balloon objects, we can model this with two nested encapsulations, a inner full encapsulation

$$\mathbf{e}_{\textit{inner-balloon}}(b) = \begin{pmatrix} \mathcal{B} = \{b\} \\ \mathcal{I} = \{b\}\triangleright\!\!\triangleright \\ \mathcal{R} = \{\,\} \end{pmatrix} \quad \text{where } \textit{balloon}(b)$$

and an outer unique object:

$$\mathbf{e}_{\textit{outer-balloon}}(b) = \begin{pmatrix} \mathcal{B} = \triangleright\{b\} \\ \mathcal{I} = \{b\} \\ \mathcal{R} = \{b\}\triangleright\!\!\triangleright \end{pmatrix} \quad \text{where } \textit{balloon}(b) \wedge |\triangleright\{b\}| = 1$$

or as a single encapsulation function encompassing both the balloon object and the contents of the balloon, and having as its boundary the (sharable) object holding the single reference to balloon object proper:

$$\mathbf{e}_{\textit{balloon}}(b) = \begin{pmatrix} \mathcal{B} = \triangleright\{b\} \\ \mathcal{I} = \{b\} \cup \{b\}\triangleright\!\!\triangleright \\ \mathcal{R} = \{\,\} \end{pmatrix} \quad \text{where } \textit{balloon}(b) \wedge |\triangleright\{b\}| = 1$$

These are then nested, so that $\mathbf{e}_{\textit{balloon}} \leq \mathbf{e}_{\textit{inner-balloon}}$ and $\mathbf{e}_{\textit{balloon}} \leq \mathbf{e}_{\textit{outer-balloon}}$.

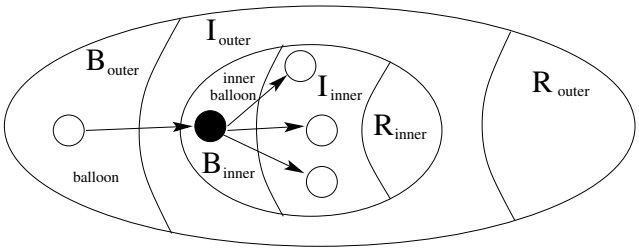

Fig. 6. Balloon Types

2.4 Flexible Alias Protection

Flexible Alias Protection [16] was proposed to resolve problems with full static alias protection schemes such as Islands and Balloons: these schemes were simultaneously too lax, in that dynamic references could penetrate Islands and plain balloons, and too strict, as no external references were permitted. Flexible Alias Protection divided the objects within an "alias-protected container" into two main categories: *representation*

objects which were private, and the *arguments* objects which could be shared. Flexible alias protection used "modes" annotating static types to track which objects were representation and which arguments: we can model this here with a pair of labelling functions ($rep, arg : \mathcal{N} \to \mathbb{P}\mathcal{N}$). Unlike Islands, Flexible Alias Protection maintained the same encapsulation on dynamic references as static references — representation objects could never be accessed externally.

To a first approximation, the encapsulation function for Flexible Alias Protection is:

$$\mathbf{e}_{flexible}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \\ \mathcal{I} = rep(o) \\ \mathcal{R} = arg(o) \end{pmatrix}$$

with an object's representation being its inside, and arguments its external references (see figure 7 compare with 3). Flexible Alias Protection's restrictions on the use of modes (types of different mode are not assignment compatible; representation modes cannot appear in a protected container's interface, and so on) ensure that the labelling actually resulted in an encapsulation.

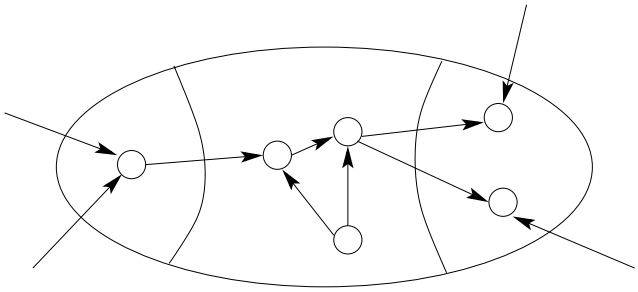

Fig. 7. Flexible Encapsulation Topology

2.5 Ownership Types

Ownership Types [9] were first designed to formalise the topological restrictions of Flexible Alias Protection, although they have since found many difference applications. The key idea behind ownership types is that a type system statically enforces topological restrictions based on an ownership relation between objects: we can model this relationship by labelling the basic object graph so that every node has an owner ($owner : \mathcal{N} \to \mathcal{N}$) which is another node. We write i **ownedby** o for the transitive closure of the *owner* function, and also o **owns** i for the inverse. The ownership relation is constrained to form a convergent tree at some nominated root ($owner(root) = root$) and the type system then enforces a containment invariant:

$$s \longrightarrow t \Rightarrow s \textbf{ ownedby } \mathit{owner}(t)$$

that is, the source of an edge must be owned by the owner of the target of an edge. This containment invariant ensures that ownership partitions the graph into a series of nested encapsulations based on each object owning its inside (see figure 8).

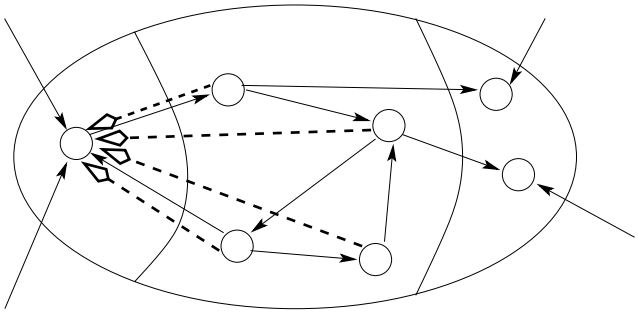

Fig. 8. Ownership Types

Using this *ownership graph* as the access graph, the most basic encapsulation function for ownership types have the form of:

$$\mathbf{e}_{\mathit{simple\text{-}ownership}}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \\ \mathcal{I} = \{i | o \textbf{ owns } i\} \\ \mathcal{R} = \{r | o \textbf{ ownedby } \mathit{owner}(r)\} \end{pmatrix}$$

The external reference expression for ownership types can be simplified if (the types of) objects are explicitly parameterised with the ownership of the objects to which they may refer ($\mathit{params} : \mathcal{N} -> \mathbb{P}\mathcal{N}$):

$$\mathbf{e}_{\mathit{param\text{-}ownership}}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \\ \mathcal{I} = \{i | o \textbf{ owns } i\} \\ \mathcal{R} = \{r | \mathit{owner}(r) \in \mathit{params}(o)\} \end{pmatrix}$$
$$\text{where } \forall p \in \mathit{params}(o) : o \textbf{ ownedby } p$$

These ownership type systems are known as *owners-as-dominators* models, because the containment invariant ensures that every object is dominated by their sole owner ($\mathit{owner}(o) \sqsubseteq o$). The encapsulation function shows how this forms an encapsulation, furthermore, the tree of owners gives a matching tree of nested encapsulations:

$$o_1 \textbf{ owns } o_2 \Rightarrow \mathbf{e}(o_1) \leq \mathbf{e}(o_2)$$

Because the owners-as-dominators versions of ownership types limits a component's boundary to be a single object, these simple ownership types restrict some programming idioms. Recently, Boyapati and Liskov have extended ownership types to allow multiple objects in a component's boundary, where the extra objects are Java (or BETA) style inner objects nested within the main object [6]. This changes the encapsulation function as follows:

$$\mathbf{e}_{\textit{inner-ownership}}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \cup \{b | \textit{outer}(b) = o\} \\ \mathcal{I} = \{i | o \textbf{ owns } i\} \\ \mathcal{R} = \{r | o \textbf{ ownedby } \textit{owner}(r)\} \end{pmatrix}$$

where $\textit{outer} : \mathcal{N} \rightarrow \mathcal{N}$ gives the outer object within which a (non-static) inner class instance is nested (see figure 9). Banerjee and Naumann's later work on heap confinement also supports this topology [4].

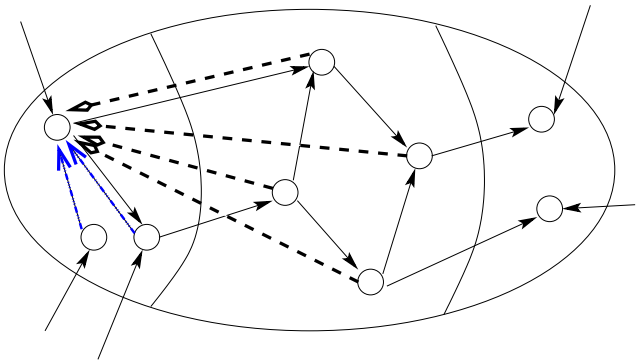

Fig. 9. Extended Ownership Types

2.6 Confined Types

There are a range of other type-based encapsulation schemes. Confined types [5, 11] are a package-wide type discipline: some classes are marked as confined, and instances of these classes can only be accessed by instances from the same package. Objects that are not confined within the package provide the interface by which their confined types are manipulated (figure 10). We model this by decorating every object with the package to which their class belongs ($\textit{package} : \mathcal{N} \rightarrow \mathcal{C}$) and a bit to indicate whether or not that class is confined[2] ($\textit{confined} : \mathcal{N} \rightarrow \mathbb{B}$):

[2] We could add another layer of indirection here to treat classes explicitly, but this would needlessly complicate the model.

$$\mathbf{e}_{\mathit{confined}}(p) = \begin{pmatrix} \mathcal{B} = \{o | \mathit{package}(o) = p \wedge \neg \mathit{confined}(o)\} \\ \mathcal{I} = \{o | \mathit{package}(o) = p \wedge \mathit{confined}(o)\} \\ \mathcal{R} = \mathcal{O} \end{pmatrix}$$

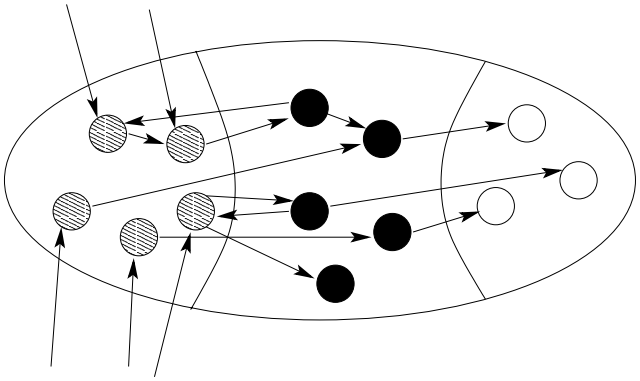

Fig. 10. Confined Types

2.7 Universes

Universes [15] are an alternative to ownership types, based on read-only references rather than ownership parameterisation. In particular, objects are encapsulated only with respect to read-write references: read-only references are unencapsulated (see figure 11). A universe plays a similar role to an owner in ownership types (every object belongs to a universe — $\mathit{universe} : \mathcal{N} \rightarrow \mathcal{C}$), however as well as encapsulation based on objects (called object universes), universes also supports wider encapsulation (called type universes).

Considering only read-write references, then, every object has one object universe ($\mathit{objectu} : \mathcal{N} \rightarrow \mathcal{C}$, one-to-one) but may be associated with a set of type universes ($\mathit{typeu} : \mathcal{N} \rightarrow \mathbb{P}\mathcal{C}$). An object that uses only its own object universe encapsulates all the objects in that universe:

$$\mathbf{e}_{\mathit{object\text{-}universe}}(u) = \begin{pmatrix} \mathcal{B} = \{o | \mathit{objectu}(o) = u\} \\ \mathcal{I} = \{i | \mathit{universe}(i) = u\} \\ \mathcal{R} = \mathcal{O} \end{pmatrix}$$

where u is the unique object universe of o

as shown in figure 11. Alternatively, we can consider the encapsulation afforded solely by a type universe:

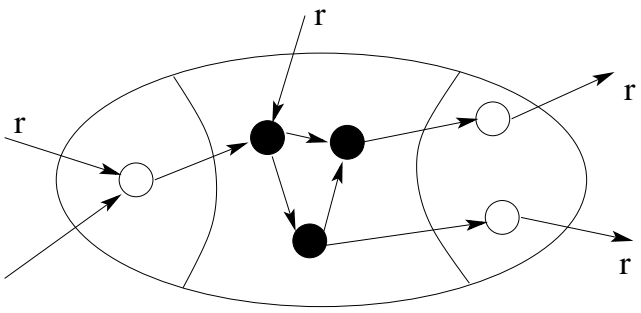

Fig. 11. Universes

$$\mathsf{e}_{\textit{type-universe}}(u) = \begin{pmatrix} \mathcal{B} = \{o | \textit{typeu}(o) \ni u\} \\ \mathcal{I} = \{i | \textit{universe}(i) = u\} \\ \mathcal{R} = \mathcal{O} \end{pmatrix}$$
where u is a type universe

such a component's boundary contains all the objects which share that type universe. Note that both these encapsulation functions do not restrict the outgoing references in any way, and may apply to both read-only and read-write references.

Determining encapsulation based on objects (rather than based on universes) is rather more difficult. There are two issues here. First, object universes must be transitive (rather like ownership); because an object owns all the objects in its object universe, it also effectively owns all objects in those objects' object universes, and so on (again, we can write "o **owns** i" mean i is (transitively) in o's object universe). On the other hand, any of these objects may use a type universe, meaning they may access any object transitively owned by that universe, but they must share these objects with other objects that also use the same type universe.

The resulting encapsulation function has the advantage that is offers full encapsulation, that is, in this case, that there are no outgoing read-write references.

$$\mathsf{e}_{\textit{universes}}(o) = \begin{pmatrix} \mathcal{B} = \{o\} \cup \{b | \textit{typeu}(b) \cap T \neq \{\,\}\} \\ \mathcal{I} = \{i | o \textbf{ owns } i\} \cup \{t | \textit{universe}(t) \in T\} \\ \mathcal{R} = \{\,\} \end{pmatrix}$$
where T is the set of all type universes transitively reachable from o

3 Conclusion

In this position paper we have outlined our generic model of encapsulation, and applied that to a number of alias protection schemes over object graphs. In the future, we plan to apply this model to more object-graph schemes [1, 17]; to model finer grained access graphs for programming languages; and to encapsulation in non-object oriented languages and in computer systems more generally.

References

1. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA Proceedings*, November 2002.
2. Paulo Sérgio Almeida. Balloon Types: Controlling sharing of state in data types. In *ECOOP Proceedings*, June 1997.
3. Anindya Banerjee and Dave Naumann. Representation independence, confinement, and access control. In *Proceedings of ACM Principles of Programming Languages (POPL)*, pages 166–177, 2002.
4. Anindya Banerjee and Dave Naumann. Ownership: transfer, sharing, and encapsulation. In Dave Clarke, Sophia Drossopoulou, and James Noble, editors, *Proceedings of IWAOOS'03: The ECOOP 2003 International Workshop on Aliasing, Confinement, and Ownership*, July 2003.
5. Boris Bokowski and Jan Vitek. Confined types. In *OOPSLA Proceedings*, 1999.
6. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2003.
7. Dave Clarke and Tobias Wrigstrad. External uniqueness. In *Foundations of Object-Oriented Languages (FOOL)*, 2003.
8. Dave Clarke and Tobias Wrigstrad. External uniqueness is unique enough. In *ECOOP Proceedings*, 2003.
9. David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA Proceedings*, 1998.
10. Peter Grogono and Mark Gargul. A graph model for object oriented programming. *SIGPLAN Notices*, pages 21–28, July 1994.
11. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *OOPSLA Proceedings*, 2001.
12. C.A.R. Hoare and He Jifeng. A trace model of pointers and objects. In *ECOOP Proceedings*, 1999.
13. John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA Proceedings*, November 1991.
14. Dan H. H. Ingalls. Design principles behind Smalltalk. *BYTE*, pages 286–298, August 1981.
15. P. Müller and A. Poetzsch-Heffter. A type system for controlling representation exposure in Java. In *ECOOP Workshop on Formal Techniques for Java Programs*, number TR 269 in Technical Report. Fernuniversität Hagen, 2000.
16. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP Proceedings*, 1998.
17. Alan Cameron Wills. *Formal Methods applied to Object-Oriented Programming*. PhD thesis, University of Manchester, 1992.
18. Tian Zhao, Jens Palsberg, and Jan Vitek. Lightweight confinement for featherweight java. In *OOPSLA Proceedings*, November 2003. A preliminary version appeared in the Proceedings of IWAOOS'03: The ECOOP 2003 International Workshop on Aliasing, Confinement, and Ownership.
19. Thomas Zimmermann and Andreas Zeller. Visualizing Memory Graphs. In *Proceedings of Dagstuhl Seminar on Software Visualization, 2001*, pages 191–204. Springer-Verlag, May 2001.

Lightweight Confinement for Featherweight Java

Tian Zhao Jens Palsberg† Jan Vitek†

University of Wisconsin, † Purdue University

Abstract. Confinement properties impose a structure on object graphs which can be used to enforce encapsulation – which is essential to certain program optimizations, to modular reasoning, and in many cases to software assurance. This paper formalizes the notion of *confined type* in the context of Featherweight Java. A static type system that mirrors the informal rules of [16] is proposed and proven sound. The definition of confined types is extended to confined instantiation of generic classes. This allows for confined collection types in Java and for classes that can be confined *post hoc*. Confinement type rules are given for Generic Featherweight Java, and proven sound.

1 Introduction

While object-oriented languages provide syntactic support for encapsulating fields of object structures, this form of encapsulation is only skin deep. Data hiding is only an illusion, one need only take a step back and look at entire object graphs to realize it. The graph induced by the reference relationship between objects during program execution owes more to a much maligned Italian national dish than to anything else. The problem is the lack of constraints on patterns of references. While the access modifiers provided by the language can prevent some part of the system from reading a variable, there is no syntax for constraining the spread of the objects they denote.

The last five years (starting with [7]) have seen a renewed interest in the study of ownership structures as well as related work on aliasing. The goal of most of these works is ambitious: extend existing language with support for object encapsulation. Some of the most promising results suggest that it may be possible to capture these properties through advanced type systems. The work of Clarke *et al.* was the first to propose a Java-like language with ownership types [6]. More recently, Boyapati *et al.* [2, 3] have extended these results and applied to a subset of the Real-Time Specification for Java.

This work takes a slightly different view, and rather than going for the most expressive notion of confinement, we look for the *least disruptive* one. The reason for this apparently counterintuitive choice is that before settling on one particular notion of confinement and writing a compiler for a new language, it is necessary to get first hand experience with the benefits and costs of developing large software with these new constructs. Even more than type systems, restrictions on references will impose constraints on programs. The risk is that

they become too cumbersome for programmers without providing sufficient benefit. In [16] Bokowski and Vitek proposed a lightweight notion of encapsulation for Java called *confined types*. The idea is simple: use Java's notion of software module (packages) as an encapsulation boundary. A class is termed *confined* if references to instances of the class may not leak out of the class' defining package. In other words, a confined object can only be stored in fields of objects defined in the same package and manipulated by code of classes belonging to its package. What makes this approach lightweight is that very few annotations are required (one annotation per confined class, and some extra annotation for inherited method) and that conformance to the confinement rules can be checked modularly. In later work Grothoff, Palsberg and Vitek [10] designed a tool for inferring confinement without any annotations, the result of a thorough analysis of a large corpus of code (101,893 classes) is that confined classes occur naturally in large systems[1]. Confinement has been shown to be applicable in practice as evidenced by the work of Clarke *et al.* on Enterprise Java Beans [5].
A longer version of this paper including proofs, support for generics and a survey of related work is available in technical report form.

2 Confined Types

In object-oriented programming languages such as Java or C#, confinement can be achieved by a disciplined use of built-in static access control mechanisms combined with some simple coding idioms. The goal of confinement, as stated in [10], is to enforce the following informal soundness property:

> *An object of confined type is encapsulated in its defining module.*

To this end object types are categorized into *public types* and *confined types*. The idea is that modules are composed of two distinct software layers: an interface composed of public classes and interfaces and a core consisting of confined classes. What confinement adds to the visibility rules provided by the language is the guarantee that subtyping can not be used to 'leak' reference to core classes. Furthermore confinement annotation make the programmer's intent explicit and allow for automated checking. In the Java current realization the unit of modularity is the package.
Consider the following example. A class `Bucket` is used to implementation of a hash table class, `mine.HTable`. A fragment of the code of both classes is displayed in Figure 1. Hash table buckets are a fine example of internal data structures which should not escape the context of the enclosing class. In Java, the first step towards that goal is to declare class `Bucket` package scoped, thus ensuring that its visibility is restricted to the package of the `HTable` class, in this case `mine`.
But what if one of `HTable`'s public methods, say `get()`, were to return a `Bucket` instance or, just as bad, store a reference to a bucket in a public field? One can

[1] All software and benchmarks referred to in this paper are freely available from `www.ovmj.org`.

view this as an escape problem: can references to instances of a package-scoped class escape their enclosing package? If the answer to this question is no, then the objects of such a class are said to be *encapsulated*. This form of encapsulation is what confined types ensure. We emphasize that a stronger notion of confinement could provide even tighter bounds on reference patterns, for instance ensure that entries of one hash table are not mixed up with entries of another [6].
The pleasant characteristics of confinement is that it can be viewed as a programming discipline, *i.e.* it is possible to list a small (and simple!) set of software design rules that will result in a confined type. Moreover, these rules are local to the confining package, which implies that confined types can be used when appropriate and ignored when unnecessary.
In order to allow reuse of code by inheritance, we add a small number of rules to enforce a second property which we call `anonymity`. The need for anonimity arises as soon as we consider the behavior of the methods inherited by a confined type. Some of these methods may have been written *outside* of the package, yet they access a confined object through the distinguished `this` pointer. There are two design choices either disallow inheritance or restrict it to a safe subset. Anonymous method are well-behaved in the sense that they do not 'leak' the `this` pointer.
Confinement can be enforced (or inferred) by two sets of constraints. The first set of constraints, *confinement rules*, apply to the enclosing package, the package in which the confined class is defined. These rules track values of confined types and ensure that they are neither exposed in public members, nor widened to non-confined types. The second set of constraints, so-called *anonymity rules*, applies to methods inherited by the confined classes, potentially including library code,

```
package mine;

public class HTable {
   private Bucket[] buckets;
   public Object get(Object key) { ...} }

class Bucket implements marker.ConfinedClass {
   Bucket next;
   Object key, val; }


package marker;

interface ConfinedClass {}

final public class AnonymousMethod extends Error {}
```

Fig. 1. Example. The class Bucket is confined in its enclosing package. Confined classes are marked using a simple idiom based on marker types.

and ensures that these methods do not leak a reference to the distinguished variable `this` which may refer to an object of confined type. Enforcing confinement implies tracking the spread of confined objects within a package and preventing them from crossing package boundaries. A confinement breach occur if an instance of a confined type escapes from its package. Since confinement is couched in terms of object types, widening a value from a confined type to a non-confined type presents a risk as it is not possible to ascertain what will happen with the object after the cast, thus casts are a considered to be confinement violations. We present the rules of [10] with some of implementation specific details omitted.

Anonymity Rules. Anonymity rules apply to inherited methods which may reside in classes outside of the enclosing package. These rules prevent a method from leaking the `this` reference. A method is *anonymous* if the following rules hold.

$\mathcal{A}1$	An anonymous method cannot widen `this` to a non-confined type.
$\mathcal{A}2$	Methods invoked on `this` must be anonymous.

The first rule prevents an inherited method from storing or returning `this` unless the static type of `this` also happens to be confined. Rule $\mathcal{A}1$ detects a direct anonymity violations, and $\mathcal{A}2$ tracks transitive violations.

Confinement Rules. These rules apply to all classes of a package containing confined types.

$\mathcal{C}1$	All methods invoked on a confined type must be anonymous.
$\mathcal{C}2$	A confined type cannot be public.
$\mathcal{C}3$	A confined type cannot appear in the type of a public (or protected) field or the return type of a public (or protected) method of a non-confined type.
$\mathcal{C}4$	Subtypes of a confined type must be confined.
$\mathcal{C}5$	A confined type cannot be widened to a non-confined type.

Rule $\mathcal{C}1$ ensures that no inherited method invoked on a confined type will leak the `this` pointer. Rule $\mathcal{C}2$ states that public classes can not be confined. Rule $\mathcal{C}3$ prevents exposure of confined types in the public interface of the package as client code could break confinement by casting them to non-confined types. Rule $\mathcal{C}4$ prevents non-confined classes (or interfaces) from extending confined types. Finally, rule $\mathcal{C}5$ prevents values of confined type from being cast to non-confined types.

Modular enforcement. Confinement annotations can trivially be checked as part of the source level type checking or by bytecode verification. One of the key design requirements for Confined Types was that code that did not use them should not have to be checked and should not be required to use a modified compiler. Confined Types meet this requirement as the rules outlined above place no constraints on clients of a confined package (rule $\mathcal{C}3$ is crucial in this respect). Likewise confined type inference can be performed on a per-package basis. Inference of anonymous methods is somewhat trickier as it requires analyzing parent classes.

3 Confined Featherweight Java

Featherweight Java, or FJ, is core object calculus that was developed by Igarashi, Pierce and Wadler [13] for modeling Java's type system. The calculus is minimal in that it has only five forms of expression: object creation, method invocation, field access, casting and variable access. It is in this spartan setting that Igarashi *et al.* studied a number aspects of Java, notably generics and inner classes.
Featherweight Java is a good vehicle for semantic investigations due to its economy of principles. Thus to stay true to its designer's spirit, *Confined Featherweight Java*, or ConfinedFJ, has been modified as little as possible. In particular, we have resisted to the temptation to add assignment. The syntax appears on Figure 2, the only departures from FJ are the addition of package names in class declarations and of an access modifier on classes. `+` means the class is public and `-` means that the class is confined – in Java this also correspond to the default access modifier. We follow the Smalltalk-80 convention and assume that fields are package scoped by default.
Consider the following ConfinedFJ program in which a public class `Table`, in `my` package, contains a linked list of `Bucket` objects. The `Bucket` class also belongs to `my` package but is declared confined. Yet, the program is invalid as the body of the `get` method of `Table` returns its instance of `Bucket`. This is a direct violation of the widening rule.

```
+ class my.Table extends Object {
    Bucket buck;
    Table(Bucket buck) {
      super(); this.buck = buck; }
    Object get() { return this.buck; } }

- class my.Bucket extends Object {
    Bucket() { super(); } }
```

The breach can be exhibited by constructing a class, aptly named `Breach`, in the `outside` package.

```
+ class outside.Breach extends Object {
    Object something;
```

```
Breach(Object something) {
   super(); this.something = something; } }
```

Finally, in the context of these definitions, the expression:

```
new Breach( new Table( new Bucket() ).get() );
```

evaluates to

```
new Breach( new Bucket() );
```

While the original expression is a valid object structure, the result isn't. In the original, `Breach` refers to `Table` which, in turn, refers to `Bucket` enforces the desired isolation property. Namely, the confined type is only referenced by another class of the same package. In the resulting expression the confined bucket is directly held by the `Breach` object.
In another prototypical breach of confinement consider the following situation in which the confined class `Self` extends a `Broken` class which resides `outside`. Assume further that the table inherits its parent's code for the `reveal` method.

```
- class my.Self extends outside.Broken {
   Self() { super(); } }

+ class my.Ifc extends Object {
   Ifc() { super(); }
   Object get() {
      return new Self().reveal(); } }
```

Inspection of this code does not reveal any breach of confinement. But if we widen the scope of our analysis a little and look at the `Broken` class, then we may see:

```
+ class outside.Broken extends Object {
   Broken() { super(); }
   Object reveal() { return this; } }
```

The invoking `reveal` on an instance of `Self` will return a reference to the object itself (suitably widened to `Object`).
We now proceed to introduce the syntax and semantics of ConfinedFJ, before presenting a type system which will declare both of the above programs to be ill-typed.

3.1 Syntax

The syntax is shown in Figure 2. We use metavariables `L` to range over class declarations, `A, B, C, D, E` to range over classes, `K, M` to range over constructors and methods, and `f` and `x` to range over fields and parameters to methods. We use over-bar to represent a finite array, for instance, $\overline{\texttt{f}}$ represents $\texttt{f}_1, \texttt{f}_2, \ldots, \texttt{f}_\texttt{n}$. We use `P, Q` to range over package names and we use $\rho_\texttt{C}$ to refer to the package that the class `C` is defined in. We assume a class table CT which stores the definitions of all classes and $CT(\texttt{C})$ is the definition of class `C`.

Syntax:

$$\circ ::= \texttt{+} \mid \texttt{-}$$

$$\texttt{L} ::= \circ\ \texttt{class P.C} \lhd \texttt{D}\ \{\ \overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\ \}$$

$$\texttt{K} ::= \texttt{C}(\overline{\texttt{C}}\ \overline{\texttt{f}})\ \{\ \texttt{super}(\overline{\texttt{f}});\ \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\ \}$$

$$\texttt{M} ::= \texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\ \{\ \texttt{return e};\ \}$$

$$\texttt{e} ::= \texttt{x} \mid \texttt{this} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{(C) e} \mid \texttt{new C}(\overline{\texttt{e}})$$

$$\texttt{v} ::= \texttt{new C}(\overline{\texttt{v}})$$

Subtyping:

$$\texttt{C} <: \texttt{C}$$

$$\frac{\texttt{C} <: \texttt{D} \quad \texttt{D} <: \texttt{E}}{\texttt{C} <: \texttt{E}}$$

$$\frac{CT(\texttt{C}) = \circ\ \texttt{class P.C} \lhd \texttt{D}\ \{\ \ldots\ \}}{\texttt{C} <: \texttt{D}}$$

Computation:

$$\frac{\mathit{fields}(\texttt{C}_0) = (\overline{\texttt{C}}\ \overline{\texttt{f}})}{\texttt{new C}_0(\overline{\texttt{v}})\texttt{.f}_\texttt{i} \ \rightarrow\ \texttt{v}_\texttt{i}} \qquad \text{(R-Field)}$$

$$\frac{\texttt{C} <: \texttt{D}}{\texttt{(D) new C}(\overline{\texttt{v}}) \ \rightarrow\ \texttt{new C}(\overline{\texttt{v}})} \qquad \text{(R-Cast)}$$

$$\frac{\mathit{mbody}(\texttt{m},\ \texttt{C}_0) = (\overline{\texttt{x}},\ \texttt{e})}{\texttt{new C}_0(\overline{\texttt{u}})\texttt{.m}(\overline{\texttt{v}}) \ \rightarrow\ [\overline{\texttt{v}}/_{\overline{\texttt{x}}},\ {}^{\texttt{new C}_0(\overline{\texttt{u}})}/_{\texttt{this}}]\texttt{e}} \qquad \text{(R-Invk)}$$

We use $\texttt{e}, \texttt{d}$ to range over expressions and $\texttt{u}, \texttt{v}, \texttt{w}$ to range over *fully-evaluated* objects of the form $\texttt{new C}(\overline{\texttt{v}})$. An *expression* e can be either a variable x, the this pseudo variable, a field access e.f, a method invocation $\texttt{e.m}(\overline{\texttt{e}})$, a cast (C)e, an object $\texttt{new C}(\overline{\texttt{e}})$.
The subtyping relation $\texttt{C} <: \texttt{D}$ denotes that class C is a subtype of class D, irrespective of access mode and or package, if class C transitively extends D.
The definitions and auxiliary functions used in our semantics and typing rules are listed in Figures 2 and 3. The predicate $\mathit{visible}(\texttt{C}, \texttt{D})$ holds if the type C is *visible* in the package of D. That is, if C and D are not in the same package, then C must be declared public for it to be visible from D. This definition models the Java's class access modifiers so that only *public* classes can be referred to from other packages.
The partial order $\preceq$ on types is such that $\texttt{C} \preceq \texttt{D}$ holds iff C is a subtype of D, and either C is public or D is confined. This definition will be used in the typing rules to prevent *reference widening*, where the reference to an object of confined type should not be widened to public types.

Other definitions:

$$\frac{CT(\texttt{C}) = \texttt{- class P.C} \lhd \texttt{D}\ \{\ \ldots\ \}}{conf(\texttt{C})}$$

$$\frac{CT(\texttt{C}) = \texttt{+ class P.C} \lhd \texttt{D}\ \{\ \ldots\ \}}{public(\texttt{C})}$$

$$\frac{public(\texttt{C}) \vee \rho_{\texttt{C}} = \rho_{\texttt{D}}}{visible(\texttt{C}, \texttt{D})}$$

$$\frac{\texttt{C} <: \texttt{D} \quad public(\texttt{C}) \vee conf(\texttt{D})}{\texttt{C} \preceq \texttt{D}}$$

Congruence:

$$\frac{\texttt{e} \rightarrow \texttt{e}'}{\texttt{new C}(\ldots\texttt{e}\ldots) \rightarrow \texttt{new C}(\ldots\texttt{e}'\ldots)}$$

$$\frac{\texttt{e} \rightarrow \texttt{e}'}{\texttt{(C) e} \rightarrow \texttt{(C) e}'} \qquad \frac{\texttt{e} \rightarrow \texttt{e}'}{\texttt{e.f} \rightarrow \texttt{e}'\texttt{.f}}$$

$$\frac{\texttt{e} \rightarrow \texttt{e}'}{\texttt{e.m}(\overline{\texttt{e}}) \rightarrow \texttt{e}'\texttt{.m}(\overline{\texttt{e}})} \qquad \frac{\texttt{e} \rightarrow \texttt{e}'}{\texttt{e}_0\texttt{.m}(\ldots\texttt{e}\ldots) \rightarrow \texttt{e}_0\texttt{.m}(\ldots\texttt{e}'\ldots)}$$

Fig. 2. Confined FJ

The *anon* predicate holds for a method `m` in class `C`, if the pseudo variable `this` is used solely for field selection and method invocation of methods that are anonymous themselves.
The other definitions are utilities mostly straight out of the FJ paper. *fields* returns the list of all fields of a class including inherited ones. *mdef* returns the name of defining class for a given method. *mtype* returns the type signature of a method. *mbody* returns the body of the method in a given class. The predicate *override* holds if a method is valid redefinition of an inherited method.

3.2 Reduction

The dynamic semantics of our language is given by a reduction relation of the form $\texttt{e} \rightarrow \texttt{e}'$. As usual, $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$. The congruence rules are explicitly specified though not very surprising. There are three reductions rules. (R-Field) evaluates a field access expression. (R-Cast) evaluates a cast expression, and (R-Invk) evaluates a method invocation expression.
ConfinedFJ has a by-value semantics which requires that arguments be fully evaluated before performing method invocations or field access. Notice that the left hand sides of the reduction rules (R-Field), (R-Cast), and (R-Invk) contain expressions of the form $\texttt{new C}(\overline{\texttt{v}})$. While this is superficially closer to a real

Field look-up:

$$\mathit{fields}(\texttt{Object}) = ()$$

$$\frac{\mathit{fields}(\texttt{D}) = (\overline{\texttt{D}}\ \overline{\texttt{g}}) \quad CT(\texttt{C}) = \circ\ \texttt{class P.C} \lhd \texttt{D}\ \{\ \overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\ \}}{\mathit{fields}(\texttt{C}) = (\overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}})}$$

Method definition lookup:

$$\frac{CT(\texttt{C}) = \circ\ \texttt{class P.C} \lhd \texttt{D}\ \{\ \overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\ \} \quad \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\ \{\,\texttt{return e};\ \} \in \overline{\texttt{M}}}{\mathit{mdef}(\texttt{m},\ \texttt{C}) = \texttt{C}}$$

$$\frac{CT(\texttt{C}) = \circ\ \texttt{class P.C} \lhd \texttt{D}\ \{\ \overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\ \} \quad \text{method m is not defined in } \overline{\texttt{M}}}{\mathit{mdef}(\texttt{m},\ \texttt{C}) = \mathit{mdef}(\texttt{m},\ \texttt{D})}$$

Method type lookup:

$$\frac{\mathit{mdef}(\texttt{m},\ \texttt{C}) = \texttt{D} \quad CT(\texttt{D}) = \circ\ \texttt{class P.D} \lhd \texttt{E}\ \{\ \overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\ \} \quad \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\ \{\,\texttt{return e};\ \} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m},\ \texttt{C}) = \overline{\texttt{B}} \to \texttt{B}}$$

Method body look-up:

$$\frac{\mathit{mdef}(\texttt{m},\ \texttt{C}) = \texttt{D} \quad CT(\texttt{D}) = \circ\ \texttt{class P.D} \lhd \texttt{E}\ \{\ \overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\ \} \quad \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\ \{\,\texttt{return e};\ \} \in \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m},\ \texttt{C}) = (\overline{\texttt{x}},\ \texttt{e})}$$

language such as Java, the reason for this choice is that it deals with values greatly simplifies the proof of the confinement property.

3.3 Typing

Typing rules are given in Figure 4, and they check for reference widening of types. In the typing rules, the condition of the form $\texttt{C} \preceq \texttt{D}$ requires that C be a subtype of D and if C is a confined type, then D must be confined as well. The typing rules for expressions are similar to those in FJ except that reference widening is not allowed. In particular, Rule T-New prevents instantiating an object with fields of public types by arguments of confined types. Rule T-Invk prevents passing arguments of confined types to a method with parameters of public

Valid method overriding:

$$\frac{\mathit{mtype}(\mathtt{m},\ \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{D}\ \Rightarrow\ (\mathtt{C} = \mathtt{D}\ \wedge\ \overline{\mathtt{C}} = \overline{\mathtt{D}})}{\mathit{override}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{C}} \to \mathtt{C})}$$

Anonymous method:

$$\frac{\mathit{mdef}(\mathtt{m},\ \mathtt{C}) = \mathtt{D} \quad \mathit{mbody}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e}) \quad \mathit{anon}(\mathtt{e}, \mathtt{D})}{\mathit{anon}(\mathtt{m}, \mathtt{C})}$$

$$\frac{\mathit{anon}(\mathtt{e}, \mathtt{C})}{\mathit{anon}((\mathtt{D})\ \mathtt{e}, \mathtt{C})} \qquad \frac{\mathit{anon}(\overline{\mathtt{e}}, \mathtt{C})}{\mathit{anon}(\mathtt{new}\ \mathtt{D}(\overline{\mathtt{e}}), \mathtt{C})} \qquad \frac{}{\mathit{anon}(\mathtt{x}, \mathtt{C})}$$

$$\frac{}{\mathit{anon}(\mathtt{this.f}, \mathtt{C})} \qquad \frac{\mathit{anon}(\mathtt{e}, \mathtt{C})}{\mathit{anon}(\mathtt{e.f}, \mathtt{C})}$$

$$\frac{\mathit{anon}(\mathtt{m}, \mathtt{C}) \quad \mathit{anon}(\overline{\mathtt{e}}, \mathtt{C})}{\mathit{anon}(\mathtt{this.m}(\overline{\mathtt{e}}), \mathtt{C})} \qquad \frac{\mathit{anon}(\mathtt{e}, \mathtt{C}) \quad \mathit{anon}(\overline{\mathtt{e}}, \mathtt{C})}{\mathit{anon}(\mathtt{e.m}(\overline{\mathtt{e}}), \mathtt{C})}$$

Fig. 3. Auxiliary Definitions

types. Moreover, if the called method is defined in a class of public type while the receiver expression of the call is of confined type, then the method must be anonymous in order to prevent implicit reference widening of the variable `this`. Rule T-Cast prevents casting an expression of confined type to public type.

Expression typing:

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

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C}_0 \quad \mathit{fields}(\mathtt{C}_0) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}})}{\Gamma \vdash \mathtt{e.f}_\mathtt{i} : \mathtt{C}_\mathtt{i}} \qquad \text{(T-Field)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{e} : \mathtt{C}_0 \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \\ \mathit{mtype}(\mathtt{m},\ \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \\ \overline{\mathtt{C}} \preceq \overline{\mathtt{D}} \quad \mathit{mdef}(\mathtt{m},\ \mathtt{C}_0) = \mathtt{D}_0 \\ \mathtt{C}_0 \preceq \mathtt{D}_0 \vee \mathit{anon}(\mathtt{m}, \mathtt{D}_0)\end{array}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(T-Invk)}$$

$$\frac{\begin{array}{l}\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{D}}\ \overline{\mathtt{f}}) \\ \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \quad \overline{\mathtt{C}} \preceq \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(T-New)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C}' \quad \mathit{public}(\mathtt{C}') \vee \mathit{conf}(\mathtt{C})}{\Gamma \vdash (\mathtt{C})\ \mathtt{e} : \mathtt{C}} \qquad \text{(T-Cast)}$$

Method typing:

$$\frac{\begin{array}{l}\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C}_0 \vdash \mathtt{e} : \mathtt{D} \quad \mathtt{D} \preceq \mathtt{C} \\ \mathit{override}(\mathtt{m}, \mathtt{D}_0, \overline{\mathtt{C}} \to \mathtt{C}) \\ \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C}_0 \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}_0)\end{array}}{\mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\,\{\,\mathtt{return}\,\mathtt{e};\,\}\ \mathtt{OK\ IN}\ \mathtt{C}_0 \lhd \mathtt{D}_0} \quad \text{(T-Method)}$$

Class typing:

$$\frac{\begin{array}{l}\mathit{visible}((\overline{\mathtt{C}}, \mathtt{D}), \mathtt{C}) \quad \mathit{public}(\mathtt{D}) \ \vee\ \mathit{conf}(\mathtt{C}) \\ \mathit{fields}(\mathtt{D}) = (\overline{\mathtt{D}}\ \overline{\mathtt{g}}) \quad \overline{\mathtt{M}}\ \mathtt{OK\ IN}\ \mathtt{C} \lhd \mathtt{D} \\ \mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{C}}\ \overline{\mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}\end{array}}{\circ\ \mathtt{class\ P.C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \quad \text{(T-Class)}$$

The definition of anonymous method is given in Figure 3. A method defined in class D is anonymous if the return expression e of the method is anonymous in D. That is, in e, the variable this can only be used for accessing fields and for invoking anonymous methods defined or inherited in D.

In order to type check a package modularly, we only want to check for reference widening for the types that are confined within the package. To verify condition $\mathtt{C} \preceq \mathtt{D}$, we seems to lose the ability to type check modularly. However, because in method typing rule, we require that every subexpression of a method body must have a type visible in the class where the method is defined, the type C in the condition $\mathtt{C} \preceq \mathtt{D}$ is always visible inside the package where the condition is checked. Also, in typing rule for class, we require the subtype of a confined type to be confined in the same package. Thus, to verify $\mathtt{C} \preceq \mathtt{D}$, we only need to check that $\mathtt{C} <: \mathtt{D}$ and if C is a confined type in the current package, then D must be confined type in the same package as well.

Figure 4 contains typing rules for expressions, methods, and classes, and also rules for checking visibility of expressions. Expression typing rules are similar to those in FJ with some additional constraints. For a method call of the form $\mathtt{e.m}(\overline{\mathtt{e}})$, Rule (T-Invk) prevents explicit reference widening when the actual parameter $\overline{\mathtt{e}}$ is passed to the method m; also if e is of confined type and m is defined in a public class, then m must be anonymous in order to prevent implicit reference widening. For a new expression of the form $\mathtt{new\ C}(\overline{\mathtt{e}})$, Rule (T-New) prevents explicit reference widening when $\overline{\mathtt{e}}$ is passed to the constructor of C. Lastly, Rule (T-Cast) prevents an expression of confined type from being casted to public type.

In the typing rule for method, every subexpression of a method body must have a type visible in the class where the method is defined. The consequences of this requirement are that a method m that returns value of confined type can not be invoked outside the package where m is defined; a field of confined type in a object of type C can not be accessed outside the package where C is defined; a confined type C can not be used in cast or object instantiation outside the package where

Expression visibility:

$$\frac{\Gamma \vdash \mathtt{x} : \mathtt{C} \quad \mathit{visible}(\mathtt{C}, \mathtt{C}_0)}{\Gamma \vdash \mathit{visible}(\mathtt{x}, \mathtt{C}_0)} \quad \text{(V-VAR)}$$

$$\frac{\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}_0) \quad \Gamma \vdash \mathtt{e.f_i} : \mathtt{C} \quad \mathit{visible}(\mathtt{C}, \mathtt{C}_0)}{\Gamma \vdash \mathit{visible}(\mathtt{e.f_i}, \mathtt{C}_0)} \quad \text{(V-FIELD)}$$

$$\frac{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{C} \quad \mathit{visible}(\mathtt{C}, \mathtt{C}_0) \quad \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}_0) \quad \Gamma \vdash \mathit{visible}(\overline{\mathtt{e}}, \mathtt{C}_0)}{\Gamma \vdash \mathit{visible}(\mathtt{e.m}(\overline{\mathtt{e}}), \mathtt{C}_0)} \quad \text{(V-INVK)}$$

$$\frac{\mathit{visible}(\mathtt{C}, \mathtt{C}_0) \quad \Gamma \vdash \mathit{visible}(\overline{\mathtt{e}}, \mathtt{C}_0)}{\Gamma \vdash \mathit{visible}(\mathtt{new\ C}(\overline{\mathtt{e}}), \mathtt{C}_0)} \quad \text{(V-NEW)}$$

$$\frac{\mathit{visible}(\mathtt{C}, \mathtt{C}_0) \quad \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}_0)}{\Gamma \vdash \mathit{visible}((\mathtt{C})\ \mathtt{e}, \mathtt{C}_0)} \quad \text{(V-CAST)}$$

Fig. 4. Typing Rules

C is defined. These consequences correspond to the confinement rule restrictions in [16] that the access modifiers of methods and fields of confined types can not be protected or public and the access modifiers of confined classes can not be public.
We assume that all classes are maintained in a class table CT and CT is well-typed if all classes in CT are well-typed. For the rest of the paper, we assume that CT is well-typed.

3.4 Properties

In this section, we describe the properties of confined objects. Suppose we have an object o of confined type C defined in package P. The typing rules guarantee that

1. only objects of types defined in P can hold references to o, and
2. only methods defined in P or anonymous methods inherited by C can access the fields and methods of o.

We prove the above properties in Theorem 2 and 3. In Theorem 2, we prove that in a well-typed object expression $\mathtt{new\ C_0}(\overline{\mathtt{e}})$, $\overline{\mathtt{e}}$ can only be reduced to objects visible in the package of $\mathtt{C_0}$. In Theorem 3, we show that for a well-typed method call expression $\mathtt{new\ C_0}(\overline{\mathtt{u}}).\mathtt{m}(\overline{\mathtt{v}})$, where m is defined in class $\mathtt{D_0}$, all accessible objects during the evaluation of the method call are visible in the package of $\mathtt{D_0}$ except that if m is anonymous, then $\mathtt{new\ C_0}(\overline{\mathtt{u}})$ is accessible for field select and invocation of anonymous methods in $\mathtt{C_0}$.

We also prove the usual subject reduction lemma and state the progress lemma. For the subject reduction lemma, we show that an expression of public type can not be reduced to an expression of confined type. In Theorem 1, we prove that a well-type expression will not get stuck and an expression of public type will not be reduced to an object of confined type.

Lemma 1. Subject Reduction: *If* $\emptyset \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{e} \rightarrow \mathtt{e}'$ *then* $\emptyset \vdash \mathtt{e}' : \mathtt{C}'$ *for some* $\mathtt{C}' \preceq \mathtt{C}$.

Proof. See appendix.

Lemma 2. Progress: *If* $\emptyset \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{e}$ *contains subexpression* $\mathtt{e}_0$ *where*

$$\begin{aligned} \mathtt{e}_0 = {} & \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}}).\mathtt{f}_\mathtt{i} \\ & \mid (\mathtt{C})\ \mathtt{new}\ \mathtt{C}'(\overline{\mathtt{v}}),\ \mathtt{C}' <: \mathtt{C}, \\ & \mid \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{u}}).\mathtt{m}(\overline{\mathtt{v}}), \end{aligned}$$

then there exists $\mathtt{e}' \neq \mathtt{e}$ *such that* $\mathtt{e} \rightarrow \mathtt{e}'$.

In normal termination, an expression reduces to a fully-evaluated object of the form $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})$. An irreducible expression of the form $(\mathtt{C})\ \mathtt{new}\ \mathtt{C}'(\overline{\mathtt{v}})$, where $\mathtt{C}'$ is not a subclass of $\mathtt{C}$, represents a failed cast, An irreducible expression is *stuck* if it contains subexpressions of the form $\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{v}}).\mathtt{f}_\mathtt{i}$ or $\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{u}}).\mathtt{m}(\overline{\mathtt{v}})$.

Theorem 1. *If* $\emptyset \vdash \mathtt{e} : \mathtt{C}$ *and* $\mathtt{e} \rightarrow^* \mathtt{e}'$, *then* $\mathtt{e}'$ *is not stuck, and* $\exists \mathtt{C}'$ *such that* $\emptyset \vdash \mathtt{e}' : \mathtt{C}'$, *where* $\mathtt{C}' \preceq \mathtt{C}$.

Proof. Immediate from Lemma 1 and 2.

Theorem 2. *If* $\emptyset \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}}) : \mathtt{C}_0$, $\mathtt{e} \in \overline{\mathtt{e}}$, *and* $\mathtt{e} \rightarrow^* \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})$, *then visible*$(\mathtt{C}, \mathtt{C}_0)$.

Proof. From Rule (T-New) and $\emptyset \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{e}}) : \mathtt{C}_0$, if the object expressions $\overline{\mathtt{e}}$ have type $\overline{\mathtt{C}}$ and *fields*$(\mathtt{C}_0) = (\overline{\mathtt{D}}\ \overline{\mathtt{f}})$, then we must have $\overline{\mathtt{C}} \preceq \overline{\mathtt{D}}$. From Rule (T-Class), $\overline{\mathtt{D}}$ must be types visible in the package of $\mathtt{C}_0$. Therefore, $\overline{\mathtt{C}}$ must be visible in the package of $\mathtt{C}_0$ because otherwise $\overline{\mathtt{C}} \preceq \overline{\mathtt{D}}$ would not hold. Since $\mathtt{e} \in \overline{\mathtt{e}}$, there exists $\mathtt{i}$ such that $\emptyset \vdash \mathtt{e} : \mathtt{C}_\mathtt{i}$. From Theorem 1 and $\mathtt{e} \rightarrow^* \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})$, we have $\mathtt{C} \preceq \mathtt{C}_\mathtt{i}$. Thus, $\mathtt{C}$ must be visible in the package of $\mathtt{C}_0$.

Theorem 3. *Suppose that*

$$\begin{aligned} & \emptyset \vdash \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{u}}).\mathtt{m}(\overline{\mathtt{v}}) : \mathtt{D},\ \mathit{mbody}(\mathtt{m},\ \mathtt{C}_0) = (\overline{\mathtt{x}},\ \mathtt{e}_0), \\ & \mathit{mdef}(\mathtt{m},\ \mathtt{C}_0) = \mathtt{D}_0,\ \text{and}\ \mathtt{d}_0 = [\,{}^{\overline{\mathtt{v}}}/_{\overline{\mathtt{x}}},\ {}^{\mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{u}})}/_{\mathtt{this}}]\mathtt{e}_0. \end{aligned}$$

for every subexpression $\mathtt{d}$ *of* $\mathtt{d}_0$,

1. *either* $\mathtt{m}$ *is anonymous and* $\mathtt{d} = \mathtt{new}\ \mathtt{C}_0(\overline{\mathtt{u}})$ *is used as the receiver of a method call or field select in* $\mathtt{d}_0$,
2. *or* $\mathtt{d} \rightarrow^* \mathtt{new}\ \mathtt{C}(\overline{\mathtt{u}'})$ *implies visible*$(\mathtt{C}, \mathtt{D}_0)$.

Proof. See appendix.

4 Related work

Reference semantics permeate object-oriented programming languages, and the issue of controlling aliasing has been the focus of numerous papers in the recent years [12, 11, 8, 1, 15, 9, 14, 7]. In [15], flexible alias protection is presented as a means to control potential aliasing amongst components of an aggregate object. Clarke, Potter, and Noble [7] have formalized representation containment by means of ownership types. Boyland, Noble and Retert [4] introduced capabilities as a uniform system to describe restrictions imposed on references. Recent work by Boyapati *et al.* [2, 3] has applied ownership types to scoped memory in the Real-time Specification for Java.

5 Conclusion

This paper has formalized the notion of *confined type* [16] in the context of an minimal object calculus modeled after Featherweight Java. A static type system that mirrors the informal rules confinement was proposed and proven sound. The confinement invariant was shown to hold for well-typed programs.
In the second part of the paper, definition of confined types was extended to confined instantiation of generic classes. This allows for confined collection types in Java and for classes that can be confined *post hoc*. Confinement type rules are given for Generic Featherweight Java, and proven sound. A generic confinement invariant is established and proven for well-typed programs.
Acknowledgments. The authors thank James Noble for frequent inspiration and encouragements, Dave Clarke for one fewer reduction rule, Christian Grothoff for his boundless energy in generating experimental data, John Boyland for explaining borrowing, and to Phil Wadler providing us with the initial impetus. Finally we thank the anonymous referees of IWACO for valuable comments on a version of this work.

References

1. Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *LNCS*, pages 32–59, Jyväskylä, Finland, 9–13 June 1997. Springer-Verlag.
2. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, November 2002.
3. Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. June 2003.
4. John Boyland, James Noble, and William Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, number 2072 in Lecture Notes in Computer Science, Berlin, Heidelberg, New York, 2001. Springer.

5. Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad Beans: Deployment-time confinement checking. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, Anaheim, CA, November 2003.
6. David Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.
7. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, October 1998.
8. D. Detlefs, K. Rustan M. Leino, and G. Nelson. Wrestling with rep exposure. Technical report, Digital Equipment Corporation Systems Research Center, 1996.
9. Daniela Genius, Martin Trapp, and Wolf Zimmermann. An approach to improve locality using Sandwich Types. In *Proceedings of the 2nd Types in Compilation workshop*, volume LNCS 1473, Kyoto, Japan, March 1998. Springer Verlag.
10. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, 2001.
11. John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Proceedings of the OOPSLA '91 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 271–285, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.
12. John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992.
13. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
14. S.J.H. Kent and I. Maung. Encapsulation and Aggregation. In *Proceedings of TOOLS PACIFIC 95 (TOOLS 18)*. Prentice Hall, 1995.
15. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98— Object-Oriented Programming*, volume 1445 of *Lecture Notes In Computer Science*, pages 158–185, Berlin, Heidelberg, New York, July 1988. Springer-Verlag.
16. J. Vitek and B. Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.

A Static Capability Tracking System

Scott F. Smith and Mark Thober

The Johns Hopkins University
{scott,mthober}@cs.jhu.edu

Abstract. Capabilities may be used within programming languages to define and enforce security policies: a user must hold a reference to an object or method to be able to use its features. We define a system for statically tracking capability flow via types. This work is related to Bokowski and Vitek's Confined Types, and Skalka and Smith's `pop` system. These previous systems concentrated on constraining capability flow at an object level, and enforcement at object or package points. We here propose a general system for confining the flow of capabilities in a program. We define a toy functional language, *fcap*, which includes a general mechanism for declaring and enforcing secure boundaries for capability flow, and we show how these boundaries may be statically enforced through types. We also sketch how Confined Types and `pop` may be encoded in *fcap*.

1 Introduction

The notion of capabilities was first introduced by Dennis and Van Horn in 1966 [3]. They define a capability as a location of an object (a pointer) and a specific set of actions that can be performed on this object.

In this paper, we take a capability-based view of security; in particular, we treat each function as a capability. Thus, in the context of this paper, each function *is* a capability, and possession of a capability really just means possession of a function. Programming languages such as J-Kernel [7], and the E language [6] show how capabilities alone may be used to create powerful security architectures within programming languages.

Previous work on confinement of object references [12, 10, 2] also is takes a capability-based view of security: an object reference is a capability to an object. These systems additionally use types to statically enforce object reference confinement. Our approach is similar to these systems; but, we allow the programmer to place capability escape checks at arbitrary points in a program. This gives the programmer the ability to define code boundaries of their choosing, and does not limit code boundaries to objects or packages. We show our system can encode the core ideas of the above confinement systems via simple translations.

We create a language where each function has a corresponding capability tag which will allow us to explicitly monitor and restrict the flow of the function. The eventual possessor of the function has complete access to it since it represents a pure capability. We use functions instead of objects here because the calculi are simpler; in general a reference to any value can be considered a capability, and we restrict ourselves here to functions only for simplicity of the presentation.

We define a language *fcap*, for this purpose, defining each function with a corresponding *capability tag*. For example, $\lambda^{c_1} x.e$ defines the function $\lambda x.e$ with corresponding capability tag c_1.

The capability tags may then be used to explicitly delimit the extent of function value flow at runtime, via a runtime *check*. These checks also are enforced statically by the type system. Thus, if a capability erronously escapes, an appropriate check will catch this and will lead to a type error, eliminating the need for runtime monitoring of capability flow.

We use a translation approach in the style of [8] to translate *fcap* to an ML-subset language, *fml*. This allows us to rely upon the properties and proofs of ML, such as soundness results, without

having to recreate the proofs from scratch for *fcap*. *fml* contains variants, and they are the key to implementing the tracking of capability flow. This also shows how our approach here is dual to the encoding of `pop` in [10], where records and not variants are used to track capability flow.

We begin in section 2 by defining the source language, *fcap*, which contains function capabilities. Section 3 defines the target language, *fml*, and section 4 gives the translation of *fcap* programs into *fml*. The Type Translation and type system for *fml* and *fcap* are given in section 5. In Section 6, we discuss related work, giving simplified encodings of [12, 10]. We conclude in Section 7.

2 The Source Language *fcap*

We begin by defining our explicit capability language, *fcap*. The grammar for *fcap* is given in Figure 1. *fcap* is a λ-calculus with records, and functions are given capability tags. The set $\mathcal{C}$ is a countably infinite set of atomic capability tags $\{c_1, c_2, \dots\}$, and all functions are tagged with a capability from $\mathcal{C}$. Thus, we write functions as $\lambda^{c_1} x.e$. Our syntax only provides for tagging functions, but records or other data could also be tagged and in fact it is trivial to encode this in *fcap*: let $x = \ e$ in$\lambda^c y.x$ for $y \neq x$ labels arbitrary e with tag c.

The check expression, $\mathit{check}(e, [c_1, \dots, c_n])$, dynamically checks if e is tagged with a capability tag from $[c_1, \dots, c_n]$, a list of capability tags. $\mathit{dc}(e, [c_1, \dots, c_n])$ performs a deeper check of e, verifying that no nested data or return values are tagged with tags not in $[c_1, \dots, c_n]$. We will shortly elaborate on this.

$c \in \mathcal{C}$	*capabilities*
$v ::= k$ (*integer constant*) $\mid b$ (*boolean constant*) $\mid ()$ (*unit*) $\mid \lambda^c x.e$	*values*
$e ::= v \mid$ *if* e *then* e *else* $e \mid e\ e \mid$ let $x = e$ in $e \mid e; e \mid e + e$	*expressions*
$\{m_1 = e_1; \dots; m_n = e_n\} \mid e.m \mid \mathit{check}(e, [c_1, \dots, c_n]) \mid \mathit{dc}(e, [c_1, \dots, c_n])$	

Fig. 1. Grammar for *fcap*

The key operational semantics rules for *fcap*, in big step form, are given in Figure 2; the rules not listed are standard. The rules (check) and (deepcheck) are discussed in section 2.1.

2.1 Capability Checking

The checking rules of *fcap* serve to limit the flow of certain capabilities, creating barriers that only allow certain capabilities to pass through, disallowing all others. We define two different forms of check, *check* and *dc* (deepcheck).

check expressions Assuming e is a function tagged with c, the program may only proceed if c is listed in $[c_1, \dots, c_n]$. If e is not a function, the check always succeeds. This means *check* statements can be used to ensure no unwanted capabilities will be allowed to pass through this check point. Checks that fail will cause the program execution to get stuck; the goal of our type system will be to ensure typed programs never get stuck in this manner.

deep check expressions The deep check, *dc*, provides a more complete monitoring of capability flow, detecting if any value could ever directly escape in the future. We say a function, f, *escapes* a surrounding function, g, if after applying g, the function f can be used. For instance, the function

$$\frac{e_1 \to \lambda^c x.e,\ e_2 \to v',\ e[v'/x] \to v}{e_1\ e_2 \to v} \quad \text{(appl)}$$

$$\frac{e_1 \to \lambda^{c_l} x.e}{check(e_1, [c_1, \ldots, c_l, \ldots, c_n]) \to \lambda^{c_l} x.e} \quad \text{(check-fun)}$$

$$\frac{e_1 \to v, v \neq \lambda^{c_l} x.e}{check(e_1, [c_1, \ldots, c_m, \ldots, c_n]) \to v} \quad \text{(check-val)}$$

$$\frac{e_1 \to \lambda^{c_l} x.e_2}{dc(e_1, [c_1, \ldots, c_l, \ldots, c_n]) \to \lambda^{c_l} x.dc(e_2, [c_1, \ldots, c_l, \ldots, c_n])} \quad \text{(deepcheck-fun)}$$

$$\frac{e \to \{m_1 = e_1; \ldots; m_n = e_n\}}{dc(e, [c_1, \ldots]) \to \{m_1 = dc(e_1, [c_1, \ldots]); \ldots; m_n = dc(e_n[, c_1, \ldots])\}} \quad \text{(deepcheck-rec)}$$

$$\frac{e_1 \to v, v \neq \lambda^{c_l} x.e}{dc(e_1, [c_1, \ldots, c_n]) \to v} \quad \text{(deepcheck-val)}$$

Fig. 2. Key Operational Semantics Rules for *fcap*

$\lambda^{c_x} x.\lambda^{c_y} y.e$, when applied will return the inner function, $\lambda^{c_y} y.e$. Thus, the capability tagged with c_y escapes from the original function.

A *dc* statement recursively checks inside functions and records to verify that only explicitly allowed capababilties may escape. Thus, in the previous example, if c_y was not listed in $[c_1, \ldots, c_n]$, a *dc* on $\lambda^{c_x} x.\lambda^{c_y} y.e$ would fail. Notice a regular *check* on this expression would succeed, so long as c_x was in $[c_1, \ldots, c_n]$. The deep check is performed lazily, so the example $dc(\lambda^{c_x} x.\lambda^{c_y} y.e, [c_x])$ would not itself get stuck, but any attempt to apply the result of this deep check would get stuck, e.g. $dc(\lambda^{c_x} x.\lambda^{c_y} y.e, [c_x])()$. One other observation from this example is the deep check must be used as a filter: the result returned from the *dc* must be the value that is allowed to escape, because it may contain a recursive *dc* to check future values; see the (deepcheck-fun) rule.

While we can use *dc* to capture any capability actually observed to be escaping, capabilities can indirectly escape and this is a much more difficult problem. Indirectly escaping capabilities are those in which all or some of the functionality of the capability escapes through some function, but the capability itself does not escape. For example, the function $\lambda^{c_x} x.(\lambda^{c_y} y.e)\ x$ is essentially the function $\lambda^{c_y} y.e$ put in a wrapper by c_x. Therefore, the functionality of c_y indirectly leaks from the original function, even though c_y is never directly accessible and so a deep check which disallows c_y would not fail. We do not address indirectly escaping capabilities in this work, but our system could be generalized to track some of them. In general, the more indirect the information tracked, the less precise are the static guarantees that are possible. Information flow [13, 8] tracks all indirect data flow, but to enforce properties statically many well-behaved programs in fact have to be rejected, questioning the logic of the approach.

3 Target Language *fml*

The target language, *fml*, in the style of the pml_B language from [11], is a subset of ML that includes pairs, records, match statements, recursive functions, and variants. The grammar for *fml* is given in Figure 3. The set $\mathcal{V}$ is a countably infinite set of variants $\{c_1, c_2, \ldots\}$. The grammar rules for *fml* are standard. We make a distinction between match-statements with regular variants, c, and those carrying an argument, c of e. However, in order to simplify the language, we use c of () to represent a variant c, which carries no argument. The operational semantics rules for *fml* are standard and so are not presented here.

$c \in \mathcal{V}$	*variants*
$v ::= k$ (*integer constant*) $\mid b$ (*boolean constant*) $\mid ()$ (*unit*) $\mid \lambda x.e \mid c$ of e	*values*
$e ::= v \mid$ *if* e *then* e *else* $e \mid e\ e \mid$ let $x = e$ in $e \mid e; e \mid e + e \mid (e, e) \mid$ $fst(e, e) \mid snd(e, e) \mid \{m_1 = e_1; \ldots; m_n = e_n\} \mid e.m \mid$ let rec $f\ x$ in $e \mid$ match e with $(c$ of $x \to e \mid c$ of $x \to e \mid \ldots)$	*expressions*

Fig. 3. Target Grammar

4 Source-to-Target Translation

Meaning of *fcap* programs is defined by translation to *fml* programs. The *fcap*-to-*fml* translation is given in Figure 4. Each value is translated to a variant of that value, ie. *Int* of x. This in effect labels each value with run-time type information and allows *check* and *dc* to operate on all types of expressions.

Functions in *fcap* are translated to pairs consisting of the function definition, and an associated capability tag. This is then wrapped into the *Fun* variant. Application is then translated to an expression which takes off the *Fun* wrapper, then applies the *fst* of the pair (this being the function definition) to the argument. Application ignores the tag, it is only used by the checks.

For records to be successfully encoded to implement *dc*, each unique record must be assigned its own specific variant. So, before doing the actual translation of an expression, a first pass is done to pick out all the records in the program, and assign a unique variant, Rec_i to the ith occuring record definition in the program. This is necessary for the recursive call in the *dc* translation, which deepchecks every field of the record.

The check expression is an identity operation except for the case the argument is a function, in which case it verifies its tag is one of those in $[c_1, \ldots, c_n]$, via a match. If the tag is not one of those in this list, the match statement will get stuck, indicating a security violation. Such programs will not typecheck, giving a static guarantee the ill-tagged function will not escape.

The deepcheck translation is related to *check* but is a recursive function. Functions and records are recursively analyzed by a call to *dm*, which recursively deepchecks within the Function or Record. Since the recursive call to *dm* for functions is inside a λ, it will be lazily performed if/when this function is later used. The *dc* expression thus behaves as a filter, in that the output of *dc* must be the value that escapes, and computation may later get stuck due to delayed discovery of an enclosing function that improperly leaks a capability.

[8] uses a somewhat similar methodology to this paper in the use of an encoding which places explicit tags on data; the propagation of tags in our aproach differs significantly because information flow takes a very conservative approach with respect to the possibility for indirect data flow. And, there is no dynamic check in information flow typing.

We now assert the soundness of our translation into *fml* with respect to the original *fcap* operational semantics.

Definition 4.1 (β-conversion). *$v_1 =_\beta v_2$ is the least congruence satisfying the β-conversion rule: $\lambda x.e_1\ e_2 \equiv e_1[e_2/x]$.*

Theorem 4.2. *If $e \to v$, then $[\![e]\!] \to v'$, where $v' =_\beta [\![v]\!]$. If e goes wrong, then $[\![e]\!]$ goes wrong.*

5 Type Translation

We now define the types and type systems for *fcap* and *fml*, and a translation from *fcap* types to *fml* types. We show that type soundness of *fcap* can easily be established given the type soundness

$$
\begin{array}{rl}
[\![x]\!] & = x \\
[\![k]\!] & = \mathit{Int}\ \text{of}\ k \\
[\![b]\!] & = \mathit{Bool}\ \text{of}\ b \\
[\![()]\!] & = \mathit{Unit}\ \text{of}\ () \\
[\![\lambda^c x.e]\!] & = \mathit{Fun}\ \text{of}\ (\lambda x.[\![e]\!], c\ \text{of}\ ()) \\
[\![\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3]\!] & = \text{match}\ [\![e_1]\!]\ \text{with} \\
 & \quad \mathit{Bool\ of}\ b \rightarrow\ \mathit{if}\ b\ \mathit{then}\ [\![e_2]\!]\ \mathit{else}\ [\![e_3]\!] \\
[\![\text{let}\ x\ =\ e_1\ \text{in}\ e_2]\!] & = \text{let}\ x\ =\ [\![e_1]\!]\ \text{in}\ [\![e_2]\!] \\
[\![e_1; e_2]\!] & = [\![e_1]\!]; [\![e_2]\!] \\
[\![e_1 + e_2]\!] & = \text{match}[\![e_1]\!]\ \text{with}\ \mathit{Int}\ \text{of}\ k_1 \rightarrow \\
 & \quad \text{match}[\![e_2]\!]\ \text{with}\ \mathit{Int}\ \text{of}\ k_2 \rightarrow \mathit{Int}\ \text{of}\ k_1 + k_2 \\
[\![e_1\ e_2]\!] & = \text{match}([\![e_1]\!])\ \text{with}\ \mathit{Fun}\ \text{of}\ x \rightarrow \mathit{fst}(x)\ [\![e_2]\!] \\
[\![\{\overline{m = e}\}]\!] & = \mathit{Rec}_i\ \text{of}\ \{\overline{m = [\![e]\!]}\} \\
 & \quad \text{for}\ \{m = [\![e]\!]\}\ \text{the}\ i^{th}\ \text{record in the program.} \\
[\![e.m]\!] & = \text{match}\ [\![e]\!]\ \text{with} \\
 & \quad \mathit{Rec}_i\ \text{of}\ r\ \rightarrow\ r.m \\
[\![\mathit{check}\ (e_1, [c_1, \ldots, c_n])]\!] & = \text{match}([\![e_1]\!])\ \text{with} \\
 & \quad \mathit{Fun}\ \text{of}\ x \rightarrow \text{match}\ \mathit{snd}(x)\ \text{with} \\
 & \qquad c_1\ \text{of}\ () \rightarrow \mathit{Fun}\ \text{of}\ x \mid \ldots \mid c_n\ \text{of}\ () \rightarrow \mathit{Fun}\ \text{of}\ x \\
 & \quad \mid \mathit{Rec}_i\ \text{of}\ r \rightarrow \mathit{Rec}_i\ \text{of}\ r \\
 & \quad \mid \mathit{Int}\ \text{of}\ x \rightarrow \mathit{Int}\ \text{of}\ x \\
 & \quad \mid \mathit{Bool}\ \text{of}\ x \rightarrow \mathit{Bool}\ \text{of}\ x \\
 & \quad \mid \mathit{Unit}\ \text{of}\ x \rightarrow \mathit{Unit}\ \text{of}\ x) \\
[\![\mathit{dc}\ (e_1, [c_1, \ldots, c_n])]\!] & = (\text{let rec}\ \mathit{dm}\ z\ = \\
 & \quad \text{match}\ z\ \text{with} \\
 & \quad \mathit{Fun}\ \text{of}\ x \rightarrow \\
 & \qquad (\text{match}\ \mathit{snd}(x)\ \text{with}\ c_1\ \text{of}\ () \rightarrow () \mid \ldots \mid c_n\ \text{of}\ () \rightarrow ()); \\
 & \qquad \mathit{Fun}\ \text{of}\ (\lambda y.(\mathit{dm}(\mathit{fst}(x)\ y)), \mathit{snd}(x)) \\
 & \quad \mid \mathit{Rec}_1\ \text{of}\ r \rightarrow \\
 & \qquad \mathit{Rec}_1\ \text{of}\ \{m_1 = \mathit{dm}(r.m_1); \ldots; m_n = \mathit{dm}(r.m_n)\} \\
 & \quad \mid \ldots \\
 & \quad \mid \mathit{Rec}_m\ \text{of}\ r \rightarrow \\
 & \qquad \mathit{Rec}_m\ \text{of}\ \{p_1 = \mathit{dm}(r.p_1); \ldots; p_n = \mathit{dm}(r.p_n)\} \\
 & \quad \mid \mathit{Int}\ \text{of}\ x \rightarrow \mathit{Int}\ \text{of}\ x \\
 & \quad \mid \mathit{Bool}\ \text{of}\ x \rightarrow \mathit{Bool}\ \text{of}\ x \\
 & \quad \mid \mathit{Unit}\ \text{of}\ x \rightarrow \mathit{Unit}\ \text{of}\ x)\ [\![e_1]\!]
\end{array}
$$

Fig. 4. Source to Target Translation

of *fml*. An expressive type system here will allow more checks to be statically determined as valid, so we use expressive type constraint systems in our presentation [1, 5, 9].

Types for *fcap* The types for *fcap* are given as follows:

$\tau \in \mathbf{Type} ::= \tau v \mid t \mid [c_1, \ldots, c_n] \mid \tau \xrightarrow{\tau} \tau \mid \{m_1 : \tau\ ;\ \ldots\ ;\ m_2 : \tau\}$
$\tau v \in \mathbf{ValueType} ::= \mathit{int} \mid \mathit{bool} \mid \mathit{unit}$
$t \in \mathbf{TypeVar}$
$\tau_1 <: \tau_2,\ \tau <: \mathit{ifFun}([c_1| \ldots |c_n]) \in \mathbf{Constraint}$
$C \in \mathbf{ConstraintSet} \subseteq 2^{\mathbf{Constraint}}$
$c \in \mathcal{C}$

Types *int*, *bool*, and *unit* are primitive types. $[c_1, \ldots, c_n]$ is a capability type indicating a possible set of capability labels. $\tau \xrightarrow{\tau} \tau$ is a function type, which is annotated with a capability type indicating

the set of capability tags the function could have. $\{m_1 : \tau\ ;\ \ldots\ ;\ m_2 : \tau\}$ is a standard record type. $\tau <: \mathit{ifFun}([c_1, \ldots, c_n])$ is a specal constraint which requires τ, if it is a function, to have capability tag in $[c_1, \ldots, c_n]$. For τ not a function, this constraint will have no effect on the type. This can be seen in the (Check) closure rule in Figure 9.

Types for *fml*

$\tau \in \mathbf{Type} ::= \tau v \mid t \mid [c_1 \text{ of } \tau | \ldots | c_n \text{ of } \tau] \mid \tau \to \tau \mid \tau \times \tau \mid \{m_1 : \tau\ ;\ \ldots\ ;\ m_2 : \tau\}$
$\tau v \in \mathbf{ValueType} ::= \mathit{int} \mid \mathit{bool} \mid \mathit{unit}$
$t \in \mathbf{TypeVar}$
$\tau_1 <: \tau_2 \in \mathbf{Constraint}$
$C \in \mathbf{ConstraintSet} \subseteq 2^{\mathbf{Constraint}}$
$c \in \mathcal{V}$

$\tau \times \tau$ are pair types, $\tau \to \tau$ are function types, and $[c_1 \text{ of } \tau | \ldots | c_n \text{ of } \tau]$ are variant types.

5.1 The Type Translation

$$
\begin{aligned}
(\!|\, \mathit{int} \,|\!) &= \mathit{Int} \text{ of } \mathit{int} \\
(\!|\, \mathit{bool} \,|\!) &= \mathit{Bool} \text{ of } \mathit{bool} \\
(\!|\, \mathit{unit} \,|\!) &= \mathit{unit} \\
(\!|\, [c_1, \ldots, c_n] \,|\!) &= [c_1 \text{ of } \mathit{unit} | \ldots | c_n \text{ of } \mathit{unit}] \\
(\!|\, \tau_1 \xrightarrow{\tau_3} \tau_2 \,|\!) &= \mathit{Fun} \text{ of } (((\!|\, \tau_1 \,|\!) \to (\!|\, \tau_2 \,|\!)) \times (\!|\, \tau_3 \,|\!)) \\
(\!|\, \{m_1 : \tau_1; \ldots; m_n : \tau_n\} \,|\!) &= \mathit{Rec}_i \text{ of } \{m_1 : (\!|\, \tau_1 \,|\!); \ldots; m_n : (\!|\, \tau_n \,|\!)\} \\
&\quad \text{for } \{m_1 : \tau_1; \ldots; m_n : \tau_n\} \text{ the } i^{th} \text{ record in the program.} \\
(\!|\, \tau <: \mathit{ifFun}([c_1, \ldots, c_n]) \,|\!) &= (\!|\, \tau \,|\!) <: [\mathit{Fun} \text{ of } (t' \times [c_1 \text{ of } () | \ldots | c_n \text{ of } ()]) | \\
&\quad \mathit{Int} \text{ of } t_i | \mathit{Bool} \text{ of } t_b | \mathit{Unit} \text{ of } t_u | \ \mathit{Rec}_1 \text{ of } t_1 | \ldots | \mathit{Rec}_n \text{ of } t_n] \\
(\!|\, \tau_1 <: \tau_2 \,|\!) &= (\!|\, \tau_1 \,|\!) <: (\!|\, \tau_2 \,|\!) \\
&\quad \text{for } \tau_2 \neq \mathit{ifFun}([c_1, \ldots, c_n]).
\end{aligned}
$$

Fig. 5. Type Translation

The type translation is given in Figure 5. Capability types in *fcap* are traslated directly to variants in *fml*. Function types in *fcap* are translated to a pair consisting of the translated function, and the capability tag translated to a variant.

5.2 Type System for *fml*

Key type rules for *fml* are given in Figure 6. These and the missing type rules are standard type constraint rules, see *e.g.* [5]. The closure, $\mathit{Closure}(C)$, of the constraint set C is defined in Figure 7. Standard inconsistencies within the constraint set will cause a type error; ie. $\mathit{Int} <: \mathit{Bool}$ is inconsistent. For variants, $[c_1 | \ldots | c_m | c_l] <: [c_1 | \ldots | c_m | \ldots | c_n]$ is inconsistent because c_l is a variant not in the right-hand list. (*i.e.*, the left list has a variant the right does not.) The algorithm for type inference is as follows: Using the type rules given in Figure 6, produce the obvious, unique proof $\vdash_t\ e : \tau \backslash C$. Let $C' = \mathit{Closure}(C)$, with $\mathit{Closure}(C)$ given above. Check C' for inconsistencies, if any give a type error. If there are no errors, the inferred type for e is $\tau \backslash C'$.

$$\frac{\Gamma,\ x:\ t\ \vdash_t\ e:\ \tau \setminus C}{\Gamma\ \vdash_t\ \lambda x.e:\ t\ \rightarrow\ \tau \setminus C}\ \text{(Function)}$$

$$\frac{\Gamma\ \vdash_t\ e:\ \tau \setminus C,\ \Gamma\ \vdash_t\ e':\ \tau' \setminus C'}{\Gamma\ \vdash_t\ e\ e':\ t \setminus C\ \cup\ C'\ \cup\ \{\tau\ <:\ \tau'\ \rightarrow\ t\}}\ \text{(Appl)}$$

$$\frac{\Gamma \vdash_t e : \tau \setminus C}{\Gamma\ \vdash_t\ c\ \text{of}\ e:\ [c\ \text{of}\ \tau] \setminus C}\text{(Inject)}$$

$$\frac{\Gamma\ \vdash_t\ e:\tau\setminus C,\ \Gamma, x_1:t_1\ \vdash_t\ e_1':\tau_1'\setminus C_1, \ldots \Gamma, x_n:t_n\ \vdash_t\ e_n':\tau_n'\setminus C_n}{\begin{array}{l}\Gamma\ \vdash_t\ match\ e\ with\ c_1\ \text{of}\ x_1 \rightarrow e_1'|\ldots|c_n\ \text{of}\ x_n \rightarrow e_n' : t' \setminus \\ C \cup C_1 \cup \cdots \cup C_n \cup \{\tau <: [c_1\ \text{of}\ t_1|\ldots|c_n\ \text{of}\ t_n]\} \cup \{\tau_1' <: t', \ldots \tau_n' <: t'\}\end{array}}\ \text{(Match-of)}$$

Fig. 6. Key Type Rules for *fml*

$$\text{(Trans)}\ \frac{\tau v <: t, t <: \tau}{\tau v <: \tau} \qquad \text{(Func)}\ \frac{\tau_1 \rightarrow \tau_1' <: \tau_2 \rightarrow \tau_2'}{\tau_2 <: \tau_1,\ \tau_1' <: \tau_2'} \qquad \text{(Pair)}\ \frac{\tau_1 \times \tau_2 <: \tau_3 \times \tau_4}{\tau_1 <: \tau_3, \tau_2 <: \tau_4}$$

$$\text{(Rec)}\ \frac{\{m_1 : \tau_1, \ldots, m_n : \tau_n, \ldots, m_l : \tau_l\} <: \{m_1 : \tau_1', \ldots, m_n : \tau_n'\}}{\tau_1 <: \tau_1', \ldots, \tau_n <: \tau_n'}$$

$$\text{(Variant)}\ \frac{[c_1\ \text{of}\ \tau_1|\ldots|c_i\ \text{of}\ \tau_i|\ldots|c_n\ \text{of}\ \tau_n] <: [c_1\ \text{of}\ \tau_1|\ldots|c_i\ \text{of}\ \tau_j|\ldots|c_m\ \text{of}\ \tau_m]}{\tau_i <: \tau_j}$$

Fig. 7. Closure rules for *fml*

5.3 Derived Type System for *fcap*.

We give the key type rules for *fcap* in figure 8, based on the type rules for *fml*, and the *fcap*-to-*fml* translation. The standard type rules have been omitted for brevity. We introduce a new constraint construct, *ifFun*, in order to check the capabilities of functions, while allowing other types through the check statement.

We do not present a type rule for deep-check, so the type system given in figure 8 applies to *fcap* programs with simple checks only. We omit this rule for brevity and simplicity of the *fcap* type system. Such a rule can be given, however, it will make the entire *fcap* type system more complex. Typing a deep-check statement may be implicitly done by translating an *fcap* program to *fml*, then using the *fml* type system to type the program. For example, the *fcap* program $dc(\lambda^{c_x} x.\lambda^{c_y} y.e, [c_x])$, when translated to *fml* will produce a type error, since c_y escapes the outer function, and only c_x is allowed by the deep-check. $dc(\lambda^{c_x} x.\lambda^{c_y} y.e, [c_x, c_y])$ however, will type check, since both c_x and c_y are allowed to escape. Even though this translation is complicated, it can easily be done as a black box for the programmer.

Computing the closure, $Closure(C)$ of the constraint set, C is given in Figure 9. (Check) uses the *ifFun* construct to create a new constraint, which ensures a capability (τ_3 here) is contained in the list $[c_1, \ldots, c_n]$.

As in the *fml* type system, standard inconsistencies within the constraint set will cause a type error; ie. *Int* <: *Bool* is inconsistent. For capabilities, $[c_1, \ldots, c_m, c_l] <: [c_1, \ldots, c_m, \ldots, c_n]$ is inconsistent, where c_l is a capability not in the right-hand list. The algorithm for type inference is the same as in *fml*, using the *fcap* type rules and closure.

$$\frac{\Gamma,\ x:\ t\ \vdash_s\ e:\ \tau\setminus C}{\Gamma\ \vdash_s\ \lambda^c x.e : t \xrightarrow{[c]} \tau\setminus C}\ \text{(Function)}$$

$$\frac{\Gamma\ \vdash_s\ e:\ \tau\setminus C,\ \Gamma\ \vdash_s\ e':\ \tau'\setminus C'}{\Gamma\ \vdash_s\ e\ e':\ t\setminus C\ \cup\ C'\ \cup\{\tau <: \tau' \xrightarrow{t''} t\}}\ \text{(Appl)}$$

$$\frac{\Gamma\ \vdash_s\ e:\tau\setminus C}{\Gamma\ \vdash_s\ \mathit{check}(e,[c_1,\ldots,c_n]) : t\setminus C\cup\{\tau <: \mathit{ifFun}([c_1,\ldots,c_n]), \tau <: t\}}\ \text{(Check)}$$

Fig. 8. Key Type Rules for *fcap*

$$\text{(Trans)}\ \frac{\tau v <: t, t <: \tau}{\tau v <: \tau} \qquad \text{(Func)}\ \frac{\tau_1 \to \tau_1' <: \tau_2 \to \tau_2'}{\tau_2 <: \tau_1,\ \tau_1' <: \tau_2'} \qquad \text{(Check)}\ \frac{\tau_1 \xrightarrow{\tau_3} \tau_2 <: \mathit{ifFun}([c_1,\ldots,c_n])}{\tau_3 <: [c_1,\ldots,c_n]}$$

$$\text{(Rec)}\ \frac{\{m_1:\tau_1,\ldots,m_n:\tau_n,\ldots,m_l:\tau_l\} <: \{m_1:\tau_1',\ldots,m_n:\tau_n'\}}{\tau_1 <: \tau_1',\ldots,\tau_n <: \tau_n'}$$

Fig. 9. Closure rules for *fcap*

Soundness of the *fcap* type system The *fcap* type system can be shown to be sound by showing its rules to be the derived rules obtained by translating *fcap* programs to *fml* and typing them in the *fml* type system. Thus, the *fcap* type system is sound provided the *fml* type system is.

There are some slight format differences in the two type systems—some type simplifications need to be performed after translation. Thus we first define a simplification algorithm for a constraint set, C, based on the garbage collection schemes from [4, 9]

We call the simplified constraint set $\mathcal{S}(C)$, a function defined by the following simplification rules.

$t <: \tau, \tau' <: t$ *becomes* $\tau' <: \tau$
$t <: \tau, \tau' <: t \times \tau''$ *becomes* $\tau' <: \tau \times \tau''$
$t <: \tau, \tau' <: [c \text{ of } t]$ *becomes* $\tau' <: [c \text{ of } \tau]$
$t <: \tau, \tau' <: \tau'' \to t$ *becomes* $\tau' <: \tau'' \to \tau$

These rules apply for any $t, \tau, \tau', \tau'', c$ in the constraint set C, as long as t does not occur in any other constraints in C. Thus, the set $\mathcal{S}(C)$ is a simplified set C'.

Lemma 5.1 (Type Soundness of *fcap*). *If* $\vdash_s e : \tau\setminus C$ *is derivable, then so is* $\vdash_t [\![e]\!] : (\!| \tau |\!)\setminus C'$, *where* $\mathcal{S}((\!| C |\!)) = \mathcal{S}(C')$.

5.4 Examples

We now give some examples of *fcap* programs containing *check* statements.

Suppose we have the following program:

```
let obj =
   (let f1 = λ^c1 x1.e1 in
      let f2 = λ^c2 x2.check(e2, []) in
         let f3 = λ^c3 x3.check(if x3 = 0 then f1 else f2, [c2]) in
            {m2 = f2; m3 = f3}) in
obj.m3 10
```

Within obj, we allow the use of f_1, however, we do not want f_1 to escape the definition of obj. Thus, we place *check*'s on the return values for those functions available outside the definition of obj, namely f_2 and f_3. Here, the *check* of the return value for f_3 will cause a type error, as f_1 may be returned. Since f_1 is the function with the tag c_1, the constraint $[c_1] <: [c_2]$ will be in the closure of the constraint set for this program. This inconsistent constraint will cause a type error.

We can fix this program as follows:

let obj =
(let $f_1 = \lambda^{c_1} x_1.e_1$ in
let $f_2 = \lambda^{c_2} x_2.check(e_2, [])$ in
let $f_3 = \lambda^{c_3} x_3.check(f_2, [c_2])$ in
$\{m_2 = f_2; m_3 = f_3\}$) in
obj.m_3 10

Here, f_3 does not leak f_1, and this program will now type check.

6 Related Work

Our system is similar to other static object confinement security mechanisms, e.g. [12, 10]. Since our system has a very basic form of tagging and checking, it can be used to encode the core concepts of these existing systems.

6.1 Confined Types

[12] uses confined types to restrict the flow of object references between packages in Java. Our checks also may be used to restrict flow, and the checks can be used anywhere at the discretion of the programmer, not just at package boundaries. While our system does not specifically use objects and packages, we sketch how confined types of [12] can be modeled by our approach in Figure 10.

co	$::= \lambda^{c_c} d.\{m_1 = \lambda x_1.e_1; \ldots; m_n = \lambda x_n.e_n\}$	Confined Objects
uo	$::= \lambda^{c_u} d.\{m_1 = \lambda x_1.check(e_1, [c_u, c_1, \ldots, c_k]); \ldots;$ $m_n = \lambda x_n.check(e_n, [c_u, c_1, \ldots, c_k])\}$	Unconfined Objects
p	$::=$ let $confined = \{f_1 = co_1, \ldots, f_n = co_n\}$ in $\{u_1 = uo_1, \ldots, u_n = uo_n\}$	Packages

Fig. 10. Confined Types encoding

Packages are viewed as let-statements, with the first part of the let-statement containing the confined objects, and the second part containing the unconfined objects. The different objects within the package are grouped in records.

An object with methods $m_1 \ldots m_n$ is simply encoded as a record: $\{m_1 = \lambda x_1.e_1; \ldots; m_n = \lambda x_n.e_n\}$. This encoding simplifies out many features such as self-reference, instances, classes, etc, and is only intended to show how the basic concepts relate.

We use a dummy λ-expression to mark each object with a capability tag, c_c or c_u, which labels the object as *confined* or *unconfined*; d is a dummy variable. To ensure no confined objects escape, we place *check* statements at all return points in the unconfined objects. $[c_1, \ldots, c_k]$ are capabilities which are allowed to escape from a package. c_c cannot be one of these capablities. This check will ensure only unconfined objects and allowed capabilities will escape the package. Confined objects cannot leave the package, as each confined object is marked with the c_c capability, which is disallowed in the check-statement for unconfined objects.

We informally justify how *check* expressions will indeed keep confined objects from escaping a package. For any unconfined object, a method cannot directly return a confined object. In order for a confined object to successfully escape from the package, it must at some point be returned by a method of an unconfined object. Since every unconfined object is constrained to not allow confined objects to be returned, by induction on the depth of the type no unconfined object can return a confined object.

Examples We now give a few examples of our encoding of Confined Types in *fcap*.

1. let $package =$
   (let $confined = \{f_1 = \lambda^{c_c} d.\{m_1 = \lambda^{c_{priv}} x.e\}\}$ in
   $\{d_1 = \lambda^{c_u} d.\{m_1 = check(\lambda^{c_{ok}} y.confined.f_1(0).m_1(y), [c_{ok}, c_u])\}\}$
   in $package.d_1(0).m_1(arg)$
   Here, we show how a package is used, and an unconfined object in the package uses a confined object. This is allowed, as long as no capabilities from the confined object are leaked.
2. let $package =$
   (let $confined = \{c_1 = \lambda^{c_c} d.\{m_1 = \lambda^{c_{priv}} x.e\}\}$ in
   $\{d_1 = \lambda^{c_u} d.\{m_1 = check(confined.c_1, [c_u, c_{ok}])\}\}$)
   in $package.d_1.m_1(arg)$
   This example shows a package, which leaks a confined object, and thus will fail at the check point.
3. let $package =$
   (let $confined = \{f_1 = \lambda^{c_c} d.\{m_1 = \lambda^{c_{priv}} x.e\}\}$ in
   $\{d_1 = \lambda^{c_u} d.\{m_1 = check(confined.f_1(d).m_1, [c_u, c_{ok}])\}\}$)
   in $package.d_1.m_1(arg)$
   This package leaks a method of a confined object, which is also not allowed.

6.2 Use-Based Object Confinement

The pop system [10] confines objects in a different manner, by performing access checks when objects are used, rather than when they are communicated. Since our system allows check statements to be placed anywhere in a program, we can model a use-based approach. Figure 11 gives a simplified grammar for pop, which includes the core of the language (it excludes state and other features).

$m, x, \in ID, \iota \subseteq ID$	*identifiers*
$d \in \mathcal{D}, D \subseteq \mathcal{D}$	*domains*
$\varphi \in \mathcal{D} \rightarrow 2^{ID}$	*interfaces*
$o ::= [m_i(x) = e_i \ ^{0<i\leq n}] \cdot d \cdot \varphi$	*object definitions*
$v ::= x \mid o$	*values*
$e ::= v \mid e.m(e) \mid \text{let } x = v \text{ in } \text{e}$	*expressions*

Fig. 11. Grammar for pop

Objects in pop are defined as a list of methods, $[m_i(x) = e_i \ ^{0<i\leq n}]$, together with a domain, d, and a user interface, φ. φ defines an interface for the object, which lists the methods that can be used in each domain, ie. $\varphi = \{d \mapsto \{read, write\}, \partial \mapsto \{read\}\}$. ∂ is the domain label, which matches any domain. Thus, in this instance, $read, write$ may be used in domain d, and $read$ may

be used in any domain. The interface then defines the security policy for which object methods can be used in certain domains.

We encode this use-based confinement system using the translation shown in Figure 12. The use of objects in `pop` is different from our approach, with security checks depending on domains. However, we can still model this by passing the current domain, *dom* to each method when it is called. The encoding of `pop` thus requires a global translation and is less satisfactory than the encoding of confined types.

$$
\begin{array}{rl}
[\![\, \delta, e \,]\!] & = \text{let } d = (\lambda dum.0, \delta) \text{ in } [\![\, e \,]\!] \\
[\![\, x \,]\!] & = x \\
[\![\, e_1.m(e_2) \,]\!] & = ((([\![\, e_1 \,]\!].m)\ d)\ [\![\, e_2 \,]\!] \\
[\![\, [m_1(x) = e_1, \ldots, m_n(x) = e_n] \cdot d_l \cdot \varphi \,]\!] & = \text{let } d = (\lambda dum.0, d_l) \text{ in} \\
 & \quad \{m_1 = (\!|\, [m_1(x) = e_1] \cdot d_l \cdot \varphi \,|\!), \ldots, \\
 & \quad m_n = (\!|\, [m_n(x) = e_n] \cdot d_l \cdot \varphi \,|\!), \} \\
 & \\
(\!|\, [m_i(x) = e_i] \cdot d_l \cdot \varphi \,|\!) & = (\lambda dom.(\lambda x.[\![\, e_i \,]\!], c_w), c_w) \\
\varphi = \{d_j \mapsto \{[\ldots]\}, \ldots, \partial \mapsto \{[m_i, \ldots]\}\} & \\
(\!|\, [m_i(x) = e_i] \cdot d_l \cdot \varphi \,|\!) & = (\lambda dom.(check(dom, [d_j, \ldots, d_k]); \\
\varphi = \{d_j \mapsto \{[m_i, \ldots]\}, \ldots, \{d_k \mapsto \{[m_i, \ldots]\}\} & \quad (\lambda x.[\![\, e_i \,]\!], c_w), c_w)
\end{array}
$$

Fig. 12. `pop` to *fcap* translation

Objects are encoded as records, with each field, m_i, corresponding to the object method, $m_i(x)$. We encode interfaces in our system via check statements. For every object method, m_i, we find all domains, d_j in φ, in which m_i is in the set of methods d_j maps to, ie. $d_j \mapsto \{m_i, \ldots\}$. Thus, the check for a method, m_i is $check(dom, [d_j])$, where *dom* is the current domain, and $[d_j]$ is the list of authorized domains for m_i. Each field in the record of the translation is a λ-expression, which takes the current domain as input, and performs the check on this domain. Thus, methods will be of the form $m_i = (\lambda dom.(check(dom, [d_j, d_k]); (\lambda x.[\![\, e_i \,]\!], c_w), c_w)$. If a method is included in ∂, it is allowed in any domain, and thus no check is needed, so the translation is simply $m_i = (\lambda dom.(\lambda x.[\![\, e_i \,]\!], c_w), c_w)$. Note, c_w is an unused capability.

7 Conclusion

In summary, we have defined the language, *fcap*, which considers functions as capabilities and allows the containment of capabilities at certain check-points. This language gives the programmer much flexibility in the amount of security desired in a program, allowing checks to occur at any point (or not occur at any point). These checks are done statically, during type-checking, which is clearly a beneficial feature, ensuring that a code-segment is secure before it even begins to execute.

Capabilities in *fcap* are pure capability-based. We do not allow revocation of capabilities; thus, once a capability is obtained by a module or block of code, it cannot be taken away. Narrowing, or restricting access of a capability is also not explicitly possible in *fcap*, however this can be encoded by splitting the capability into two separate capabilities. For example, a capability may be split into one which grants read access and one which grants write access to a resource.

We have given a translation from *fcap* to *fml*, an ML-like language, and type-systems for *fml* and *fcap*. We have also proved the correctness of the translation. Both the translation and type-system have been implemented, confirming the accuracy of the system.

We have shown the generality and expressiveness of our system by encoding two different object confinement systems within our own.

References

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[2] B. Bokowski and J. Vitek. Confined types. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on ObjectOriented Programming Systems, Languages, and Applications (OOPSLA '99)*, November 1999.

[3] J. Dennis and E. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[4] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA '95)*, pages 169–184. ACM Press, 1995.

[5] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. http://www.elsevier.nl/locate/entcs/volume1.html.

[6] M. Miller *et. al.* The E programming language. URL: `http://www.erights.org`.

[7] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, pages 259–270, New Orleans, LA, 1998.

[8] F. Pottier and S. Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 46–57, Montral, Canada, 2000.

[9] François Pottier. Simplifying subtyping constraints. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, volume 31(6), pages 122–133, 1996.

[10] C. Skalka and S. Smith. Static use-based object confinement. In *Foundations of Computer Security*, July 2002.

[11] Christian Skalka and Scott Smith. Set types and applications. *Electronic Notes in Theoretical Computer Science*, 75, 2003.

[12] J. Vitek and B. Bokowski. Confined types in Java. *Software—Practice and Experience*, 31(6):507–532, May 2001.

[13] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.