Lightweight Confinement for Featherweight Java

Tian Zhao Jens Palsberg[†] Jan Vitek[†]
University of Wisconsin, Milwaukee
† Purdue University

ABSTRACT

Confinement properties impose a structure on object graphs which can be used to enforce encapsulation properties essential to certain program optimizations, modular reasoning, and 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 Grothoff *et al.* [17] is 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. One need only take a step back and look at the object graphs induced by the reference relationship between objects during program execution to see that they owe more to a much maligned Italian national dish than to the result of careful engineering. The problem lies in the lack of any structure on patterns of references. While access modifiers provided by most languages can prevent some part of the system from reading protected variables, there is no syntax for constraining the spread of the objects they denote.

The last five years have seen a renewed interest (starting with [8]) in the study of ownership structures as well as related work on aliasing. The goal of these works is ambitious: extend existing language with support for object encapsulation. Some promising results suggest that it is possible to capture many interesting encapsulation properties by type systems. Clarke *et al.* were the first to propose a Java-like language with ownership types [7]. More recently, Boyapati *et al.* [3, 4] have extended these results and applied to the Real-Time Specification for Java.

Our work takes a slightly different view. Rather than going for the most expressive notion of confinement, we look for the *least disruptive* one. The rationale is that before settling on one particular notion of confinement and incorporating that in a new language design, 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, limitations on references patterns inherent to any confinement discipline will impose constraints on programs. The risk is that they become too cumbersome for programmers without providing sufficient benefit. In [17] 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 [11] 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 [6].

Two drawbacks of our previous work, highlighted by the corpus analysis, are that (a) classes can only be confined within a single package and (b) standard collection classes (such as vectors, lists) can not be used to hold confined objects. Furthermore, the informal set of rule posited in [17] was not formally shown to be correct. This paper makes the following contributions:

- Give a static type system for the rules of [17] in the context of a simple object calculus modeled on Featherweight Java [14], and prove its soundness.
- Propose a notion of generic *confined* types which addresses the two above mentioned problems.
- Give a static type system for Generic Featherweight Java with confined types, and prove its soundness.

The paper proceeds with a detailed description of confined types. Then Section 3 introduces Confined FJ (ConfinedFJ) giving it an operational semantics and a static type system. Section 4 presents our Generic Confined FJ (ConfinedFGJ). Section 5 discusses related work. Finally we conclude with some thoughts about future developments.

[1] Software referred to in this paper is available from `www.ovmj.org`.

2. CONFINED TYPES

In object-oriented programming languages such as Java or C#, confinement can be achieved by disciplined use of built-in static access control mechanisms combined with some simple coding idioms. Confinement, as defined in [11], enforces the following informal soundness property:

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

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 a core consisting of confined classes (see Figure 1). 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 annotations make the programmer's intent explicit and allow for automated checking. In the Java current realization the unit of modularity is the package. We rely on a programming idiom to identify confined classes: marker interfaces are used to label classes and marker exceptions are used to label methods. Thus we assume the following definitions.

```
package marker;

interface ConfinedClass {}

final public class AnonymousMethod
      extends Error {}
```

Consider the following example. A class `Bucket` is used to implementation of a hash table class, `mine.HTable`. Hash table buckets are an 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`.

```
package mine;

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

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

But what if one of `HTable`'s public methods, say `get()`, were to return a `Bucket` instance or store a reference to a bucket in one of its public fields? One can 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 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 keep buckets of one hash table from being mixed up with those of another [7], but that would place a considerably greater burden on programmers.

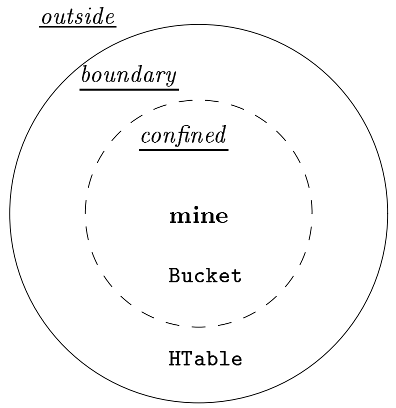

Figure 1: A package consist of a confined core and a boundary of public classes visible to the outside. Operations on a confined class can only be initiated by a classes belonging to the same package.

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.

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, and ensures that these methods do not leak a reference to the distinguished variable `this` which may refer to an object of confined type. There are two design choices for dealing with inherited methods either disallow inheritance or restrict it to a safe subset. We have chosen the later; anonymous method are well-behaved in the sense that they can not 'leak' their `this` pointer. 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 [11] with some of implementation specific details omitted[2].

Anonymity Rules. Anonymity rules apply to inherited methods which may reside in classes outside of the enclos-

[2] We do not discuss native methods, reflection, and open packages as these are not germane to the formalization. The reader is referred to [11] for a discussion of those features and associated restrictions.

ing package. These rules prevent a method from leaking the `this` reference. A method is *anonymous* if it has the following property.

$\mathcal{A}1$	The `this` reference is only used to select fields and as receiver in invocation of other anonymous methods.

This prevents an inherited method from storing or returning `this` as well as using it as an argument to a call. Selecting a field is always safe, it can not break anonymity because only the fields visible in the current class can be accessed. Method invocation (on `this`) is restricted to other methods that are anonymous as well.

Confinement Rules. The following confinement rules must hold for all classes of a package containing confined types.

$\mathcal{C}1$	Methods invoked on a confined object must either be defined in a confined class of the same package, or be anonymous methods.
$\mathcal{C}2$	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.
$\mathcal{C}4$	Subtypes of a confined type must be confined.
$\mathcal{C}5$	Confined types can only be widened to other types confined in the same package.
$\mathcal{C}6$	Overriding must preserve the anonymity of methods.

Rules $\mathcal{C}1$ and $\mathcal{C}6$ ensure 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[3]. 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]We note that rule $\mathcal{C}4$ is not strictly needed for enforcing confinement. Rule $\mathcal{C}5$ is sufficient to ensure against leaks even if subtyping is allowed. We retain $\mathcal{C}4$ only because it will prove useful for generics.

2.1 Empirical evaluation with Kacheck/J

The Kacheck/J static analysis tool [11] was implemented to test our assumptions about confinement in real life setting. Its goal was to help understand the properties of large, well-engineered, systems. Kacheck infers confinement on unannotated Java programs by an efficient analysis of the bytecode. Since client code is not required when checking confinement, it is possible to use Kacheck on libraries. We have extended the Purdue Benchmark Suite (PBS) to 38 Java systems of varying size, purpose and origin. The entire suite contains 101,893 classes (or 282 MB of bytecode), 8,920,978 methods and 4,564 packages. To the best of our knowledge the PBS is the largest such collection of Java programs freely available (it can be obtained from `www.ovmj.org`). The results gleaned from the analysis are encouraging as 6,630 confined types occur in plain Java programs, Figure 2 gives the breakdown per program or library. We surmise that these numbers would be even higher if programmers actually designed code with confinement in mind. Another important statistic is that 48% of all methods (some 4,039,281) are anonymous which suggests that significant chunks of behavior can be safely inherited.

Generics. In Java programs, vectors, hashtables and other container type are omnipresent and every time an object is stored in a container, its type is widened to `Object` leading to a violation of the confinement rule preventing widening. We posit that if Java supported proper parametric polymorphism, the large majority of these violations would disappear – while truly heterogeneous data structures do exist, they seem to be the exception. If we tune our analysis to ignore widening violations linked to containers, which can be done by simply ignoring all widenings to `Object` that occur in calls to methods of classes `java.util`, 2,000 additional classes become confined. Figure 2 gives the generic confinement numbers for our new benchmark suite.

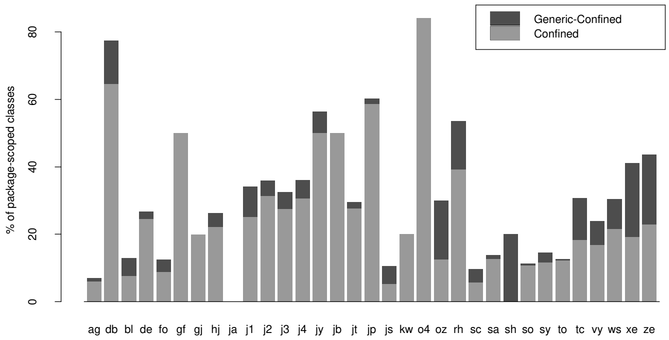

Figure 2: The Purdue Benchmark Suite. For each program, we give the ratio of confined classes and an estimate of classes that would become confined with generics.

2.2 Confinement v. Reuse

From a practical perspective, one criticism of confined types is that they seem to preclude code reuse. For a class to be confined it must be local to a particular package and, by definition, inaccessible to all other packages. This can become unwieldy when dealing with programs that require the same logic to be available in, and confined to, different packages. While the issue was addressed in [17], a truly satisfactory remained elusive.

Any solution to the reuse problem must allow to define classes in a natural fashion, *i.e.* without imposing coding conventions more restrictive than those presented above, and must permit use of those classes as confined types in certain contexts and non-confined in other. Previous work failed to provide a satisfactory solution to this problem.

Summary. This section gave an overview of confined types. While our benchmark results have provided supporting evidence to our claim that confined types are a natural construction in object oriented systems, we are left with the following questions. Is it possible to define a variant of the confinement property that would deal with collection classes? Can we solve the reuse problem alluded to above? Moreover, until now our work has been entirely informal, can we make any soundness claims?

3. CONFINED FEATHERWEIGHT JAVA

Featherweight Java, or FJ, is core object calculus that was developed by Igarashi, Pierce and Wadler [14] 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 Confined FJ, has been modified as little as possible. In particular, we have resisted the temptation to add assignment[4]. The syntax appears on Figure 3. 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, though adding access modifiers to fields would not be a major issue.

Consider the following ConfinedFJ program in which a public class `Table`, defined in `my` package, contains a linked list of `Bucket` objects. The `Bucket` class also belongs to `my` package but is declared confined. The program is invalid as the body of the `get()` method of `Table` returns an instance of the confined class `Bucket`, which 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(); } }

+ class my.Factory extends Object {
    Factory() { super(); } }
    Table table() {
       return new Table( new Bucket() ); } }
```

In the above, the `Factory` class is needed to allow code outside of the `my` package to create new tables with an initial bucket (since class `Bucket` is package scoped this is the only way to use the hash table from the outside). 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 Factory().table().get() );
```

evaluates in two reduction steps to the following expression

```
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(); } }
```

[4]It may appear paradoxical that we are trying to address aliasing issues in a calculus which does not have references. But one of the insights from our previous work is that confinement inference is only dealing with the propagation of types within the program. All values of a type are treated as equal and the confinement rules ensure that values of a confined type are never confused with other values. Dealing with references is, of course, important in ownership systems where the unit of encapsulation is finer grained.

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 of ConfinedFJ is shown in Figure 3. We use metavariables L to range over class declarations, 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, $\bar{\mathtt{f}}$ represents $\mathtt{f}_1, \mathtt{f}_2, \ldots, \mathtt{f}_n$. We use P, Q to range over package names and we use $\rho_{\mathtt{C}}$ 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 $CT(\mathtt{C})$ is the definition of class C.

We use e, d to range over expressions and u, v, w to range over *fully-evaluated* objects of the form new $\mathtt{C}(\bar{\mathtt{v}})$. An *expression* e can be either a variable x, the this pseudo variable, a field access e.f, a method invocation $\mathtt{e.m}(\bar{\mathtt{e}})$, a cast (C)e, an object new $\mathtt{C}(\bar{\mathtt{e}})$.

The subtyping relation $\mathtt{C} <: \mathtt{D}$ denotes that class C is a subtype of class D, irrespective of access mode and or package, if class C transitively extends D. We take the simplifying assumption that class names are unique.

The definitions and auxiliary functions used in our semantics and typing rules are listed in Figures 3 and 4. The predicate *visible*(C, 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 $\mathtt{C} \preceq \mathtt{D}$ 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.

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 $\mathtt{e} \rightarrow \mathtt{e}'$. 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 new $\mathtt{C}(\bar{\mathtt{v}})$. While this is superficially closer to a real 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 5, and they check for reference widening of types. In the typing rules, the condition of the form $\mathtt{C} \preceq \mathtt{D}$ 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. The expression typing rules emulate the effects of the confinement rules $\mathcal{C}1$ and $\mathcal{C}5\dot{\text{I}}$n 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 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.

The definition of anonymous method is given in Figure 4. 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 the typing rule for methods, the return expression of a method m defined in class C must be visible in C, that is, the type of every subexpression in m must be visible in C. The effects of the constraint are that

- a method that returns expression of confined type C can not be invoked outside the package where C is defined;
- a field of confined 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 C is defined.

Syntax:

$$\begin{array}{lcl}
\circ & ::= & \mathtt{+} \mid \mathtt{-} \\
\mathtt{L} & ::= & \circ\, \mathtt{class}\ \mathtt{P.C}\ \triangleleft\ \mathtt{D}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \} \\
\mathtt{K} & ::= & \mathtt{C}(\overline{\mathtt{C}}\ \overline{\mathtt{f}})\ \{\ \mathtt{super}(\overline{\mathtt{f}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\ \} \\
\mathtt{M} & ::= & \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return}\ \mathtt{e};\ \} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{this} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid (\mathtt{C})\ \mathtt{e} \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) \\
\mathtt{v} & ::= & \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})
\end{array}$$

Subtyping:

$$\mathtt{C} <: \mathtt{C} \qquad \frac{\mathtt{C} <: \mathtt{D} \quad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}}$$

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

Computation:

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

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

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

Congruence:

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

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

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

Other definitions:

$$\frac{CT(\mathtt{C}) = \mathtt{-}\ \mathtt{class}\ \mathtt{P.C}\ \triangleleft\ \mathtt{D}\ \{\ \ldots\ \}}{\mathit{conf}(\mathtt{C})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{+}\ \mathtt{class}\ \mathtt{P.C}\ \triangleleft\ \mathtt{D}\ \{\ \ldots\ \}}{\mathit{public}(\mathtt{C})}$$

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

$$\frac{\mathtt{C} <: \mathtt{D} \quad \mathit{public}(\mathtt{C}) \vee (\mathit{conf}(\mathtt{D}) \wedge \rho_\mathtt{C} = \rho_\mathtt{D})}{\mathtt{C} \preceq \mathtt{D}}$$

Figure 3: Confined FJ

Field look-up:

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

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

Method definition lookup:

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

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

Method type lookup:

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

Method body look-up:

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

Valid method overriding:

$$\frac{\mathit{mtype}(\mathtt{m},\ \mathtt{C}_0') = \overline{\mathtt{D}} \rightarrow \mathtt{D} \ \Rightarrow\ \mathtt{C}, \overline{\mathtt{C}} = (\mathtt{D}, \overline{\mathtt{D}}) \quad \mathit{anon}(\mathtt{m}, \mathtt{C}_0') \ \Rightarrow\ \mathit{anon}(\mathtt{m}, \mathtt{C}_0)}{\mathit{override}(\mathtt{m}, \mathtt{C}_0, \mathtt{C}_0', \overline{\mathtt{C}} \rightarrow \mathtt{C})}$$

Anonymous method:

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

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

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

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

Figure 4: Auxiliary Definitions

The visibility constraint of method body is meant to simplify our typing rules and it is not useful in practice since it requires checking the whole program. The confinement rules $\mathcal{C}2$ and $\mathcal{C}3$achieve the same effect and they can be checked locally within the defining packages of the confined types.

The confinement rule $\mathcal{C}4$ requires that a confined class be inherited only by the confined classes defined in the same package. We find such a constraint unnecessary in proving the main results of confinement properties. Even if a confined class `C` is inherited by a confined class `D` in another package, it is not possible for an object o of type `D` to be widened to the type `C` since our typing rules require o to be types confined in the package of `D`. It is possible for the public subclasses of a confined class to inherit fields of confined types or methods that return confined types, but those fields and methods are not accessible to the subclasses by Rule $\mathcal{C}3$Thus, confined objects can not be leaked outside their defining package through inheritance.

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.

The two examples in the beginning of the section are not typable according to the typing rules. In the first example, the method `get` of the class `my.Table` does not type-check because Rule T-METHOD requires that if the type of the return expression of a method is confined than the return type of the method must be confined as well. The type of the return expression of the `get` method is `my.Bucket` while the return type of the method is `Object`. Since `Bucket` is confined and `Object` is public, we have a violation of the typing rule T-METHOD.

The second example does not type-check because the confined class `my.Self` inherits a non-anonymous method `reveal` from the class `outside.Broken` and the method `reveal` is called on a `my.Self` object in the `get` method of the class `my.Ifc`. The method `reveal` violates the typing rule T-INVK because the rule requires that all methods invoked on an object of confined type is either anonymous or defined in the same package as the confined type. Notice that the method `reveal` is not anonymous because the variable `this` is returned while according to the rules for anonymous methods in Figure 4, the variable `this` can only be used for field access or invocation of anonymous methods in the return expression.

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 *only methods defined in* `P` *or anonymous methods inherited by* `C` *can access the fields and methods of* o.

We prove the property in Theorem 7. We show that for a well-typed method call expression $\mathtt{new}\ \mathtt{C_0}(\overline{\mathtt{u}}).\mathtt{m}(\overline{\mathtt{v}})$, 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 $\mathtt{new}\ \mathtt{C_0}(\overline{\mathtt{u}})$ is accessible for field select and invocation of anonymous methods. We also prove the usual subject reduction lemma and state the progress lemma. For the subject reduction lemma, we show

Expression typing:

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

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C_0} \quad \mathit{fields}(\mathtt{C_0}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}})}{\Gamma \vdash \mathtt{e.f_i} : \mathtt{C_i}} \qquad \text{(T-FIELD)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{e} : \mathtt{C_0} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \\ \mathit{mtype}(\mathtt{m},\ \mathtt{C_0}) = \overline{\mathtt{D}} \rightarrow \mathtt{C} \\ \overline{\mathtt{C}} \preceq \overline{\mathtt{D}} \quad \mathit{mdef}(\mathtt{m},\ \mathtt{C_0}) = \mathtt{D_0} \\ \mathtt{C_0} \preceq \mathtt{D_0} \vee \mathit{anon}(\mathtt{m}, \mathtt{D_0})\end{array}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(T-INVK)}$$

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

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{e} : \mathtt{D} \\ \mathit{public}(\mathtt{D}) \vee (\mathit{conf}(\mathtt{C}) \wedge \rho_{\mathtt{C}} = \rho_{\mathtt{D}})\end{array}}{\Gamma \vdash (\mathtt{C})\ \mathtt{e} : \mathtt{C}} \qquad \text{(T-CAST)}$$

Method typing:

$$\frac{\begin{array}{l}\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C_0} \vdash \mathtt{e} : \mathtt{D} \quad \mathtt{D} \preceq \mathtt{C} \\ \mathit{override}(\mathtt{m}, \mathtt{C_0}, \mathtt{D_0}, \overline{\mathtt{C}} \rightarrow \mathtt{C}) \\ \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C_0} \vdash \mathit{visible}(\mathtt{e}, \mathtt{C_0})\end{array}}{\mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\mathtt{return}\ \mathtt{e};\ \}\ \mathtt{OK\ IN}\ \mathtt{C_0} \triangleleft \mathtt{D_0}} \qquad \text{(T-METHOD)}$$

Class typing:

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

Expression visibility:

$$\frac{\Gamma \vdash \mathtt{x} : \mathtt{C} \quad \mathit{visible}(\mathtt{C}, \mathtt{C_0})}{\Gamma \vdash \mathit{visible}(\mathtt{x}, \mathtt{C_0})} \qquad \text{(V-VAR)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C_0}) \\ \Gamma \vdash \mathtt{e.f_i} : \mathtt{C} \quad \mathit{visible}(\mathtt{C}, \mathtt{C_0})\end{array}}{\Gamma \vdash \mathit{visible}(\mathtt{e.f_i}, \mathtt{C_0})} \qquad \text{(V-FIELD)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{C} \quad \mathit{visible}(\mathtt{C}, \mathtt{C_0}) \\ \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C_0}) \quad \Gamma \vdash \mathit{visible}(\overline{\mathtt{e}}, \mathtt{C_0})\end{array}}{\Gamma \vdash \mathit{visible}(\mathtt{e.m}(\overline{\mathtt{e}}), \mathtt{C_0})} \qquad \text{(V-INVK)}$$

$$\frac{\mathit{visible}(\mathtt{C}, \mathtt{C_0}) \quad \Gamma \vdash \mathit{visible}(\overline{\mathtt{e}}, \mathtt{C_0})}{\Gamma \vdash \mathit{visible}(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}), \mathtt{C_0})} \qquad \text{(V-NEW)}$$

$$\frac{\mathit{visible}(\mathtt{C}, \mathtt{C_0}) \quad \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C_0})}{\Gamma \vdash \mathit{visible}((\mathtt{C})\ \mathtt{e}, \mathtt{C_0})} \qquad \text{(V-CAST)}$$

Figure 5: Typing Rules

that an expression of public type can not be reduced to an expression of confined type. In Theorem 6, we prove that a well-type expression will not get stuck and an expression of public type will not be reduced to a value of confined type.

LEMMA 1. **Subject Reduction:** *If* $\emptyset \vdash \texttt{e} : \texttt{C}$ *and* $\texttt{e} \to \texttt{e}'$ *then* $\emptyset \vdash \texttt{e}' : \texttt{C}'$ *for some* $\texttt{C}' \preceq \texttt{C}$.

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

$$\begin{array}{rcl} \texttt{e}_0 & = & \texttt{new}\ \texttt{C}_0(\overline{\texttt{v}}).\texttt{f}_\texttt{i} \\ & | & (\texttt{C})\ \texttt{new}\ \texttt{C}'(\overline{\texttt{v}}),\ \texttt{C}' <: \texttt{C}, \\ & | & \texttt{new}\ \texttt{C}_0(\overline{\texttt{u}}).\texttt{m}(\overline{\texttt{v}}), \end{array}$$

then there exists $\texttt{e}' \neq \texttt{e}$ *such that* $\texttt{e} \to \texttt{e}'$.

LEMMA 3. *If* $mtype(\texttt{m},\ \texttt{C}_0) = \overline{\texttt{D}} \to \texttt{D}$, $mdef(\texttt{m},\ \texttt{C}_0) = \texttt{D}_0$, *and* $mbody(\texttt{m},\ \texttt{C}_0) = (\overline{\texttt{x}},\ \texttt{e})$, *then there exists some* $\texttt{C} \preceq \texttt{D}$ *such that* $\overline{\texttt{x}} : \overline{\texttt{D}}, \texttt{this} : \texttt{D}_0 \vdash \texttt{e} : \texttt{C}$.

The following two lemmas prove term substitution preserves typing for expressions in anonymous methods and nonanonymous methods.

LEMMA 4. *If* $\overline{\texttt{x}} : \overline{\texttt{B}}, \texttt{this} : \texttt{D}_0 \vdash \texttt{e} : \texttt{D}$, $\emptyset \vdash \overline{\texttt{v}} : \overline{\texttt{A}}$, $\overline{\texttt{A}} \preceq \overline{\texttt{B}}$, $\emptyset \vdash \texttt{new}\ \texttt{C}_0(\overline{\texttt{u}}) : \texttt{C}_0$, $\texttt{C}_0 <: \texttt{D}_0$, *and* $anon(\texttt{e}, \texttt{D}_0)$, *then* $\emptyset \vdash [\overline{\texttt{v}}/\overline{\texttt{x}},\ {}^{\texttt{new}\ \texttt{C}_0(\overline{\texttt{u}})}/_{\texttt{this}}]\texttt{e} : \texttt{C}$ *for some* $\texttt{C} \preceq \texttt{D}$.

LEMMA 5. *If* $\overline{\texttt{x}} : \overline{\texttt{B}} \vdash \texttt{e} : \texttt{D}$, $\emptyset \vdash \overline{\texttt{v}} : \overline{\texttt{A}}$, $\overline{\texttt{A}} \preceq \overline{\texttt{B}}$. *then* $\emptyset \vdash [\overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e} : \texttt{C}$ *for some* $\texttt{C} \preceq \texttt{D}$

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

THEOREM 6. *If* $\emptyset \vdash \texttt{e} : \texttt{C}$ *and* $\texttt{e} \to^* \texttt{e}'$, *then* $\texttt{e}'$ *is not stuck, and there exists* $\texttt{C}'$ *such that* $\emptyset \vdash \texttt{e}' : \texttt{C}'$ *where* $\texttt{C}' \preceq \texttt{C}$.

PROOF. Immediate from Lemma 1 and 2. □

Intuitively, the confinement theorem states that all expressions that are manipulated within the body of a method either evaluate to methods that are visible from the method's defining package. The only exception is for anonymous methods, as they may have access to `this` which can evaluate to an instance of a class confined in another package, if this occur the method must be anonymous and the use of `this` is restricted to the receiver position.

THEOREM 7. **Confinement.** *If* $\texttt{e} = \texttt{new}\ \texttt{C}_0(\overline{\texttt{u}}).\texttt{m}(\overline{\texttt{v}})$, $\emptyset \vdash \texttt{e} : \texttt{D}$, $mdef(\texttt{m},\ \texttt{C}_0) = \texttt{C}$ *and* $\texttt{e} \to d$, *then for every subexpression* $\texttt{d}'$ *of* d *such that* $\texttt{d}' \to^* \texttt{new}\ \texttt{D}_0(\overline{\texttt{u}}')$ *we have:*

1. $visible(\texttt{D}_0, \texttt{C})$, *or*

2. $\texttt{d}' = \texttt{new}\ \texttt{C}_0(\overline{\texttt{u}})$, m *is anonymous and* $\texttt{d}'$ *appears in a subexpression of the form* $\texttt{d}'.\texttt{f}$ *or* $\texttt{d}'.\texttt{m}'(\overline{\texttt{e}})$.

PROOF. Suppose method m is such that $mtype(\texttt{m},\ \texttt{C}_0) = \overline{\texttt{D}} \to \texttt{D}$, $mdef(\texttt{m},\ \texttt{C}_0) = \texttt{C}$, and also $mbody(\texttt{m},\ \texttt{C}) = (\overline{\texttt{x}},\ \texttt{e}_0)$. Let $\Gamma = \overline{\texttt{x}} : \overline{\texttt{D}}, \texttt{this} : \texttt{C}$. By Rule T-INVK, if $\emptyset \vdash \overline{\texttt{v}} : \overline{\texttt{D}}'$, then $\overline{\texttt{D}}' \preceq \overline{\texttt{D}}$; and either $\texttt{C}_0 \preceq \texttt{C}$ or $anon(\texttt{m}, \texttt{C})$. By Lemma 3, we have that $\Gamma \vdash \texttt{e}_0 : \texttt{D}$. Let $\texttt{d}'$ be any subexpression of d, there exists a subexpression $\texttt{e}'$ of $\texttt{e}_0$ such that $\texttt{d}' = [\overline{\texttt{v}}/\overline{\texttt{x}},\ {}^{\texttt{new}\ \texttt{C}_0(\overline{\texttt{u}})}/_{\texttt{this}}]\texttt{e}'$. Since $\Gamma \vdash \texttt{e}_0 : \texttt{D}$, there exists $\texttt{D}'$ such that $\Gamma \vdash \texttt{e}' : \texttt{D}'$.

By Rule T-METHOD, we have $\Gamma \vdash visible(\texttt{e}_0, \texttt{C})$ and by simple induction, we can show that $\Gamma \vdash visible(\texttt{e}', \texttt{C})$ which implies $visible(\texttt{D}', \texttt{C})$.

If $\texttt{C}_0 \preceq \texttt{C}$, then from $\overline{\texttt{D}}' \preceq \overline{\texttt{D}}$ and Lemma 5, there exists $\texttt{D}''$ such that $\emptyset \vdash \texttt{d}' : \texttt{D}''$ and $\texttt{D}'' \preceq \texttt{D}'$, By Lemma 1, if $\texttt{d}' \to^* \texttt{new}\ \texttt{D}_0(\overline{\texttt{u}}')$, then $\texttt{D}_0 \preceq \texttt{D}''$. Together with $\texttt{D}'' \preceq \texttt{D}'$ and $visible(\texttt{D}', \texttt{C})$, we have $visible(\texttt{D}_0, \texttt{C})$.

If we have $anon(\texttt{m}, \texttt{C})$, then by definition, we have $anon(\texttt{e}_0, \texttt{C})$ as well. By the definition of anonymous expression, either $anon(\texttt{e}', \texttt{C})$ or $\texttt{e}' = \texttt{this}$ appears in a subexpression of the form this.f or $\texttt{this}.\texttt{m}'(\overline{\texttt{e}})$. If $anon(\texttt{e}', \texttt{C})$, then from $\texttt{C}_0 <: \texttt{C}$, $\overline{\texttt{D}}' \preceq \overline{\texttt{D}}$, and Lemma 4, there exists $\texttt{D}''$ such that $\emptyset \vdash \texttt{d}' : \texttt{D}''$ and $\texttt{D}'' \preceq \texttt{D}'$. By Lemma 1, if $\texttt{d}' \to^* \texttt{new}\ \texttt{D}_0(\overline{\texttt{u}}')$, then $\texttt{D}_0 \preceq \texttt{D}''$. Similarly, we have $visible(\texttt{D}_0, \texttt{C})$.

Therefore, if $\texttt{d}' \to^* \texttt{new}\ \texttt{D}_0(\overline{\texttt{u}}')$ then either $visible(\texttt{D}_0, \texttt{C})$ or $\texttt{d}' = \texttt{new}\ \texttt{C}_0(\overline{\texttt{u}})$ appears in a subexpression of the form $\texttt{d}'.\texttt{f}$ or $\texttt{d}'.\texttt{m}'(\overline{\texttt{e}})$. □

This result ensures that any well-typed program consisting of a top-level method invocation expression will preserve confinement.

4. GENERICS AND CONFINEMENT

The lack of support for collections and the reuse problem were identified early on as significant issues limiting the applicability of confined types. In this section we show how to extend the confinement property to generic types. In particular, how to write a generic class $\texttt{C}\langle\texttt{X}\rangle$ which can be used to define generic collection classes that are, in and of themselves, not confined. But that become confined if instantiated with confined arguments.

Following the lead of Featherweight Generic Java, ConfinedFJ is extended with support for generic types and renamed ConfinedFGJ. The only departure from our role model is that in ConfinedFGJ, type variables are annotated with *confinability* tags. So for some generic type $\texttt{C}\langle\texttt{X}\rangle$, the type variable X can be either *public* $\texttt{X}^\texttt{p}$ or *confined* $\texttt{X}^\texttt{c}$. The semantics of these annotations is that a public type variable can only be instantiated with a public type, while a confined type variable *may* be instantiated with a confined class (or a public class). Besides the first six rules already presented, we require:

$\mathcal{C}7$	A generic type or type variable can not be widened to a type containing a different set of confined type variables.
$\mathcal{C}8$	A confined type can not replace a public type variable in the instantiation of a generic type.

Rules $\mathcal{C}5$ and $\mathcal{C}7$ correpond to the subtyping partial order that prevents reference widening for ConfinedGFJ. Rule $\mathcal{C}8$ corresponds to the extra requirement in the definition of well-formed generic types.

Consider a generic linked list class. If we desire to allow the class to be used to hold confined objects, it should be defined with a confined type variable in place of its value type. The following is likely definition.

```
+ class some.List<X^c extends Object>
      extends Object {
    X val;
    List<X> next;
    List(X val, List next) {
      super(); this.val=val; this.next=next;
    }
}
```

Where things get interesting is that we can now write code that uses lists in several contexts. Thus it is possible to use the list twice within the same package, once with a confined type – thus turning the list itself into a confined type – and once with a non confined type. The following example illustrates this. Classes `A` and `B` reside in `my` package, the latter is confined. Class `A` further defines two variables: `show` holds a list of `A` objects and `hide` holds a list of `B` objects. Since `B` is confined the type `List<B>` will be confined as well.

```
+ class my.A extends Object {
    List<A> show;
    List<B> hide;
    ...}

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

Generics can be used to address the reuse problem with a slight of hand. If a class needs to be reused across different packages and confined in each of these packages one may simply give the class a dummy type variable annotated as confined. This type variable need not be used in the body of the class, it will merely serve as a marker. Reuse is obtained by instantiating the class in each of the packages with a dummy confined class as argument. Consider the following scenario, a class `Key` is meant to provide functionality that can be used in different confined settings.

```
+ class a.Key<X^c extends Object> extends Object {
    ...
}
```

The type variable `X` need not used in the implementation of `Key`. The class can be confined in any package as long as it is instantiated with a confined type, *e.g.* `new Key<my.B>()`.

Generic types that may be instantiated with a generic argument are referred to as *confinable* generic types. The semantics of confinable generic types are surprisingly simple. Any type variable that is tagged as confined will be treated as a confined type by the type system and reference widening will be forbidden for expressions of this type. Even though a confinable generic type may not be confined in any package, reference widening should not be allowed for expressions of the type either.

For example, consider a generic container class.

```
+ class some.Container<X^c extends Object>
      extends Object {
    X val;
    Container(X val) {this.val = val}
    Object get() { return this.val; }
    Object get2() { return this; }
}
```

The `Container` class has a method `get` that returns the field `val` and a method `get2` that returns the pointer `this`. Both methods violate the confinement properties because the types of the return expressions in `get` and `get2` are widened from `X` and `Container⟨X⟩` to `Object` respectively. Since `X` is tagged as confined type variables, it may be replaced by confined types when `Container` is instantiated as in the class `my.A`.

```
+ class my.A extends Object {
    Container<B> f = new Container<B>(new B());
    Object reveal() {
      return f.get();
    }
    Object reveal2() {
      return f.get2();
    }
}
```

The class `my.A` has a field `f` that points to a `Container` object instantiated with a confined object of the type `my.B`. The method `reveal` calls the `get` method on the `Container` object. Apparently, the type of the expression `new my.B()` is widened from `my.B` to the type `Object` by the method `reveal` because of the call to the `get` method. The method `reveal2` also violates confinement properties because it widens confined object of the type `Container<my.B>` to `Object`. Thus, our typing rules disallow methods such as `get` and `get2` in classes of confinable generic types to prevent widening of references to confined objects.

Any confinable generic type may be confined in more than one packages, For instance, if a class `P.A⟨X⟩` contains a reference to an object of the type `Q.B⟨P.C, X⟩`, where `P.C` is a confined type in P, then $\mathtt{Q.B}\langle\mathtt{P.C},\mathtt{X}\rangle$ should be confined in P. If the class $\mathtt{P.A}\langle\mathtt{X}\rangle$ is instantiated with a confined type $\mathtt{P'.C'}$, then an object of the type $\mathtt{P.A}\langle\mathtt{P'.C'}\rangle$ is confined in $\mathtt{P'}$ and it also contains a reference to an object of the type $\mathtt{Q.B}\langle\mathtt{P.C},\mathtt{P'.C'}\rangle$, which is now confined in both P and $\mathtt{P'}$.

In the previous section we argued that the rule limiting subtyping of confined types was redundant. This is not the case when generics are added to ConfinedFJ. $\mathcal{C}4$ is needed since a generic class may contain fields of confined types. However, we can not force all fields of confined variable types to be declared private or default as this would make such fields inaccessible when the generic class is instantiated in other packages with non-confined arguments types. This is undesirable as it limits the ability to reuse generic classes.

4.1 Syntax

The syntax for generic class is shown in Figure 6. For simplicity, we omit the parameterized types in methods. We use X, Y to represent type variables and we use N, W to represent types other than type variables; We use S, T to range over types and type variables. The type variable X in a generic class can be either *public* $\mathtt{X}^{\mathtt{p}}$ or *confined* $\mathtt{X}^{\mathtt{c}}$. In a class definition of the form $\circ$ **class** $\mathtt{P.C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle \lhd \mathtt{N}\,\{\ \dots\ \}$, $\overline{\mathtt{N}}$ are the upper bounds for the type variables $\overline{\mathtt{X}}$

Syntax:

$$\circ \ ::= \ \mathtt{+}\ |\ \mathtt{-},\quad \mathtt{N} ::= \mathtt{C}\langle\overline{\mathtt{T}}\rangle,\quad \mathtt{T} ::= \mathtt{X}\ |\ \mathtt{N},\quad \mathtt{X} ::= \mathtt{X}^{\mathtt{c}}\ |\ \mathtt{X}^{\mathtt{p}}$$

$$\mathtt{L} \ ::= \ \circ\ \mathbf{class}\ \mathtt{P.C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle \lhd \mathtt{N}\,\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}$$

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

$$\mathtt{M} \ ::= \ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\,\{\,\mathtt{return}\,\mathtt{e};\ \}$$

$$\mathtt{e} \ ::= \ \mathtt{x}\ |\ \mathtt{this}\ |\ \mathtt{e.f}\ |\ \mathtt{e.m}(\overline{\mathtt{e}})\ |\ (\mathtt{N})\ \mathtt{e}\ |\ \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}})$$

$$\mathtt{v} \ ::= \ \mathtt{new}\ \mathtt{N}(\overline{\mathtt{v}})$$

Figure 6: Syntax

4.2 Typing

Figure 7 contains subtyping rules, definitions for well-formed types, and other miscellaneous definitions. The subtyping rules are the same as those in Generic FJ.

A type $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$ is public iff C is public and $\overline{\mathtt{T}}$ are public types or public type variables. We use $\mathit{conf}(\mathtt{N},\mathtt{P})$ to denote that the type N is confined in the package P. A type $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$ is confined in P iff C is confined in P or there exists a type $\mathtt{N}\in\overline{\mathtt{T}}$ such that $\mathit{conf}(\mathtt{N},\mathtt{P})$ is true. Note that a confinable generic type may not be be confined in any package. However, it can become confined in some packages after instantiation. Thus, it is necessary to prevent reference widening for expressions of confinable generic types. A type N is confined iff there exists P such that N is confined in P.

If the type $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$ is well-formed, then for all i, either $\mathtt{T}_{\mathtt{i}}$ is public or the type variable it replaces must be confined. This restriction prevents reference widening when generic types are instantiated.

Subtyping:

$$\Delta \vdash \mathtt{T} <: \mathtt{T}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \quad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}}$$

$$\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})$$

$$\frac{\mathit{CT}(\mathtt{C}) = \circ\ \mathbf{class}\ \mathtt{P.C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle \lhd \mathtt{N}\,\{\ \dots\ \}}{\Delta \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}}$$

Public, Confined Types and Type Variables:

$$\mathit{conf}(\mathtt{X}^{\mathtt{c}}) \qquad \mathit{public}(\mathtt{X}^{\mathtt{p}})$$

$$\frac{(\mathit{conf}(\mathtt{C}) \wedge \rho_{\mathtt{C}} = \mathtt{P})\ \vee\ \exists\mathtt{T}\in\overline{\mathtt{T}},\ \mathit{conf}(\mathtt{T},\mathtt{P})}{\mathit{conf}(\mathtt{C}\langle\overline{\mathtt{T}}\rangle,\mathtt{P})}$$

$$\frac{\mathit{conf}(\mathtt{C}\langle\overline{\mathtt{T}}\rangle,\mathtt{P})\ \text{for some}\ \mathtt{P}}{\mathit{conf}(\mathtt{C}\langle\overline{\mathtt{T}}\rangle)}$$

$$\frac{\mathit{public}(\mathtt{C})\ \wedge\ \forall\mathtt{T}\in\overline{\mathtt{T}},\ \mathit{public}(\mathtt{T})}{\mathit{public}(\mathtt{C}\langle\overline{\mathtt{T}}\rangle)}$$

Well-formed Types:

$$\Delta \vdash \mathtt{Object}$$

$$\frac{\mathtt{X} \in \mathrm{dom}(\Delta)}{\Delta \vdash \mathtt{X}}$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \circ\ \mathbf{class}\ \mathtt{P.C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle \lhd \mathtt{N}\,\{\ \dots\ \}\\ \Delta \vdash \overline{\mathtt{T}} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}\\ \forall\mathtt{i},\ \mathit{public}(\mathtt{T}_{\mathtt{i}}) \vee \mathit{conf}(\mathtt{X}_{\mathtt{i}})\end{array}}{\Delta \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle}$$

Set of Confined Type Variables in Types:

$$\mathit{ConfVar}(\mathtt{X}) = \{\mathtt{X}\ \mid\ \mathit{conf}(\mathtt{X})\}$$

$$\mathit{ConfVar}(\mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \bigcup_{\forall\mathtt{T}\in\overline{\mathtt{T}}} \mathit{ConfVar}(\mathtt{T})$$

Subtyping without Reference Widening:

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{S} <: \mathtt{T} \quad \mathit{conf}(\mathtt{S},\mathtt{P}) \Rightarrow \mathit{conf}(\mathtt{T},\mathtt{P})\\ \mathit{ConfVar}(\mathtt{S}) = \mathit{ConfVar}(\mathtt{T})\end{array}}{\Delta \vdash \mathtt{S} \preceq \mathtt{T}}$$

Visibility of Types and Type Variables

$$\mathit{visible}(\mathtt{X},\mathtt{C}) \qquad \frac{\mathit{conf}(\mathtt{N},\mathtt{P}) \Rightarrow \mathtt{P} = \rho_{\mathtt{C}}}{\mathit{visible}(\mathtt{N},\mathtt{C})}$$

$$\frac{\mathit{conf}(\mathtt{N},\mathtt{P}) \Rightarrow (\mathtt{P} = \rho_{\mathtt{C}} \vee \mathit{conf}(\mathtt{C}\langle\overline{\mathtt{N}}\rangle,\mathtt{P}))}{\mathit{visible}(\mathtt{N},\mathtt{C}\langle\overline{\mathtt{N}}\rangle)}$$

Figure 7: Subtyping Rules, Well-formed Types, and Miscellaneous Definitions

As in confined FJ, we define a partial order $\preceq$ on types to represent subtying relation without reference widening. To have $\Delta \vdash \mathtt{S} \preceq \mathtt{T}$, we must have $\Delta \vdash \mathtt{S} <: \mathtt{T}$ and if S is confined in package P then so is T; also, S, T must contain the same set of confined type variables. With the last restriction,

Bound of type:

$$bound_\Delta(\mathtt{X}) = \Delta(\mathtt{X}) \quad bound_\Delta(\mathtt{N}) = \mathtt{N}$$

Field look-up:

$$fields(\mathtt{Object}) = ()$$

$$\frac{CT(\mathtt{C}) = \circ\ \mathtt{class\ P.C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \quad fields([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}) = (\overline{\mathtt{U}}\ \overline{\mathtt{g}})}{fields(\mathtt{C}\langle\overline{\mathtt{T}}\rangle) = (\overline{\mathtt{U}}\ \overline{\mathtt{g}},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}})}$$

Method Definition Lookup:

$$\frac{CT(\mathtt{C}) = \circ\ \mathtt{class\ P.C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \quad \mathtt{m}\text{ is defined in }\overline{\mathtt{M}}}{mdef(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}}$$

$$\frac{CT(\mathtt{C}) = \circ\ \mathtt{class\ P.C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \quad \mathtt{m}\text{ is not defined in }\overline{\mathtt{M}}}{mdef(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = mdef(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})}$$

Method Type Lookup:

$$\frac{mdef(\mathtt{m},\ \mathtt{N}) = \mathtt{D}\langle\overline{\mathtt{T}}\rangle \quad CT(\mathtt{D}) = \circ\ \mathtt{class\ P.D}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \quad \mathtt{U\ m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\,\{\mathtt{return\ e};\ \} \in \overline{\mathtt{M}}}{mtype(\mathtt{m},\ \mathtt{N}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}} \to [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}}$$

Method body look-up:

$$\frac{mdef(\mathtt{m},\ \mathtt{N}) = \mathtt{D}\langle\overline{\mathtt{T}}\rangle \quad CT(\mathtt{D}) = \circ\ \mathtt{class\ P.D}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \quad \mathtt{U\ m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\,\{\mathtt{return\ e};\ \} \in \overline{\mathtt{M}}}{mbody(\mathtt{m},\ \mathtt{N}) = (\overline{\mathtt{x}},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e})}$$

Valid method overriding

$$\frac{mtype(\mathtt{m},\ \mathtt{N}') = \overline{\mathtt{U}} \to \mathtt{U}\ \Rightarrow\ \mathtt{T}, \overline{\mathtt{T}} = (\mathtt{U}, \overline{\mathtt{U}}) \quad anon(\mathtt{m}, \mathtt{N}') \Rightarrow anon(\mathtt{m}, \mathtt{N})}{override(\mathtt{m}, \mathtt{N}, \mathtt{N}', \overline{\mathtt{T}} \to \mathtt{T})}$$

Figure 8: Auxiliary Functions

the partial order on $\mathtt{S}, \mathtt{T}$ still holds even if type variables in $\mathtt{S}, \mathtt{T}$ are replaced by confined types. We use $ConfVar(\mathtt{N})$ to denote the set of confined type variables in $\mathtt{N}$.

A type variable is visible in any class, while a non-variable type $\mathtt{N}$ is visible in the class $\mathtt{C}$ iff $conf(\mathtt{N}, \mathtt{P})$ implies that $\mathtt{C}$ is defined in $\mathtt{P}$. A type $\mathtt{N}$ is visible in $\mathtt{C}\langle\overline{\mathtt{N}}\rangle$ iff $conf(\mathtt{N}, \mathtt{P})$ implies either $\mathtt{C}$ is defined in $\mathtt{P}$ or $conf(\mathtt{C}\langle\overline{\mathtt{N}}\rangle, \mathtt{P})$.

Figure 8 contains the helper functions used in the typing rules and they are similar to those in Generic FJ.

Anonymous methods of a generic class $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$ stay anonymous even if the type parameters $\overline{\mathtt{X}}$ in class $\mathtt{C}$ are replaced by some types. In the rest of the paper, $anon(\mathtt{m}, \mathtt{C}\langle\overline{\mathtt{T}}\rangle)$ is equivalent to $anon(\mathtt{m}, \mathtt{C})$.

Figure 9 contains typing rules for expressions, methods, and classes, and also visibility rules for expressions. The expression typing rules are similar to those in Generic FJ with

Expression typing:

$$\Delta; \Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(GT-Var)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e_0} : \mathtt{T_0} \quad fields(bound_\Delta(\mathtt{T_0})) = (\overline{\mathtt{T}}\ \overline{\mathtt{f}})}{\Delta; \Gamma \vdash \mathtt{e_0.f_i} : \mathtt{T_i}} \qquad \text{(GT-Field)}$$

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash \mathtt{e_0} : \mathtt{T_0} \quad \Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \\ mdef(\mathtt{m},\ bound_\Delta(\mathtt{T_0})) = \mathtt{N_0} \\ mtype(\mathtt{m},\ \mathtt{N_0}) = \overline{\mathtt{U}} \to \mathtt{U} \quad \Delta \vdash \overline{\mathtt{V}} \preceq \overline{\mathtt{U}} \\ \Delta \vdash \mathtt{T_0} \preceq \mathtt{N_0}\ \vee\ anon(\mathtt{m}, \mathtt{N_0})\end{array}}{\Delta; \Gamma \vdash \mathtt{e_0.m}(\overline{\mathtt{e}}) : \mathtt{U}} \qquad \text{(GT-Invk)}$$

$$\frac{\Delta \vdash \mathtt{N} \quad fields(\mathtt{N}) = (\overline{\mathtt{T}}\ \overline{\mathtt{f}}) \quad \Delta; \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \quad \Delta \vdash \overline{\mathtt{S}} \preceq \overline{\mathtt{T}}}{\Delta; \Gamma \vdash \mathtt{new\ N}(\overline{\mathtt{e}}) : \mathtt{N}} \qquad \text{(GT-New)}$$

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T} \quad \Delta \vdash \mathtt{N} \\ conf(\mathtt{T}, \mathtt{P}) \Rightarrow conf(\mathtt{N}, \mathtt{P}) \\ ConfVar(\mathtt{T}) = ConfVar(\mathtt{N})\end{array}}{\Delta; \Gamma \vdash (\mathtt{N})\ \mathtt{e} : \mathtt{N}} \qquad \text{(GT-Cast)}$$

Method typing:

$$\frac{\begin{array}{l}\Delta = \overline{\mathtt{X}} <: \overline{\mathtt{N}} \quad \Gamma = \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{X}}\rangle \\ \Delta \vdash \overline{\mathtt{T}}, \mathtt{T} \quad \Delta; \Gamma \vdash \mathtt{e} : \mathtt{S} \quad \Delta \vdash \mathtt{S} \preceq \mathtt{T} \\ \Delta; \Gamma \vdash visible(\mathtt{e}, \mathtt{C}) \\ override(\mathtt{m}, \mathtt{C}\langle\overline{\mathtt{X}}\rangle, \mathtt{N}, \overline{\mathtt{T}} \to \mathtt{T})\end{array}}{\mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\,\{\mathtt{return\ e};\ \}\ \mathtt{OK\ IN\ C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle \lhd \mathtt{N}} \qquad \text{(GT-Method)}$$

Class typing:

$$\frac{\begin{array}{c}\overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{N}}, \mathtt{N}, \overline{\mathtt{T}} \quad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle \lhd \mathtt{N} \\ fields(\mathtt{N}) = (\overline{\mathtt{U}}\ \overline{\mathtt{g}}) \\ \mathtt{K} = \mathtt{C}(\overline{\mathtt{U}}\ \overline{\mathtt{g}},\ \overline{\mathtt{T}}\ \overline{\mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ visible(\mathtt{N}, \mathtt{C}) \quad conf(\mathtt{N}) \Rightarrow conf(\mathtt{C})\end{array}}{\circ\ \mathtt{class\ P.C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}} \qquad \text{(GT-Class)}$$

Expression visibility:

$$\frac{\Delta; \Gamma \vdash \mathtt{x} : \mathtt{T} \quad visible(\mathtt{T}, \mathtt{C})}{\Delta; \Gamma \vdash visible(\mathtt{x}, \mathtt{C})} \qquad \text{(GV-Var)}$$

$$\frac{\Delta; \Gamma \vdash visible(\mathtt{e}, \mathtt{C}) \quad \Delta; \Gamma \vdash \mathtt{e.f_i} : \mathtt{T} \quad visible(\mathtt{T}, \mathtt{C})}{\Delta; \Gamma \vdash visible(\mathtt{e.f_i}, \mathtt{C})} \qquad \text{(GV-Field)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{T} \quad visible(\mathtt{T}, \mathtt{C}) \quad \Delta; \Gamma \vdash visible(\mathtt{e}, \mathtt{C}) \quad \Delta; \Gamma \vdash visible(\overline{\mathtt{e}}, \mathtt{C})}{\Delta; \Gamma \vdash visible(\mathtt{e.m}(\overline{\mathtt{e}}), \mathtt{C})} \qquad \text{(GV-Invk)}$$

$$\frac{visible(\mathtt{N}, \mathtt{C}) \quad \Delta; \Gamma \vdash visible(\overline{\mathtt{e}}, \mathtt{C})}{\Delta; \Gamma \vdash visible(\mathtt{new\ N}(\overline{\mathtt{e}}), \mathtt{C})} \qquad \text{(GV-New)}$$

$$\frac{visible(\mathtt{N}, \mathtt{C}) \quad \Delta; \Gamma \vdash visible(\mathtt{e}, \mathtt{C})}{\Delta; \Gamma \vdash visible((\mathtt{N})\ \mathtt{e}, \mathtt{C})} \qquad \text{(GV-Cast)}$$

Figure 9: Typing Rules

some additional constraints to prevent explicit or implicit reference widening.

In the method typing rule, we require that the return expression of a method in class `C` be visible in `C`. The visibility rules of expressions in a generic class are similar to those of non-generic class. As in the case of ConfinedFJ, the visibility constraint of method body is meant to simplify the typing rules and it requires whole program analysis. To enable modular checking, we can substitute visibility constraint of method body with the confinement rules $\mathcal{C}2$, $\mathcal{C}3$, and $\mathcal{C}4$.

Unlike in ConfinedFJ, Rule $\mathcal{C}4$ is needed because Rules $\mathcal{C}2$, $\mathcal{C}3$ do not apply to fields of variable types or methods with variable return types.

For example, consider a generic confinable class `my.A<X^c>` and its instantiation `my.A<C>`. `my.A<C>` is confined because the type argument `my.C` is confined. The class `my.B` extends `my.A<C>` and the inherited field `f` in the class `my.B` is assigned a confined object `new C()` by the method `m` in class `my.D`. The method `breach` in the class `outside.E` is able to access the field `f` of `new B()` and return the confined object `new C()`. Since the class `outside.E` is defined outside the package of `my.C`, we have a violation of confinement property.

```
+ class my.A<X^c> {
    X f;
    A (X f) {this.f = f;}
}
- class my.C {}
+ class my.B extends A<C> {}
+ class my.D {
    B m() { return new B(new C()); }
}
+ class outside.E {
    Object breach(){return ((new D()).m()).f;}
}
```

The method `breach` is not typable because the Rule (T-Method) requires the type of `((new D()).m()).f` to be visible in `outside.E`, which is not true.

Rules $\mathcal{C}2$, $\mathcal{C}3$ alone are not able to prevent this problem because the type of the field `f` is variable type `X`. We may not want to set the access modifier of `f` as private or default, because the class `my.A<X^c>` would be less reusable: for example, `f` becomes inaccessible if `my.A<X^c>` is instantiated outside the package `my`. However, according to Rule $\mathcal{C}4$, the class `my.B` should be confined because $\mathcal{C}4$ requires the subclasses of a confined class to be confined. If `my.B` were confined, then `outside.E` could not call the method `m` of the class `my.D` since `m` would not be accessible to `outside.E` according to Rule $\mathcal{C}3$.

4.3 Properties

In this section, we prove some results similar to those for ConfinedFJ. In ConfinedFGJ, a program without free type variables should have the same confinement properties as does a program in ConfinedFJ.

Our typing rules for generic classes guarantee that if the type of an expression `e` is confined in a package `P` and $\mathtt{e} \rightarrow^* \mathtt{e}'$, then the type of $\mathtt{e}'$ is confined in `P` as well. This property is shown in the subject reduction lemma and theorem 10.

Lemma 8. **Subject Reduction:** *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{N}$ *and* $\mathtt{e} \rightarrow \mathtt{e}'$ *then* $\exists \mathtt{N}'$ *such that* $\emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{N}'$, *where* $\emptyset \vdash \mathtt{N}' \preceq \mathtt{N}$.

Proof. We prove by induction over the structure of `e`. The proof is similar to that of Lemma 1 except for the case when `e` is of the form $\mathtt{new\ N_0(\overline{u}).m(\overline{v})}$. □

Lemma 9. **Progress:** *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{N}$ *and* `e` *contains subexpression* $\mathtt{e_0}$ *where*

$$\begin{array}{rcl} \mathtt{e_0} & = & \mathtt{new\ N_0(\overline{v}).f_i} \\ & | & \mathtt{(N)\ new\ N'(\overline{v})},\ \emptyset \vdash \mathtt{N}' <: \mathtt{N}, \\ & | & \mathtt{new\ N_0(\overline{u}).m(\overline{v})}, \end{array}$$

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

Theorem 10. *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{N}$ *and* $\mathtt{e} \rightarrow^* \mathtt{e}'$, *then* $\mathtt{e}'$ *is not stuck and* $\exists \mathtt{N}'$ *such that* $\emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{N}'$, *where* $\emptyset \vdash \mathtt{N}' \preceq \mathtt{N}$.

Proof. Immediate from Lemma 8 and 9. □

Theorem 11 is similar to Theorem 7. The difference is that if a method defined in the class $\mathtt{C}\langle\overline{\mathtt{N}}\rangle$ can access an object confined in the package `P`, then either `C` is defined in `P` or there exists $\mathtt{N} \in \overline{\mathtt{N}}$ such that `N` is confined in `P`. The latter case may be true when a generic type $\mathtt{C}\langle\overline{\mathtt{X}}\rangle$ is instantiated with confined type arguments.

Theorem 11. *If* $\mathtt{e} = \mathtt{new\ N_0(\overline{u}).m(\overline{v})}$, $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{N}$, $mdef(\mathtt{m},\ \mathtt{N_0}) = \mathtt{N}'_0$, *and* $\mathtt{e} \rightarrow \mathtt{d}$, *then for every subexpression* $\mathtt{d}'$ *of* `d` *such that* $\mathtt{d}' \rightarrow^* \mathtt{new\ N'(\overline{u}')}$ *we have:*

1. *either* $visible(\mathtt{N}', \mathtt{N}'_0)$, *or*
2. $\mathtt{d}' = \mathtt{new\ N_0(\overline{u})}$, `m` *is anonymous, and* $\mathtt{d}'$ *appears in a subexpression of the form* $\mathtt{d}'.\mathtt{f}$ *or* $\mathtt{d}'.\mathtt{m}'(\overline{\mathtt{e}})$.

Proof. Similar to the proof of Theorem 7. □

5. 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 [13, 12, 9, 1, 16, 10, 15, 8]. In [16], flexible alias protection is presented as a means to control potential aliasing amongst components of an aggregate object. Clarke, Potter, and Noble [8] have formalized representation containment by means of ownership types. Boyland, Noble and Retert [5] introduced capabilities as a uniform system to describe restrictions imposed on references. Recent work by Boyapati *et al.* [3, 4] has applied ownership types to scoped memory in the Real-time Specification for Java. The paper by Banerjee and Naumann [2] demonstrated the use of object confinement to achieve representation independence. Their stronger notation of confinement is instance-based and it can be used to prove equivalence of class implementations such that if an implementation is confined, then it may be replaced by semantically equivalent ones without affecting the behavior of the whole program. Their work has significance in proving the equivalence of programs and the correctness of static analysis such as secure information flow.

In related work Clarke *et al.* [6] have adapted the notion of confined types to ensuring the integrity of components in the Enterprise JavaBeans framework. One of the interesting aspects of their work is that the unit of confinement is different. Rather than confining types within a package, they confine them within a Bean using the following rules (CB1-6): CB1 declares which types are confined ($\mathcal{C}2$ in our case), CB2 prevents confined types from appearing at the Bean boundary or in static variables (roughly equivalent to $\mathcal{C}3$), CB3 prevents widening of confined types (identical to $\mathcal{C}2$), CB4 prevents unconfined types to be cast to confined types (redundant as it follows from CB3 and CB6 that no cast from an unconfined type to a confined will ever succeed), CB5 prevents confined code from accessing unconfined classes which have confined types in their signature, and finally CB6 states that confined classes may only extend one another or `Object` (a stronger version of $\mathcal{C}4$). Rule CB6 precludes confined classes from inheriting code from non-confined classes and thus sidesteps the issue of anonymous methods. But the main difference between the systems lies in rules CB2 and CB5 which conspire to prevent the use of static variables to communicate across beans.

6. CONCLUSION

This paper has formalized the notion of *confined type* [17] 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. In future work we plan to study the relationship between confined types and more general purpose ownership type systems.

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 Phil Wadler for the initial impetus to look at generics. Finally we thank the anonymous referees and attendees of IWACO for their comments.

7. 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] Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control [extended abstract]. *ACM SIGPLAN Notices*, 37(1):166–177, January 2002.

[3] 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.

[4] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. June 2003.

[5] 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.

[6] 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.

[7] David Clarke. *Object Ownership and Containment.* PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.

[8] 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.

[9] D. Detlefs, K. Rustan M. Leino, and G. Nelson. Wrestling with rep exposure. Technical report, Digital Equipment Corporation Systems Research Center, 1996.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] S.J.H. Kent and I. Maung. Encapsulation and Aggregation. In *Proceedings of TOOLS PACIFIC 95 (TOOLS 18)*. Prentice Hall, 1995.

[16] 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.

[17] J. Vitek and B. Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.