JFP **15** (6): 1–46, 2005. © 2005 Cambridge University Press
doi:10.1017/S0956796805005691 Printed in the United Kingdom

Type-based confinement

TIAN ZHAO

Department of Electrical Engineering and Computer Science, University of Wisconsin-Milwaukee, PO Box 784, Milwaukee, WI 53201-0784, USA
(*e-mail:* `dta@cs.uwm.edu`)

JENS PALSBERG

Computer Science Department, UCLA, 4531K Boelter Hall, Los Angeles, CA 90095-1596, USA
(*e-mail:* `palsberg@ucla.edu`)

JAN VITEK

Department of Computer Sciences, Purdue University, 250 N. University Street, West Lafayette, IN 47907-2066, USA
(*e-mail:* `jv@cs.purdue.edu`)

Abstract

Confinement properties impose a structure on object graphs which can be used to enforce encapsulation properties. From a practical point of view, encapsulation is essential for building secure object-oriented systems as security requires that the interface between trusted and untrusted components of a system be clearly delineated and restricted to the smallest possible set of operations and data structures. This paper investigates the notion of package-level confinement and proposes a type system that enforces this notion for a call-by-value object calculus as well as a generic extension thereof. We give a proof of soundness of this type system, and establish links between this work and related research in language-based security.

1 Introduction

While object-oriented languages provide syntactic support for encapsulating fields of object structures via access and visibility annotations, this form of name-based encapsulation only protects variables and the values they refer to. The runtime behavior of programs clearly shows that name-based protection mechanisms are not sufficient to protect an object's representation. Reference semantics allows creating dynamic aliases to an object referred to from a protected variable, which may lead to unintended side-effects. This has implications for software engineering and information security. The software engineering drawbacks have been discussed by Leavens (1991): without strong encapsulation it is difficult to reason about programs modularly. Information security requires that boundaries between trusted and untrusted components be established. Strong encapsulation is one way to define such boundaries and ensure that some parts of a system not be exposed to untrusted components.

Research on strong encapsulation started in the early 1990's. The work on Islands (Hogg, 1991) stands out as one of the first attempts to propose a language abstraction for enforcing strong encapsulation. A good summary of the early research on aliasing appeared in Hogg *et al.* (1992). The original flexible alias protection paper (Noble *et al.*, 1998) proposed an approach that relied on type qualifiers and generic types to control aliasing. Many researchers extended this work, referred to as *ownership types*: Clarke *et al.* (1998) and Clarke (2001) formalized the type system, Boyapati *et al.* (2002; 2003c) extended the expressive power and defined domain-specific variants. A complete list is given in the related work section.

We view strong encapsulation and aliasing control as a prerequisite for writing secure systems out of components that are not necessarily trusted. In this paper we investigate a programming language extension and programming discipline for enforcing strong encapsulation, or *confinement*, in languages such as Java and C#. What sets our work apart from previous results is that rather than aiming for the most expressive confinement mechanism, we look for the *least disruptive* one: an encapsulation mechanism that requires as few changes as possible to the tool chain (compilers, verifiers, virtual machines, etc.) and the smallest possible changes to the programming model. Ideally, it should simply codify best practice principles already familiar to programmers. This paper shows that it is possible to obtain a useful degree of encapsulation with very few changes to the semantics of an object-oriented language and retain a natural programming model.

Confined types are a mechanism for strong encapsulation for the Java programming language (Vitek & Bokowski, 2001). They are non-intrusive as they require few changes to the source language and programming model and only two new program annotations. Classes that must be encapsulated are marked as `confined` and methods that can be safely inherited by confined classes are marked `anonymous`. Confined types are a proper restriction of the language as programs written with confinement annotations are valid Java programs if the annotations are erased. Vitek and Bokowski (2001) showed that confined types can be checked independently of other properties by inspection of the bytecode. They require no changes to compiler, verifier or virtual machine. The encapsulation guarantee afforded by confined types is the following: an instance of an annotated class can be manipulated only by objects defined in the same Java package. Java packages are software modules bundling a number of classes. They have very little role in the language apart from providing a scoping mechanism for class declarations. Confinement can be viewed as strengthening visibility rules to ensure that instances of a package-scoped class do not escape their defining package.

Confined types are type qualifiers in the sense of Foster *et al.* (1999), though their work addresses a language without subtyping and inheritance. For confined types it is necessary to restrict widening of types to prevent a confined type from being cast to a plain reference. The `confine` keyword introduced in Foster *et al.* (1999) is unrelated to our notion of confinement.

The encapsulation property enforced by confined types is static and coarse grained. There are a finite number of scopes, bounded by the number of distinct packages in the program and objects within a package cannot be differentiated. This can be contrasted with ownership type systems *à la* Clark (2001), where each object can define its own scope and different instances of the same class can be protected from on another. Extending confined types with generics achieves some of the flexibility of ownership types, but the number of scopes remains bounded.

A significant drawback of ownership type systems is that they require an overhaul of the language and force programmers to be aware of object ownership throughout their design. Without extensive empirical evaluation, it remains to be seen if the benefits of such language extensions outweigh their costs. Confined types are simpler in the sense that annotations are only needed for packages that require protection. The rest of the system can be programmed in plain Java without even knowing about confinement. Confinement checks are applied only to code of packages that declared confined types. In related work we developed a whole-program confinement inference algorithm (Grothoff *et al.*, 2001). Analysis of a large body of Java code reveals that many classes can be confined without any changes to the source code. This supports our contention that confinement is a natural property of well-designed Java programs. Recent work by Potanin *et al.* (2004b) provides an elegant account of generic ownership and hints at ways to incorporate a more expressive ownership system at little cost in simplicity.

The main contributions of the paper are the following:

- We present a straightforward formalization of the rules posited in Vitek and Bokowski (2001) as a type system for a simple call-by-value object calculus. Our calculus, ConfinedFJ, is based on the Featherweight Java (FJ) calculus (Igarashi *et al.*, 2001). FJ is a class-based object calculus designed to model the Java type system. We believe that the simplicity of the type rules and the backwards compatibility with Java are encouraging signs for the prospect of acceptance by practitioners.
- We prove the soundness of the type system, as well as a Confinement Theorem stating that well-typed programs preserve heap confinement. This is the first proof of confinement for a class-based calculus with a small-step operational semantics. Previous results by Foster (2002) did not treat subtyping. Clarke's ownership results are for a variant of Cardelli and Abadi's imperative object calculus (Abadi & Cardelli, 1996) with a big-step semantics. Banerjee and Naumann adopted a denotational semantics in Banerjee and Naumann (2002a). Finally, proving the soundness of the ownership type system of Boyapati (2004) remains an open problem.
- We extend the original definition of confined types to support generics in a language modeled on the Featherweight Generic Java. Significantly, supporting genericity requires adding two rules to the constraints of Vitek and Bokowski (2001). We show by way of examples that generics significantly increase the expressive power of confined types. Our proof of the Confinement Theorem is the first such proof for a generic type system that we are aware of.

ConfinedFJ abstracts Java by omitting features which do not affect the Confinement Theorem, these include exceptions, interfaces, downcasts, and state. We argue that these features can be easily incorporated in the formalization. Checked exceptions can be modeled by enriching return values. As they appear in the type signature of the method, the confinement rules for exceptions are exactly the same as for other objects. Unchecked exceptions cannot be confined, as there is no simple way to determine which unchecked exceptions may be thrown by a method. Interfaces are dealt with in the same way as with class definition. Downcasts, *i.e.* casts from a supertype to a subtype, can introduce runtime failures which complicate the formal treatment and proofs. As confinement is a downwards-closed property of the type system, downcasts cannot violate encapsulation. Finally, it may appear paradoxical that a stateless calculus is used to address issues linked to aliasing. However, confinement, unlike other ownership type systems, treats all values of the same type equally. The confinement rules partition the set of types and prevent types belonging to different partition from being confused with one another.

ConfinedFJ departs from FJ by adopting a call-by-value semantics and by keeping track of evaluation context in the dynamic semantics. These changes permit us to precisely determine which objects are accessed during the evaluation of a method. Another approach is to rely on an extended syntax to keep track of evaluation contexts. This is in line with the syntactic type abstraction of Grossman *et al.* (2000) or the box-π of Sewell and Vitek (2003). In a previous version of the calculus we tried to follow Grossman *et al.*, but with a lazy semantics, and found that the dynamic semantics was cumbersome and the proof of the Confinement Theorem was significantly more challenging.

Paper organization

Section 2 presents a motivating example and illustrates the main idea behind confined types. Section 3 introduces confinement rules. Section 4 gives a more detailed presentation of confined types. Section 5 introduces Confined Featherweight Java and gives it an operational semantics and a static type system. Section 6 presents our Confined Generic Featherweight Java. Section 7 discusses other work related to aliasing control.

The notion of confined types was first introduced by Bokowski and Vitek (2001). The paper introduced a confinement checker for the full Java language and gave an informal correctness argument. Grothoff *et al.* (2001) implemented a static analysis tool for inferring confinement annotations.

This work extends our previously published paper (Zhao *et al.*, 2003). The main difference with the earlier papers is that the set of confinement and anonymity rules has been simplified. The OOPSLA'03 version of this work did not include a full proof of the Confinement Theorem. The present paper also includes an extended discussion and examples.

2 Motivating example: information security

The original motivation for confined types arose out of a security breach in the SUN Java Virtual Machine. This section presents a simplified version of the program discussed in Vitek and Bokowski (2001). The problem resulted from a combination of two features of Java, namely, dynamic aliasing and side-effects. Figure 1 contains the definition of class `Class` which, in Java, holds meta-information about a class

```
class Class {
   private Identity[] signers;
   public Identity[] getSigners( ) {
      return signers;
   }
}
```

Fig. 1. Signatures without confined types. The `signers` field holds capabilities that are managed by the security subsystem. By returning the object referenced by `signers`, the class expose the array to updates by outside code.

loaded by the virtual machine. Each class has an array of `Identity` objects that hold the signatures of principals vouching for the class. This array is use to determine the access rights of the class. The interface `Class` includes a method `getSigners()` which returns the array of signatures. The array is declared `private` to ensure that the field is visible only in the body of `Class`. Since, `getSigners()` is public untrusted code can obtain a dynamic alias to the object referred to by `signers` and, since arrays are mutable the code can simply change its permissions.

Interestingly, `signers` was correctly identified as requiring protection, but the implementation of the class failed to enforce the designer's intention. In this particular example, what seems to be missing from the language is a way to express that it is the contents of the field and not only its name that should be protected.

This kind of security flaw cannot be easily addressed by the mechanisms provided by the Java language. There are at least three ways to try to address the problem. Firstly, one may try to restrict the scope of the `Identity` class to its defining package using access modifier. But declaring the class to be package-scoped does not guarantee that the array will not escape as it can be widened to a public supertype. A second potential solution is to use stack inspection. This mechanism checks dynamically whether an operation is permitted by reflectively inspecting the call stack of the current thread. Execution proceeds past the check if the intersection of the access rights of all methods on the stack allows it. There are two problems with that solution: firstly, there is no convenient place to add access checks, security is violated when the array is updated. Secondly, even if it was possible, the performance cost of checking all array stores would be prohibitive. Finally, a pragmatic solution is to copy the array, thus avoiding the sharing that is the root of the problem. Unfortunately this is ad hoc and error-prone as the programmer must manually identify all cases where a dynamic alias may reveal a protected object.

2.1 A solution with confined types

Confined types provide a way for programmers to declare that some objects are restricted to a scope. In the above example, the `Identity` class can be declared as confined and an automated confinement checking procedure will validate that the

```
confined class SecureIdentity {
   ... // original implementation
}

public class Identity {
   SecureIdentity target;
   Identity(SecureIdentity t) { target = t; }
   ...  // public operations on identities;

}

public class Class {
   private SecureIdentity[] signers;
   public Identity[] getSigners( ) {
      Identity[] pub = new Identity[signers.length];
      for (int i = 0; i < signers.length; i++)
         pub[i] = new Identity(signers[i]);
      return pub;
   }
}
```

Fig. 2. Signatures with confined types. The `Identity` class has been renamed `SecureIdentity` and declared confined. A new `Identity` class has been added to allow untrusted code to get information about the signers of a class without allowing modifications to the internal state.

program does not expose instances of that class. Refactoring the original program to use confined types is done in several steps. First `Identity` class is made confined. This expresses the programmer's intent that references to `Identity` instances should not escape from the implementation of `Class`. The code that manipulates objects of this class must belong to the current package. Since identities are exported through the `getSigners()` method, the checker will flag the method with a confinement error. The second step of refactoring, which is needed in order to preserve the interface of class `Class`, is to provide a public facade class, `Identity`, that can be exported to clients and rename the original `Identity` class to `SecureIdentity`. The `getSigner()` method is rewritten to create an array of `Identity` instances. The resulting code typechecks and is given in Figure 2. The final result of refactoring the program is not really surprising it follows the guidelines set for Guard Objects (Gong, 1998). The difference is that it comes with a guarantee that the guarded object (the instance of `SecureIdentity`) is not revealed by accident.

2.2 Related approaches

It is interesting to contrast confined types with other work in language-based security. Confined types are related to capability systems if one views object references as capabilities and the type system as a reference monitor. There is a substantial body of work on using facade or wrapper objects to interpose between trusted and untrusted components (Levy, 1984; Gong, 1998; Hagimont *et al.*, 1996; Wallach *et al.*, 1997; Vitek & Bryce, 2001). Discretionary access control checking can be added to these systems by stack introspection (Gong, 1999). Confined types are complementary to these approaches as they give static guarantee of encapsulation. A type-based approach to enforcing encapsulation of heap location was presented in Leroy and Rouaix (1998) in the context of a functional language. The type system considered there did not have subtyping nor runtime coercions.

3 Confined types

In modern object-oriented programming languages, confinement can be achieved by disciplined use of built-in static access control mechanisms combined with some simple coding idioms. Confinement enforces the following informal soundness property:

An object of confined type is encapsulated within its defining scope.

We assume the granularity of confinement to be a Java package to leverage existing access control mechanisms and minimize the changes to the programming model. In fact, as we show in section 3.4, many existing Java programs require no changes. Confined types establish a distinction between public types and, so called, *confined types*. The intended programming model is to have systems in which classes defined in the same packages form two distinct software layers: a package "interface" made up of public classes and a package "core" consisting of confined classes. We use the term interface loosely to refer to the classes that are exposed directly to clients of the package. Confinement ensures that core classes will not be directly accessed outside of the package by extending the existing Java visibility rules with restrictions on subtyping and inheritance.

Consider the following simple example. A class `Bucket` is used to implement a hash table class, `Table`. 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 class's defining the package. (Or `Bucket` can be a package-scoped inner class but there will be similar problems as described below.)

```
package p;

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

confined class Bucket {
   Bucket next;
   Object key, val;
}
```

$\mathscr{C}1$ *A confined type must not appear in the type of a public (or protected) field or the return type of a public (or protected) method.*

$\mathscr{C}2$ *A confined type must not be public.*

$\mathscr{C}3$ *Methods invoked on an expression of confined type must either be defined in a confined class or be anonymous methods.*

$\mathscr{C}4$ *Subtypes of a confined type must be confined.*

$\mathscr{C}5$ *Confined types can be widened only to other confined types.*

$\mathscr{C}6$ *Overriding must preserve anonymity of methods.*

Fig. 3. Confinement constraints.

$\mathscr{A}1$ *The* `this` *reference is used only to select fields and as the receiver in the invocation of other anonymous methods.* **T1**

Fig. 4. Anonymity constraint.

But what if one of `Table`'s public methods, such as `get()`, were to return a bucket or store a reference in one of its public fields? One can view this as an escape analysis problem: can references to the instances of a package-scoped class escape the scope of their enclosing package? If not, then the objects of such a class are encapsulated. Enforcing confinement implies tracking the spread of confined objects within a package and preventing them from crossing package boundaries. Since confinement is couched in terms of object types, widening a value from a confined type to a non-confined type presents a risk and is thus treated as confinement violation.

Confinement can be enforced (or inferred) using two sets of constraints. The first set of constraints, *confinement rules*, applies to the classes defined in the same package as the confined class. 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 kind of constraints, *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.

3.1 Confinement rules

The confinement rules must in Figure 3 hold for all classes of a package containing confined types.

Rule $\mathscr{C}1$ prevents exposure of confined types in the public interface of the package as client code could break confinement by accessing values of confined types through a type's public interface. Rule $\mathscr{C}2$ is needed to ensure that client code cannot instantiate a confined class. It also prevents client code from declaring field or variables of confined types. The latter restriction is needed so that code in a confining package will not mistakenly assign objects of confined types to the fields or variables outside that package. Rule $\mathscr{C}3$ ensures that methods invoked on an object enforce confinement. In the case of methods defined in the confining package, this ensues from the other confinement rules. Inherited methods defined in another package do not have access to any confined fields, since those are package-scoped (Rule $\mathscr{C}1$). However, an inherited method of confined class may leak the `this` reference, which is implicitly widened to the method's declaring class. To prevent this, Rule $\mathscr{C}3$ requires these methods to be anonymous (as explained below). Rule $\mathscr{C}4$ prevents the declaration of a public subclass of a confined type. This prevents *spoofing* leaks where a public subtype defined outside of the confined package is used to access private fields (Clarke *et al.*, 2003), and it also necessary when considering generic classes in Section 6. Rule $\mathscr{C}5$ prevents code within confining packages from assigning values of confined types to fields or variables of public types. Finally, Rule $\mathscr{C}6$ allows us to statically verify the anonymity of the methods that are invoked on expressions of confined types.

3.2 Anonymity rule

The anonymity rule applies to inherited methods which may reside in classes outside of the enclosing package. This rule prevents a method from leaking the `this` reference. A method is *anonymous* if it has the following property.

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, as it cannot break confinement 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. Note that we check this constraint assuming the static type of `this` and Rule $\mathscr{C}6$ ensures that the actual method invoked on `this` will also be anonymous.

Thus, Rule $\mathscr{C}6$ ensures that the anonymity of a method is independent of the result of method lookup. However, as explained in Grothoff *et al.* (2001), Rule $\mathscr{C}6$ is not necessary if we infer the anonymity of a method relative to a specific type (in which case we need to have Rule $\mathscr{C}4$). We choose to keep Rule $\mathscr{C}6$ because it is also needed for confined generic class in Section 6.

Rule $\mathscr{C}6$ could be weakened to apply only to methods inherited by confined classes. For instance, if an anonymous method `m` of class `A` is overridden in both class `B` and `C`, and `B` is extended by a confined class while `C` is not, then the method `m` in `B` must be anonymous while `m` of `C` needs not be. The reason is that the method `m` of `C` will never be invoked on confined objects and thus there is no need for it to be anonymous.

3.3 Checking confinement

Validation of these rules is modular. Classes can be verified independently. Moreover, the confinement invariant is backwards compatible in the sense that packages that do not use confinement or contain classes extended by confined classes can be checked by the normal Java type checker and do not require further processing. The confinement rules outlined above place no constraints on clients of a confined package (rule $\mathscr{C}1$ is crucial in this respect). The only constraints that must be enforced are that all classes within the package of a confined class must be checked and Rule $\mathscr{A}1$ must be applied to methods inherited by confined classes if these methods must be anonymous by Rule $\mathscr{C}3$ or $\mathscr{C}6$. As long as all classes in a package are known, confinement annotations can trivially be checked as part of the source-level type checking or by bytecode verification. Confined-type inference (as opposed to type checking) can be performed on a per-package basis, with the exception of anonymous methods which require analyzing parent classes (Grothoff *et al.*, 2001).

3.4 Empirical evaluation

Grothoff implemented a tool to evaluate the practicality of confined types on real programs (Grothoff *et al.*, 2001). The tool infers confinement by a whole-program static analysis. A study of over 100,000 Java classes of varying size, purpose and origin, gives empirical evidence to support the claim that confinement constraints are not too restrictive. The analysis focus on package-scoped classes, as public ones cannot be confined. Approximately 7,000 confined classes were found in the benchmark suite. Manual inspection of the source code suggests that many other classes could be confined with minimal effort. In another study, Potanin *et al.* (2004a) used dynamic analysis to get an upper bound on the number of objects that are actually confined during program execution. They report that more than 30% of all objects within their benchmark suite are effectively confined. Anonymity is also quite frequent, holding in 40% of the methods in the benchmark suite. The results also show that the single largest source of confinement violation, approximately 2000 classes, comes from collection classes. This is because all arguments to a collection type are widened to `Object`, which violates confinement. We surmise that most of these violations could be avoided with generic classes and proper extensions of confinement to handle genericity.

From a practical perspective, confined types can be criticized as 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. Thus it is, for instance, not possible to have the same confined vector class be used in several packages. This can become unwieldy when dealing with programs that require the same logic to be available in, and confined to, different packages. Any solution to this problem should allow the definition of 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. The extension of confinement to generic classes described in section 6 addresses this issue by allowing generic classes to have confined instantiations.

4 Confined Featherweight Java

Confined Featherweight Java, which we refer to as ConfinedFJ, is a minimal core calculus for modeling confinement for a Java-like object-oriented language.

ConfinedFJ extends Featherweight Java (FJ) which was designed by Igarashi, Pierce and Wadler (2001) to model the Java type system. It is a core calculus as it limits itself to a subset of the Java language with the following five basic expressions: object construction, method invocation, field access, casts and local variable access. This spartan setting has proved appealing to researchers. ConfinedFJ stay true to the spirit of FJ. The surface differences lie in the presence of class and method level visibility annotations. In ConfinedFJ, classes can be declared to be either public or confined, and methods can optionally be declared as anonymous. One further difference is that ConfinedFJ class names are pairs of identifiers bundling a package name and a class name just as in Java.

4.1 Syntax

Let metavariable $\mathtt{L}$ range over class declarations, $\mathtt{C}, \mathtt{D}, \mathtt{E}$ range over a denumerable set of class identifiers, $\mathtt{K}, \mathtt{M}$ range over constructor and method declarations respectively, and $\mathtt{f}$ and $\mathtt{x}$ range over field names and variables (including parameters and the pseudo-variable `this`) respectively. Let $\mathtt{e}, \mathtt{d}$ range over expressions and $\mathtt{u}, \mathtt{v}, \mathtt{w}$ range over values.

We adopt FJ notational idiosyncrasies and use an over-bar to represent a finite (possibly empty) sequence. We write $\overline{\mathtt{f}}$ to denote the sequence $\mathtt{f}_1, \ldots, \mathtt{f}_\mathtt{n}$ and similarly for $\overline{\mathtt{e}}$ and $\overline{\mathtt{v}}$. We write $\overline{\mathtt{C}\,\mathtt{f}}$ to denote $\mathtt{C}_1\,\mathtt{f}_1, \ldots \mathtt{C}_\mathtt{n}\,\mathtt{f}_\mathtt{n}$, $\overline{\mathtt{C}} <: \overline{\mathtt{D}}$ to denote $\mathtt{C}_1 <: \mathtt{D}_1, \ldots, \mathtt{C}_\mathtt{n} <: \mathtt{D}_\mathtt{n}$ and finally $\mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}}$ to denote $\mathtt{this}.\mathtt{f}_1 = \mathtt{f}_1, \ldots, \mathtt{this}.\mathtt{f}_\mathtt{n} = \mathtt{f}_\mathtt{n}$.

The syntax of ConfinedFJ is given in Figure 5. An expression $\mathtt{e}$ can be either one of a variable $\mathtt{x}$ (including `this`), a field access $\mathtt{e.f}$, a method invocation $\mathtt{e.m}(\overline{\mathtt{e}})$, a cast $(\mathtt{C})\ \mathtt{e}$, an object $\mathtt{new\ C}(\overline{\mathtt{e}})$. Since ConfinedFJ has a call-by-value semantics, it is expedient to add a special syntactic form for fully evaluated objects, denoted $\mathtt{new\ C}(\overline{\mathtt{v}})$.

Class identifiers are pairs $\mathtt{p.q}$ such that $\mathtt{p}$ and $\mathtt{q}$ range over denumerable disjoint sets of names. For ConfinedFJ class name $\mathtt{p.q}$, $\mathtt{p}$ is interpreted as a *package name* and $\mathtt{q}$ as a *class name*. In ConfinedFJ, class identifiers are fully qualified. For a class identifier $\mathtt{C}$, *packof*$(\mathtt{C})$ denotes the identifier's package prefix, so, for example, the value of *packof*$(\mathtt{p.O})$ is $\mathtt{p}$.

Class declarations are annotated with an optional visibility modifier `conf`; a public class is declared by $\mathtt{class\ C} \lhd \mathtt{D}\ \{\ldots\}$ and a confined class is $\mathtt{conf\ class\ C} \lhd \mathtt{D}\{\ldots\}$. Methods can be annotated with the optional `anon` modifier to denote anonymity.

4.2 Dynamic Semantics

The dynamic semantics of ConfinedFJ is given in Figure 7 in terms of a small-step operational semantics. The main departures from FJ are the choice of a call-by-value semantics and the addition of an explicit stack, both of which are required for the proof of the Confinement Theorem of Section 5. Computation rules are of the form $P \rightarrow P'$, where P is a possibly empty sequence of frames defined by the grammar:

$$P ::= nil \mid P\ .\ \mathtt{v\ m\ e}$$

$$
\begin{array}{lcl}
\mathtt{C} & ::= & \mathtt{p.q} \\
\mathtt{L} & ::= & [\mathtt{conf}]\ \mathtt{class}\ \mathtt{C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{C}\ \mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \} \\
\mathtt{K} & ::= & \mathtt{C}(\overline{\mathtt{C}\ \mathtt{f}})\ \{\ \mathtt{super}(\overline{\mathtt{f}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\ \} \\
\mathtt{M} & ::= & [\mathtt{anon}]\ \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}\ \mathtt{x}})\ \{\ \mathtt{return}\ \mathtt{e};\ \} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid (\mathtt{C})\ \mathtt{e} \mid \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) \\
\mathtt{v} & ::= & \mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}})
\end{array}
$$

Fig. 5. ConfinedFJ: Syntax.

A frame v m e denotes the invocation of some method m on a receiver object v where e is the body of the method being evaluated. As usual, $\rightarrow^*$ denotes transitive and reflexive closure of $\rightarrow$.

We define, in Figure 7, an *evaluation context* to be an expression $E[\circ]$ with a hole and $E[\mathtt{e}]$ means E with the hole replaced by e. The syntax of method and constructor contexts $E[\circ].\mathtt{m}(\overline{\mathtt{e}})$, $\mathtt{v.m}(\overline{\mathtt{v}}, E[\circ], \overline{\mathtt{e}})$, $\mathtt{new}\ \mathtt{C}(\overline{\mathtt{v}}, E[\circ],\overline{\mathtt{e}})$ enforce left-to-right evaluation order and call-by-value semantics. Evaluation context are deterministic. For any expression e, there is exactly one evaluation context. This formally stated in Lemma 1, which can be proved by induction on the structure of e.

Lemma 1 (*Context determinacy.*)
For all closed expression e, exactly one of the following holds:

1. e is a value;
2. e has the form $E[\mathtt{v.f}]$ for some E;
3. e has the form $E[(\mathtt{C})\ \mathtt{v}]$ for some E;
4. e has the form $E[\mathtt{v.m}(\overline{\mathtt{v}})]$ for some E.

We now detail the evaluation rules.

- Rules R-Field and R-Cast evaluate field access and type cast expressions. The rules differ from FJ only in that subexpressions are fully evaluated.
- Rule R-Invk evaluates a method invocation of the form $\mathtt{e} = \mathtt{v}'.\mathtt{m}'(\overline{\mathtt{v}'})$ in some context $P\ .\ \mathtt{v}\,\mathtt{m}\,E[\circ]$. A new frame is created with $\mathtt{v}'$ as receiver, $\mathtt{m}'$ as method, and the body of $\mathtt{m}'$ as the expression being evaluated. The resulting configuration has the form $P\ .\ \mathtt{v}\,\mathtt{m}\,E[\mathtt{e}]\ .\ \mathtt{v}'\,\mathtt{m}'\,\mathtt{e}'$. This rule differs from FJ due to the presence of frames.
- Rule R-Ret describes how the result of a method invocation is returned to its calling context. If the topmost frame is a value, and the configuration has the form $\mathtt{v}\,\mathtt{m}\,E[\mathtt{e}]\ .\ \mathtt{v}'\,\mathtt{m}'\,\mathtt{v}''$, then the top frame is popped off and expression e is replaced the result $\mathtt{v}''$. The replacement is unambiguous since, by Lemma 1, context $E[\circ]$ is unique. This rule has no correspondence in FJ.

Figure 6 gives some standard definitions. We assume a class table CT which stores the definitions of all classes of ConfinedFJ program such that $CT(\mathtt{C})$ is the definition of class C. Following Igarashi *et al.* (2001), we leave the class table as an implicit

Subtyping:

$$\texttt{C <: C} \qquad \frac{\texttt{C <: D} \quad \texttt{D <: E}}{\texttt{C <: E}} \qquad \frac{CT(\texttt{C}) = \texttt{[conf] class C} \lhd \texttt{D} \{ \ldots \}}{\texttt{C <: D}}$$

Field look-up:

$$\frac{}{\mathit{fields}(\texttt{l.Object}) = ()} \qquad \frac{\mathit{fields}(\texttt{D}) = (\overline{\texttt{D g}}) \quad CT(\texttt{C}) = \texttt{[conf] class C} \lhd \texttt{D} \{ \overline{\texttt{C f}};\ \texttt{K}\ \overline{\texttt{M}} \}}{\mathit{fields}(\texttt{C}) = (\overline{\texttt{D g}},\ \overline{\texttt{C f}})}$$

Method definition lookup:

$$\frac{CT(\texttt{C}) = \texttt{[conf] class C} \lhd \texttt{D} \{ \overline{\texttt{C f}};\ \texttt{K}\ \overline{\texttt{M}} \}}{\mathit{methods}(\texttt{C}) = \overline{\texttt{M}}}$$

$$\frac{\texttt{[anon] B m}(\overline{\texttt{B x}}) \{ \texttt{return e}; \} \in \mathit{methods}(\texttt{C})}{\mathit{mdef}(\texttt{m},\ \texttt{C}) = \texttt{C}}$$

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

Fig. 6. ConfinedFJ: Types and Lookup.

Evaluation:

$$\frac{\texttt{e} = \texttt{new C}(\overline{\texttt{v}}).\texttt{f}_\texttt{i} \quad \mathit{fields}(\texttt{C}) = (\overline{\texttt{D}}\,\overline{\texttt{f}})}{P\ .\ \texttt{v m}\ E[\texttt{e}] \ \rightarrow\ P\ .\ \texttt{v m}\ E[\texttt{v}_i]} \qquad \text{(R-Field)}$$

$$\frac{\texttt{e} = (\texttt{C}')\ \texttt{new C}(\overline{\texttt{v}}) \quad \texttt{C <: C}'}{P\ .\ \texttt{v m}\ E[\texttt{e}] \ \rightarrow\ P\ .\ \texttt{v m}\ E[\texttt{new C}(\overline{\texttt{v}})]} \qquad \text{(R-Cast)}$$

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

$$\frac{\texttt{e} = \texttt{v}'.\texttt{m}'(\overline{\texttt{v}})}{P\ .\ \texttt{v m}\ E[\texttt{e}]\ .\ \texttt{v}'\ \texttt{m}'\ \texttt{v}'' \ \rightarrow\ P\ .\ \texttt{v m}\ E[\texttt{v}'']} \qquad \text{(R-Ret)}$$

Evaluation contexts:

$$E[\circ] \quad ::= \quad \circ \ \mid\ (\texttt{C})\ E[\circ] \ \mid\ E[\circ].\texttt{f}_\texttt{i} \ \mid\ E[\circ].\texttt{m}(\overline{\texttt{e}}) \ \mid\ \texttt{v.m}(\overline{\texttt{v}}, E[\circ], \overline{\texttt{e}}) \ \mid\ \texttt{new C}(\overline{\texttt{v}}, E[\circ], \overline{\texttt{e}})$$

Fig. 7. ConfinedFJ: Dynamic semantics.

parameter to the semantics. The subtyping relation $\texttt{C} <: \texttt{D}$ denotes that class C is a subtype of class D. Every class is a subtype of `l.Object`. The function *fields*(C) return the list of all fields of the class C including inherited ones; *methods*(C) returns the list of all methods in the class C; *mdef*(m) returns the identifier of defining class for the method m.

4.3 Static semantics

Figure 8 defines relations used in the static semantics. The predicate *conf*(C) holds if the class table maps C to a class declared as confined. Functions *mtype*(m, C) and *mbody*(m, C) yield, respectively, the type signature and body of a method. Predicate *override*(m, C, D) holds if a m is a valid, anonymity preserving, redefinition of an inherited method or if this is the method's original definition. Class visibility, written *visible*(C, D), states that a class C is visible from D if, either, C is public, or if both classes are in the same package.

The *safe subtyping* relation, written $\texttt{C} \preceq \texttt{D}$, is a confinement preserving restriction of the subtyping relation $<:$. A class C is a safe subtype of D if C is a subtype of D, and either C is public or D is confined. This relation is used in the typing rules to prevent widening a confined type to a public type; confinement-preserving widening requires safe subtyping to hold. The type system further constrains subtyping by enforcing that all subclasses of a confined class must belong to the same package (see the T-Class rule and the definition of visibility). This relation is also transitive. To see that, suppose that $\texttt{C} \preceq \texttt{C}'$ and $\texttt{C}' \preceq \texttt{C}''$. Then, by definition, $\texttt{C} <: \texttt{C}', \texttt{C}' <: \texttt{C}''$, and if C is confined, then so is $\texttt{C}'$, and in which case $\texttt{C}''$ must be confined as well. Since subtyping relation is transitive, we have $\texttt{C} <: \texttt{C}''$. Thus, $\texttt{C} \preceq \texttt{C}''$.

Figure 9 defines constraints imposed on anonymous methods. A method m is anonymous in class C, written *anon*(m, C), if its declaration is annotated with the `anon` modifier. The following syntactic restrictions are imposed on the body of an anonymous method. An expression e is anonymous in class C, written *anon*(e, C), if the pseudo-variable `this` is used solely for field selection and anonymous method invocation. (C) e is anonymous if e is anonymous. new $\texttt{C}(\overline{\texttt{e}})$ and $\texttt{e.m}(\overline{\texttt{e}})$ are anonymous if $\texttt{e} \neq \texttt{this}$ and $\texttt{e}, \overline{\texttt{e}}$ are anonymous. With the exception of `this` all variables are anonymous. `this.f` is always anonymous, and $\texttt{this.m}(\overline{\texttt{e}})$ is anonymous in C if m is anonymous in C and $\overline{\texttt{e}}$ is anonymous. We write *anon*($\overline{\texttt{e}}$, C) to denote that all expressions in $\overline{\texttt{e}}$ are anonymous.

4.3.1 Expression typing rules

The typing rules for ConfinedFJ are given in Figure 10, where type judgments have the form $\Gamma \vdash \texttt{e} : \texttt{C}$, in which Γ is an environment that maps variables to their types. The main difference with FJ is that these rules disallow unsafe widening of types. This is captured by conditions of the form $\texttt{C} \preceq \texttt{D}$ which enforce safe subtyping.

- Rules T-Var and T-Field are standard.
- Rule T-New prevents instantiating an object if any of the object's fields with a public type is given a confined argument. That is, for fields with declared

Confined types, type visibility, and safe subtyping:

$$\frac{CT(\mathtt{C}) = \mathtt{conf\ class\ C} \lhd \mathtt{D}\ \{\ldots\}}{\mathit{conf}(\mathtt{C})}$$

$$\frac{\neg\mathit{conf}(\mathtt{C})}{\mathit{visible}(\mathtt{C},\mathtt{D})} \qquad \frac{\mathit{packof}(\mathtt{C}) = \mathit{packof}(\mathtt{D})}{\mathit{visible}(\mathtt{C},\mathtt{D})}$$

$$\frac{\mathtt{C} <: \mathtt{D} \quad \mathit{conf}(\mathtt{C}) \Rightarrow \mathit{conf}(\mathtt{D})}{\mathtt{C} \preceq \mathtt{D}}$$

Method type lookup:

$$\frac{\mathit{mdef}(\mathtt{m},\ \mathtt{C}) = \mathtt{D} \quad [\mathtt{anon}]\ \mathtt{B\ m}(\overline{\mathtt{B\ x}})\,\{\,\mathtt{return\,e;}\,\} \in \mathit{methods}(\mathtt{D})}{\mathit{mtype}(\mathtt{m},\ \mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}}$$

Method body look-up:

$$\frac{\mathit{mdef}(\mathtt{m},\ \mathtt{C}) = \mathtt{D} \quad [\mathtt{anon}]\ \mathtt{B\ m}(\overline{\mathtt{B\ x}})\,\{\,\mathtt{return\,e;}\,\} \in \mathit{methods}(\mathtt{D})}{\mathit{mbody}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e})}$$

Valid method overriding:

$$\frac{\begin{array}{c}\text{either } \mathtt{m} \text{ is not defined in } \mathtt{D} \text{ or any of its parents, or}\\ \mathit{mtype}(\mathtt{m},\ \mathtt{C}) = \overline{\mathtt{C}} \to \mathtt{C}_0 \quad \mathit{mtype}(\mathtt{m},\ \mathtt{D}) = \overline{\mathtt{C}} \to \mathtt{C}_0 \quad (\mathit{anon}(\mathtt{m},\mathtt{D}) \Rightarrow \mathit{anon}(\mathtt{m},\mathtt{C}))\end{array}}{\mathit{override}(\mathtt{m},\mathtt{C},\mathtt{D})}$$

Fig. 8. ConfinedFJ: Auxiliary definitions.

Anonymous method:

$$\frac{\mathit{mdef}(\mathtt{m},\ \mathtt{C}_0) = \mathtt{C}'_0 \quad \mathtt{anon\ C\ m}\ (\overline{\mathtt{C\ x}})\ \{\ldots\} \in \mathit{methods}(\mathtt{C}'_0)}{\mathit{anon}(\mathtt{m},\mathtt{C}_0)}$$

Anonymity constraints:

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

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

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

Fig. 9. ConfinedFJ: Syntactic Anonymity Constraints.

Expression typing:

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

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

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

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

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{D} \quad \mathtt{D} \preceq \mathtt{C}}{\Gamma \vdash (\mathtt{C})\ \mathtt{e} : \mathtt{C}} \qquad \text{(T-UCAST)}$$

Method typing:

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

Class typing:

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{D}) = (\overline{\mathtt{D}\ \mathtt{g}}) \quad \mathtt{K} = \mathtt{C}(\overline{\mathtt{D}\ \mathtt{g}},\ \overline{\mathtt{C}\ \mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ \mathit{visible}(\mathtt{D}, \mathtt{C}) \quad (\mathit{conf}(\mathtt{D}) \Rightarrow \mathit{conf}(\mathtt{C})) \quad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C} \lhd \mathtt{D}\end{array}}{[\mathtt{conf}]\ \mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{C}\ \mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \qquad \text{(T-CLASS)}$$

Static expression visibility:

$$\frac{\mathit{visible}(\Gamma(\mathtt{x}), \mathtt{C})}{\Gamma \vdash \mathit{visible}(\mathtt{x}, \mathtt{C})} \qquad \frac{\Gamma \vdash \mathtt{e.f_i} : \mathtt{C'} \quad \mathit{visible}(\mathtt{C'}, \mathtt{C}) \quad \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})}{\Gamma \vdash \mathit{visible}(\mathtt{e.f_i}, \mathtt{C})}$$

$$\frac{\mathit{visible}(\mathtt{C'}, \mathtt{C}) \quad \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})}{\Gamma \vdash \mathit{visible}((\mathtt{C'})\ \mathtt{e}, \mathtt{C})} \qquad \frac{\mathit{visible}(\mathtt{C'}, \mathtt{C}) \quad \forall \mathtt{i},\ \Gamma \vdash \mathit{visible}(\mathtt{e_i}, \mathtt{C})}{\Gamma \vdash \mathit{visible}(\mathtt{new\ C'}(\overline{\mathtt{e}}), \mathtt{C})}$$

$$\frac{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{C'} \quad \mathit{visible}(\mathtt{C'}, \mathtt{C}) \quad \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}) \quad \forall \mathtt{i},\ \Gamma \vdash \mathit{visible}(\mathtt{e_i}, \mathtt{C})}{\Gamma \vdash \mathit{visible}(\mathtt{e.m}(\overline{\mathtt{e}}), \mathtt{C})}$$

Fig. 10. ConfinedFJ: Typing rules.

types $\overline{\mathtt{D}}$ and argument types $\overline{\mathtt{C}}$, relation $\overline{\mathtt{C}} \preceq \overline{\mathtt{D}}$ must hold. By definition of $\mathtt{C_i} \preceq \mathtt{D_i}$, if $\mathtt{C_i}$ is confined then $\mathtt{D_i}$ is confined as well.

- Rule T-INVK prevents widening of confined arguments to public parameters by enforcing safe subtyping of argument types with respect to parameter types. In order to prevent implicit widening of the receiver, we consider two cases. Assume that the receiver has type $\mathtt{C_0}$ and the method `m` is defined in $\mathtt{D_0}$, then it must either be the case that $\mathtt{C_0}$ is a safe subtype of $\mathtt{D_0}$ or that `m` has been declared anonymous in $\mathtt{D_0}$.
- Rule T-UCAST prevents casting a confined type to a public type by enforcing safe subtyping. The rule needs only cover upcasts as ConfinedFJ does not allow downcasts. Downcasts are not relevant as they preserve confinement, this comes the fact that by Rule T-CLASS a confined class cannot have a public subclass. Casting an object of public class to confined type will thus result in runtime exception.

4.3.2 Typing rules for methods and classes

Figure 10 also gives rules for typing methods and classes.

- Rule T-METHOD places the following constraints on a method `m` defined in class $\mathtt{C_0}$ with body `e`. The type `D` of `e` must be a safe subtype of the method's declared type `C`. The method must preserves anonymity declarations. If `m` is declared anonymous, `e` must comply with the corresponding restrictions. The most interesting constraint is the visibility enforced on the body by $\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C_0})$, which is defined recursively over the structure of terms. It ensures that the types of all subexpressions of `e` are visible from the defining class $\mathtt{C_0}$. In particular, the method parameters used in the method body `e` must have types visible in $\mathtt{C_0}$.
- Rule T-CLASS requires that if class `C` extends `D` then `D` be visible in `C` and if `D` is confined, then so is `C`. Rule T-CLASS allows the fields of a class `C` to have types not visible in `C`, but the constraint of $\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})$ in Rule T-METHOD prohibits the method of `C` from accessing such fields.

The class table CT is well-typed if all classes in CT are well-typed. For the rest of this paper, we assume CT to be well-typed.

4.3.3 Relation to the informal rules

We now relate the rules given in section 3 with the ConfinedFJ type system. The effect of Rule $\mathscr{C}1$, which limit the visibility of fields if their type is confined, is obtained as a side effect of the visibility constraint as it prevents code defined in another package from accessing a confined field. ConfinedFJ could be extended with field and method access modifier without significantly changing the type system. The expression typing rules enforce confinement rules $\mathscr{C}3$ and $\mathscr{C}5$ by ensuring that methods invoked on an object of confined type are either anonymous or defined in a confined class, and that widening is confinement preserving. Rule $\mathscr{C}2$ uses access modifiers to limit the use of confined types; and the same effect is achieved by the visibility constraint $\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})$ on expression part of T-METHOD. Rule $\mathscr{C}4$, which states that subclassing is confinement preserving, is enforced by T-CLASS. Rule $\mathscr{C}6$, which states that overriding is anonymity preserving, is enforced by T-METHOD. Finally the anonymity constraint of Rule $\mathscr{A}1$ is obtained by the *anon* predicate in the antecedent of T-METHOD.

4.4 Two ConfinedFJ examples

Consider the following stripped down version of a hash table class written in ConfinedFJ. The hash table is represented by a class `p.Table` defined in some package `p` that holds a single bucket of class `p.Buck`. The bucket can be obtained by calling the method `get()` on a table, the bucket's data can then be obtained by calling `getData()`. In this example, buckets are confined but they extend a public class `p.Cell`. The interface of `p.Table.get()` specifies that the method's return type is `p.Cell`, this is valid as that class is public. In this example a factory class, named `p.Factory`, is needed to create instances of `p.Table` because the table's constructor expects a bucket and since buckets are confined, they cannot be instantiated outside of their defining package.

This program does not preserve confinement as the body of the `p.Table.get()` method returns an instance of a confined class in violation of the widening rule. The breach can be exhibited by constructing a class `o.Breach` in package `o` which creates a new table and retrieves its bucket.

```
class o.Breach ◁ l.Object {
   l.Object main () { return new p.Factory().table().get(); }
}
```

The expression `new o.Breach().main()` thus evaluates in three reduction steps to `new p.Buck()` exposing the confined class to code defined in another package. This example is not typable in the ConfinedFJ type system. The expression `p.Table.get()` does not type-check because Rule T-METHOD requires the type of

```
class p.Table ◁ l.Object {
   p.Buck buck;
   Table(p.Buck buck) { super(); this.buck = buck; }
   p.Cell get() { return this.buck; }
}
class p.Cell ◁ l.Object {
   l.Object data;
   l.Object getData() { return this.data; }
}
```

```
conf class p.Buck ◁ p.Cell {
   p.Buck() { super(); }
}

class p.Factory ◁ l.Object {
   p.Factory() { super(); } }
   p.Table table() { return new p.Table( new p.Buck() ); }
}
```

the expression returned by the method to be a safe subtype of the method's declared return type. The expression has the confined type `p.Buck` while the declared return type is the public type `p.Cell`.

In another prototypical breach of confinement, consider the following situation in which the confined class `p.Self` extends a `o.Broken` parent class that resides in package `o`. Assume further that the class inherits its parent's code for the `reveal()` method.

```
conf class p.Self ◁ o.Broken {
   p.Self() { super(); }
}

class p.Main ◁ l.Object {
   p.Main() { super(); }
   l.Object get() { return new p.Self().reveal(); }
}
```

Inspection of this code does not reveal any breach of confinement. But if we widen the scope of our analysis to the `o.Broken` class, we may see:

```
class o.Broken ◁ l.Object {
   o.Broken() { super(); }
   l.Object reveal() { return this; }
}
```

Invoking `reveal()` on an instance of `p.Self` will return a reference to the object itself. This does not type-check because the invocation of `reveal()` in `p.Main.get()` violates the Rule T-Invk (due to that the non-anonymous method `reveal()`, inherited from a public class `o.broken`, is invoked on an object of a confined type `p.Self`). The method `reveal()` cannot be declared anonymous as the method returns `this` directly.

5 Confinement properties

In this section, we describe properties of ConfinedFJ and prove the Confinement Theorem. During the execution of a well-typed program, a confined object can be

$$\frac{\emptyset \vdash \mathtt{e.f_i} : \mathtt{C'} \quad \mathit{visible}(\mathtt{C'},\mathtt{C}) \quad (\mathtt{e} = \mathtt{new}\ \mathtt{C_0}(\overline{\mathtt{u}}) \ \vee\ \mathit{visible}_{\mathtt{C_0}}(\mathtt{e},\mathtt{C}))}{\mathit{visible}_{\mathtt{C_0}}(\mathtt{e.f_i},\mathtt{C})}$$

$$\frac{\mathit{visible}(\mathtt{C'},\mathtt{C}) \quad \mathit{visible}_{\mathtt{C_0}}(\mathtt{e},\mathtt{C})}{\mathit{visible}_{\mathtt{C_0}}((\mathtt{C'})\ \mathtt{e},\mathtt{C})} \qquad \frac{\mathit{visible}(\mathtt{C'},\mathtt{C}) \quad \forall \mathtt{i},\ \mathit{visible}_{\mathtt{C_0}}(\mathtt{e_i},\mathtt{C})}{\mathit{visible}_{\mathtt{C_0}}(\mathtt{new}\ \mathtt{C'}(\overline{\mathtt{v}}\,\overline{\mathtt{e}}),\mathtt{C})}$$

$$\frac{\emptyset \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{C'} \quad \mathit{visible}(\mathtt{C'},\mathtt{C}) \quad (\mathtt{e} = \mathtt{new}\ \mathtt{C_0}(\overline{\mathtt{u}}) \ \vee\ \mathit{visible}_{\mathtt{C_0}}(\mathtt{e},\mathtt{C})) \quad \forall \mathtt{i},\ \emptyset \vdash \mathit{visible}(\mathtt{e_i},\mathtt{C})}{\mathit{visible}_{\mathtt{C_0}}(\mathtt{e.m}(\overline{\mathtt{e}}),\mathtt{C})}$$

Fig. 11. ConfinedFJ: Runtime expression visibility.

accessed only by the methods that it "trusts". The trusted methods of an object of the type C include the methods defined in the package of C and the anonymous methods inherited by C. Thus, to satisfy the confinement properties, the evaluation of a call to any method m may only contain accesses to either objects of public types, objects of confined types defined in the package containing m, or the receiver object of the call in case m is anonymous and the receiver object is confined. In ConfinedFJ, we define access to an object to mean field selection and method invocation.

5.1 Runtime expression visibility

We check whether an expression satisfies the confinement properties using the recursive predicate $\mathit{visible}_{\mathtt{C_0}}(\mathtt{e},\mathtt{C})$ defined in Figure 11. Consider an expression e reduced from a method call $\mathtt{v.m}(\overline{\mathtt{v}})$, where v is the receiver that has type $\mathtt{C_0}$ and m is defined in the class C, we say that if $\mathit{visible}_{\mathtt{C_0}}(\mathtt{e},\mathtt{C})$ is true, then e satisfies confinement. We write $\overline{\mathtt{v}}\,\overline{\mathtt{e}}$ to denote a sequence of values followed by expressions.

For $\mathit{visible}_{\mathtt{C_0}}(\mathtt{e},\mathtt{C})$ to be true, the type of e has to be visible in C. In addition, if e has the form $(\mathtt{C'})\ \mathtt{e'}$, then $\mathit{visible}_{\mathtt{C_0}}(\mathtt{e'},\mathtt{C})$ must also hold; if e has the form $\mathtt{e'.f}$ or $\mathtt{e'.m}(\overline{\mathtt{e}})$, then either $\mathit{visible}_{\mathtt{C_0}}(\mathtt{e'},\mathtt{C})$ or $\mathtt{e'}$ has the form $\mathtt{new}\ \mathtt{C_0}(\overline{\mathtt{u}})$ for some $\overline{\mathtt{u}}$. The latter is relevant for anonymous methods because if an anonymous method is called on an object v of confined type $\mathtt{C_0}$ while the method is defined in a class C outside the package of $\mathtt{C_0}$, then the variable this in the method body is substituted by v but the type of v is not visible in C. The constraints allow this case as long as v is only used as the receiver of method calls and for field selects.

We also observe that for a fully evaluated object, $\mathtt{e} = \mathtt{new}\ \mathtt{C'}(\overline{\mathtt{v}})$, $\mathit{visible}_{\mathtt{C_0}}(\mathtt{e},\mathtt{C})$ only require $\mathtt{C'}$ to be visible in C. This should be contrasted with the situation where $\mathtt{e} = \mathtt{new}\ \mathtt{C'}(\overline{\mathtt{e}})$, in which case we must also have $\forall \mathtt{i},\ \mathit{visible}_{\mathtt{C_0}}(\mathtt{e_i},\mathtt{C})$. The intuition is that the syntax of the calculus does not differentiate between constructed objects and the expressions that construct them. Confinement must be checked only before an object is constructed. Thus before a new expression of the form $\mathtt{new}\ \mathtt{C'}(\overline{\mathtt{e}})$ is reduced to a fully-evaluated object, we need to check $\overline{\mathtt{e}}$ for any violations of confinement properties within the context of the method that contains the new expression. Since we have a by-value semantics, such a new expression may not be transfered to another method before it is fully evaluated. However, a fully-evaluated object of the form $\mathtt{new}\ \mathtt{C'}(\overline{\mathtt{v}})$ could sent to a method of a class C outside the package of $\mathtt{C'}$ if $\mathtt{C'}$

is public. In this case, the objects in the fields of $\mathtt{C}'$ may not be visible in $\mathtt{C}$. Thus, we only require that $\mathtt{C}'$ be visible in $\mathtt{C}$. This requirement is sufficient for preserving confinement properties since the confined objects in the fields of $\mathtt{C}'$ are not accessible in $\mathtt{C}$ because these fields are package-scoped.

5.2 Well-typed program and confinement

Recall that we model a program's execution with a stack. Each frame in P consists of a tuple $\mathtt{v\ m\ e}$, that corresponds to an invocation of method $\mathtt{m}$ on the object $\mathtt{v}$ and $\mathtt{e}$ is the expression reduced from the method call. Also recall that each method invocation will create a new frame. We say that a program P is well-typed if the expression $\mathtt{e}$ in each frame of P is well-typed and the type of $\mathtt{e}$ is a safe subtype of the type of the expression $\mathtt{e}'$, where $E[\mathtt{e}']$ is in the previous frame.

Definition 1 (*Well-typed*)
A program P is well-typed iff $\vdash P$ as defined below.

$$\frac{\emptyset \vdash \mathtt{e} : \mathtt{C}}{\vdash \mathit{nil}\ .\ \mathtt{v\ m\ e}} \qquad \frac{\vdash P\ .\ \mathtt{v\ m}\ E[\mathtt{e}] \quad \emptyset \vdash \mathtt{e} : \mathtt{C} \quad \emptyset \vdash \mathtt{e}' : \mathtt{C}' \quad \mathtt{C}' \preceq \mathtt{C}}{\vdash P\ .\ \mathtt{v\ m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{e}'}$$

We say that a program satisfies confinement if each frame $\mathtt{v\ m\ e}$ in the program satisfies the runtime expression visibility constraint. That is, if the method $\mathtt{m}$ invoked on $\mathtt{v}$ of type $\mathtt{C}$ is defined in the class $\mathtt{C}'$, then the predicate $\mathit{visible}_{\mathtt{C}}(\mathtt{e},\mathtt{C}')$ is true.

Definition 2 (*Confinement Satisfaction*)
A program $P = \mathtt{v}_1\ \mathtt{m}_1\ \mathtt{e}_1 \ldots \mathtt{v}_n\ \mathtt{m}_n\ \mathtt{e}_n$ satisfies confinement iff for all $\mathtt{i} \in [\mathtt{1},\mathtt{n}]$ we have $\mathit{visible}_{\mathtt{C}}(\mathtt{e}_\mathtt{i},\mathtt{C}')$, where $\mathtt{v}_\mathtt{i} = \mathtt{new\ C}(\overline{\mathtt{v}})$, $\mathit{mdef}(\mathtt{m}_\mathtt{i},\ \mathtt{C}) = \mathtt{C}'$.

We prove the properties of confined objects in Theorem 2. We show that if a well-typed program initially satisfies confinement, then it will always satisfy confinement during execution. We also prove the subject reduction lemmas for expressions and programs, and state the progress lemma for programs. For the subject reduction lemma, we show that an expression of non-confined type will not be reduced to an expression of confined type. Theorem 1 states that a well-typed program will not get stuck.

5.3 Subject reduction

Recall that, we assume the class table CT to be well-typed, which means that all classes in CT are well-typed.

Lemma 2
If $\mathit{mtype}(\mathtt{m},\ \mathtt{C}_0) = \overline{\mathtt{C}} \rightarrow \mathtt{C}$, $\mathit{mbody}(\mathtt{m},\ \mathtt{C}_0) = (\overline{\mathtt{x}},\ \mathtt{e})$, and $\mathit{mdef}(\mathtt{m},\ \mathtt{C}_0) = \mathtt{C}_0'$, then there exists some $\mathtt{C}' \preceq \mathtt{C}$ such that $\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C}_0' \vdash \mathtt{e} : \mathtt{C}'$.

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

Lemma 3
If $\overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e} : \mathtt{C}$, $\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{A}}$, $\overline{\mathtt{A}} \preceq \overline{\mathtt{B}}$, then $\emptyset \vdash [{}^{\overline{\mathtt{v}}}/_{\overline{\mathtt{x}}}]\mathtt{e} : \mathtt{C}'$ for some $\mathtt{C}' \preceq \mathtt{C}$.

Proof
If $\mathtt{e} = \mathtt{x_i}$, then $\emptyset \vdash \mathtt{e} : \mathtt{C}$, $\mathtt{C} = \mathtt{B_i}$, $[\overline{\mathtt{v}}/\overline{\mathtt{x}}]\mathtt{e} = \mathtt{v_i}$, and $\emptyset \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{A_i}$, $\mathtt{A_i} = \mathtt{C}'$. By assumption, we have $\mathtt{A_i} \preceq \mathtt{B_i}$. For other cases where $\mathtt{e}$ is of the forms $\mathtt{e_0.m}(\overline{\mathtt{e}})$, $(\mathtt{C}')\ \mathtt{e_0}$, $\mathtt{e_0.f}$, or $\mathtt{new\ C}(\overline{\mathtt{e}})$, we can show that $\emptyset \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{C}$ by applying the induction hypothesis to the immediate subterms of $\mathtt{e}$. □

Lemma 4
If $\overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : \mathtt{D_0} \vdash \mathtt{e} : \mathtt{C}$, $\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{A}}$, $\overline{\mathtt{A}} \preceq \overline{\mathtt{B}}$, $\emptyset \vdash \mathtt{new\ C_0}(\overline{\mathtt{u}}) : \mathtt{C_0}$, $\mathtt{C_0} <: \mathtt{D_0}$, and $anon(\mathtt{e}, \mathtt{D_0})$, then $\emptyset \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}, {}^{\mathtt{new\ C_0}(\overline{\mathtt{u}})}/_{\mathtt{this}}]\mathtt{e} : \mathtt{C}'$ for some $\mathtt{C}' \preceq \mathtt{C}$.

Proof
From $anon(\mathtt{e}, \mathtt{D_0})$, we have $\mathtt{e} \neq \mathtt{this}$ and if $\mathtt{e}$ is a variable, then $\mathtt{e} \in \overline{\mathtt{x}}$ and the proof is similar to Lemma 3. If $\mathtt{e} = \mathtt{this.m}(\overline{\mathtt{e}})$, then from $anon(\mathtt{e}, \mathtt{D_0})$ we have $anon(\overline{\mathtt{e}}, \mathtt{D_0})$ and $anon(\mathtt{m}, \mathtt{D_0})$. From Rule T-INVK and applying induction hypothesis to $\overline{\mathtt{e}}$, we can show that if $\overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : \mathtt{D_0} \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}}$ then $\emptyset \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}, {}^{\mathtt{new\ C_0}(\overline{\mathtt{u}})}/_{\mathtt{this}}]\overline{\mathtt{e}} : \overline{\mathtt{C}}$ and $\overline{\mathtt{C}} \preceq \overline{\mathtt{D}}$. Since method-overriding preserves the anonymity of methods (from $override(\mathtt{m}, \mathtt{C_0}, \mathtt{D_0})$ in Rule T-METHOD) and from $\mathtt{C_0} <: \mathtt{D_0}$, we have that $anon(\mathtt{m}, \mathtt{D_0})$ implies $anon(\mathtt{m}, \mathtt{C_0})$. Thus, we can conclude from Rule T-INVK that $\emptyset \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}, {}^{\mathtt{new\ C_0}(\overline{\mathtt{u}})}/_{\mathtt{this}}]\mathtt{e} : \mathtt{C}$. For other cases, we can show $\emptyset \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}, {}^{\mathtt{new\ C_0}(\overline{\mathtt{u}})}/_{\mathtt{this}}]\mathtt{e} : \mathtt{C}$ by simple induction on $\mathtt{e}$. □

Lemma 5 (Subject reduction)
If P is well-typed and $P \rightarrow P'$ then P' is well-typed.

Proof
If $P' = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{e}''$, then to prove P' is well-typed, we need to show that $P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]$ is well-typed, $\mathtt{e}''$ is well-typed and its type is a safe subtype of the type of $\mathtt{e}$. In particular, if $P = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{e}'$ then it is sufficient to show that $\emptyset \vdash \mathtt{e}'' : \mathtt{C}''$ and $\mathtt{C}'' \preceq \mathtt{C}'$ where $\mathtt{C}'$ is the type of $\mathtt{e}'$. The reason is that if $\mathtt{C}$ is the type of $\mathtt{e}$, then from the assumption that P well-typed, we have $\mathtt{C}' \preceq \mathtt{C}$, and thus $\mathtt{C}'' \preceq \mathtt{C}'$ would imply $\mathtt{C}'' \preceq \mathtt{C}$.

If $P' = nil\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{e}''$, then we only need to show that $\mathtt{e}''$ is well-typed.

There are four cases depending on the reduction rule used.

(1) If the reduction from P to P' is by Rule R-FIELD, then P has the form of $P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]$, where $\mathtt{e} = \mathtt{new\ C_0}(\overline{\mathtt{v}}).\mathtt{f_i}$, and $P' = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}']$, where $\mathtt{e}' = \mathtt{v_i}$. Since P is well-typed, if $\emptyset \vdash \mathtt{e} : \mathtt{C_i}$, then from Rule T-FIELD, $\mathtt{new\ C_0}(\overline{\mathtt{v}})$ is well-typed and if $\emptyset \vdash \mathtt{v_i} : \mathtt{C}'_\mathtt{i}$, then $\mathtt{C}'_\mathtt{i} \preceq \mathtt{C_i}$ by Rule T-NEW. By induction on the type derivation of $E[\mathtt{e}]$, we can show that if $\emptyset \vdash E[\mathtt{e}] : \mathtt{C}$, then $\exists \mathtt{C}'$ such that $\emptyset \vdash E[\mathtt{e}'] : \mathtt{C}'$ and $\mathtt{C}' \preceq \mathtt{C}$. Therefore, P' is well-typed.

(2) If the reduction is by Rule R-CAST, then P has the form $P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]$, where $\mathtt{e} = (\mathtt{C})\ \mathtt{new\ C}'(\overline{\mathtt{v}})$, and $P' = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}']$, where $\mathtt{e}' = \mathtt{new\ C}'(\overline{\mathtt{u}})$, and from Rule T-UCAST, $\emptyset \vdash \mathtt{e}' : \mathtt{C}'$ and $\mathtt{C}' \preceq \mathtt{C}$. Thus, similar to the previous case we can show that P' is well-typed.

(3) If the reduction is by Rule R-INVK, then P has the form $P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]$, where $\mathtt{e} = \mathtt{v}'.\mathtt{m}'(\overline{\mathtt{v}'})$, $\mathtt{v}' = \mathtt{new\ C_0}(\overline{\mathtt{u}})$, $mbody(\mathtt{m},\ \mathtt{C_0}) = (\overline{\mathtt{x}},\ \mathtt{e_0})$, and $P' = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{e}'$, where $\mathtt{e}' = [\overline{\mathtt{v}}/\overline{\mathtt{x}}, {}^{\mathtt{v}'}/_{\mathtt{this}}]\mathtt{e_0}$. If $mtype(\mathtt{m},\ \mathtt{C_0}) = \overline{\mathtt{C}} \rightarrow \mathtt{C}$, $mdef(\mathtt{m},\ \mathtt{C_0}) = \mathtt{C}'_0$, $\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C}'_0 \vdash \mathtt{e_0} : \mathtt{C}'$, and $\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{C}'}$, then $\mathtt{C}' \preceq \mathtt{C}$, $\overline{\mathtt{C}'} \preceq \overline{\mathtt{C}}$, and either $\mathtt{C_0} \preceq \mathtt{C}'_0$ or $anon(\mathtt{m}, \mathtt{C}'_0)$.

From Lemma 2, 3, and 4, and Rule R-INVK, $\exists \texttt{C}''$ such that $\emptyset \vdash \texttt{e}' : \texttt{C}''$ and $\texttt{C}'' \preceq \texttt{C}'$. Thus, $\texttt{C}'' \preceq \texttt{C}$ and P' is well-typed.

(4) If the reduction is by Rule R-RET, then P has the form of $P''\ .\ \texttt{v m}\ E[\texttt{e}]\ .\ \texttt{v}'\ \texttt{m}'\ \texttt{v}''$ and $P' = P''\ .\ \texttt{v m}\ E[\texttt{v}'']$. Since P is well-typed, if $\emptyset \vdash \texttt{e} : \texttt{D}$ and $\emptyset \vdash \texttt{v}'' : \texttt{D}'$, then $\texttt{D}' \preceq \texttt{D}$. By Lemma 6, if $\emptyset \vdash E[\texttt{e}] : \texttt{C}$, then $\emptyset \vdash E[\texttt{v}''] : \texttt{C}'$ and $\texttt{C}' \preceq \texttt{C}$. Therefore, P' is well-typed. □

Lemma 6
If $\emptyset \vdash E[\texttt{e}] : \texttt{C}$, $\emptyset \vdash \texttt{e} : \texttt{D}$, and $\emptyset \vdash \texttt{e}' : \texttt{D}'$, where $\texttt{D}' \preceq \texttt{D}$, then $\exists \texttt{C}'$ such that $\emptyset \vdash E[\texttt{e}'] : \texttt{C}'$ and $\texttt{C}' \preceq \texttt{C}$.

The proof is by induction on the structure of $E[\texttt{e}]$.

5.4 Progress

A terminating computation reduces to the form of *nil* . $\texttt{v m v}'$. An irreducible program P is deemed *stuck* if it is not of the form *nil* . $\texttt{v m v}'$. We show that well-typed programs do not get stuck.

Lemma 7
If P is well-typed and not in the form of *nil* . $\texttt{v m v}'$, then there exist P' such that $P \rightarrow P'$.

Theorem 1 (*Soundness*)
A well-typed program will not get stuck.

Proof
Immediate from Lemma 5 and 7. □

5.5 Confinement Theorem

The following lemma shows that the reduction of a well-typed program preserves confinement.

Lemma 8
If P is well-typed and satisfies confinement, and $P \rightarrow P'$, then P' satisfies confinement.

Proof
(1) Suppose the reduction from P to P' is by Rule R-FIELD or R-CAST. If $P = P''\ .\ \texttt{v m}\ E[\texttt{e}]$, $\texttt{e} \neq \texttt{v}'.\texttt{m}'(\overline{\texttt{u}})$, and $P' = P''\ .\ \texttt{v m}\ E[\texttt{e}']$, then by the assumption that P is well-typed, $\exists \texttt{C}$ such that $\emptyset \vdash \texttt{e} : \texttt{C}$, and $\mathit{visible}_{\texttt{C}_0}(\texttt{e}, \texttt{C}'_0)$, where $\texttt{v} = \texttt{new C}_0(\overline{\texttt{u}}')$ and $\mathit{mdef}(\texttt{m}, \texttt{C}_0) = \texttt{C}'_0$. From Lemma 5, P' is well-typed and $\exists \texttt{C}'$ such that $\emptyset \vdash \texttt{e}' : \texttt{C}'$ where $\texttt{C}' \preceq \texttt{C}$. From $\mathit{visible}_{\texttt{C}_0}(\texttt{e}, \texttt{C}'_0)$, we have $\mathit{visible}(\texttt{C}, \texttt{C}'_0)$. Since $\texttt{C}' \preceq \texttt{C}$, if $\texttt{C}'$ is confined, then so is $\texttt{C}$. From $\texttt{C}' <: \texttt{C}$ and Rule T-CLASS, we have $\mathit{visible}(\texttt{C}, \texttt{C}')$, which implies that if $\texttt{C}$ is confined then $\mathit{packof}(\texttt{C}) = \mathit{packof}(\texttt{C}')$. From $\mathit{visible}(\texttt{C}, \texttt{C}'_0)$, if $\texttt{C}$ is confined, then $\mathit{packof}(\texttt{C}) = \mathit{packof}(\texttt{C}'_0)$. Thus, if $\texttt{C}'$ is confined, then $\mathit{packof}(\texttt{C}') = \mathit{packof}(\texttt{C}) = \mathit{packof}(\texttt{C}'_0)$. Therefore, we have $\mathit{visible}(\texttt{C}', \texttt{C}'_0)$

If the reduction from P to P' is by Rule R-FIELD, then $\mathtt{e}$ has the form of $\mathtt{new\ D(\overline{u}).f_i}$ and $\mathtt{e}' = \mathtt{u_i}$. Thus, $\mathit{visible}_{\mathtt{C_0}}(\mathtt{u_i}, \mathtt{C}'_0)$. If the reduction is by Rule R-CAST, then $\mathtt{e}$ in the form of $\mathtt{(C)\ u}$ and $\mathtt{e}' = \mathtt{u}$. Thus, $\mathit{visible}_{\mathtt{C_0}}(\mathtt{u}, \mathtt{C}'_0)$. Therefore, we conclude that $\mathit{visible}_{\mathtt{C_0}}(\mathtt{e}', \mathtt{C}'_0)$. Since P is well-typed, we have $\mathit{visible}_{\mathtt{C_0}}(E[\mathtt{e}], \mathtt{C}'_0)$. By simple induction, we can show that $\mathit{visible}_{\mathtt{C_0}}(E[\mathtt{e}'], \mathtt{C}'_0)$. Thus, P' satisfies confinement.

(2) Suppose the reduction is by Rule R-INVK. If P has the form $P''\ .\ \mathtt{v\ m}\ E[\mathtt{e}]$, $\mathtt{e} = \mathtt{v'.m'(\overline{v})}$, $\mathtt{v}' = \mathtt{new\ C_0(\overline{u})}$, $\mathit{mbody}(\mathtt{m}', \mathtt{C_0}) = (\overline{\mathtt{x}}, \mathtt{e_0})$, then $P' = P''\ .\ \mathtt{v\ m}\ E[\mathtt{e}]\ .\ \mathtt{v'\ m'\ e'}$, where $\mathtt{e}' = [\overline{\mathtt{v}}/_{\overline{\mathtt{x}}}, \mathtt{v}'/_{\mathtt{this}}]\mathtt{e_0}$.

Suppose $\mathit{mtype}(\mathtt{m}', \mathtt{C_0}) = \overline{\mathtt{C}} \to \mathtt{C}$, $\mathit{mdef}(\mathtt{m}', \mathtt{C_0}) = \mathtt{C}'_0$, and $\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{C}'}$. From Rule T-METHOD, we have $\Gamma \vdash \mathit{visible}(\mathtt{e_0}, \mathtt{C}'_0)$ and $\Gamma \vdash \mathtt{e_0} : \mathtt{C}$ where $\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C}'_0$ and $\overline{\mathtt{C}'} \preceq \overline{\mathtt{C}}$.

If $\mathtt{C_0} \preceq \mathtt{C}'_0$, then from Lemma 3, we have that for each immediate subterm $\mathtt{e}'_0$ of $\mathtt{e_0}$, if $\Gamma \vdash \mathtt{e}'_0 : \mathtt{D}$, then $\emptyset \vdash [\overline{\mathtt{v}}/_{\overline{\mathtt{x}}}, \mathtt{v}'/_{\mathtt{this}}]\mathtt{e}'_0 : \mathtt{D}'$, $\mathtt{D}' \preceq \mathtt{D}$, and $\mathit{visible}(\mathtt{D}, \mathtt{C}'_0)$ implies $\mathit{visible}(\mathtt{D}', \mathtt{C}'_0)$. Thus, from $\Gamma \vdash \mathit{visible}(\mathtt{e_0}, \mathtt{C}'_0)$, we can show by induction that $\mathit{visible}_{\mathtt{C_0}}(\mathtt{e}', \mathtt{C}'_0)$ is true.

If $\mathtt{C_0} \not\preceq \mathtt{C}'_0$, then from Rule T-INVK, we have $\mathit{anon}(\mathtt{m}', \mathtt{C}'_0)$, which implies that the variable $\mathtt{this}$ can occur only in the subterms of $\mathtt{e_0}$ in the form of $\mathtt{this.f}$ or $\mathtt{this.m'(\overline{e})}$ (where $\mathtt{e_i} \neq \mathtt{this}$, $\forall \mathtt{i}$). Thus, the object $\mathtt{v}'$ can be only in the subterms of $\mathtt{e}'$ in the forms of $\mathtt{v'.f}$ or $\mathtt{v'.m'(\overline{e})}$ (where $\mathtt{e_i}$ is not of the form $\mathtt{new\ C_0(\overline{u})}$, $\forall \mathtt{i}$). From Lemma 4 and $\Gamma \vdash \mathit{visible}(\mathtt{e_0}, \mathtt{C}'_0)$, we can prove $\mathit{visible}_{\mathtt{C_0}}(\mathtt{e}', \mathtt{C}'_0)$ by simple induction. Thus, P' satisfies confinement.

(3) If the reduction is by Rule R-RET, then P has the form of $P''\ .\ \mathtt{v\ m}\ E[\mathtt{e}]\ .\ \mathtt{v'\ m'\ v''}$ and $P' = P''\ .\ \mathtt{v\ m}\ E[\mathtt{v}'']$. From Lemma 5, P' is well-typed. Thus, if $\emptyset \vdash \mathtt{e} : \mathtt{C}$ and $\emptyset \vdash \mathtt{v}'' : \mathtt{C}'$, then $\mathtt{C}' \preceq \mathtt{C}$. Suppose $\mathtt{v} = \mathtt{new\ C_0(\overline{v})}$ and $\mathit{mdef}(\mathtt{m}, \mathtt{C_0}) = \mathtt{C}'_0$. Since P satisfies confinement, we have $\mathit{visible}_{\mathtt{C_0}}(E[\mathtt{e}], \mathtt{C}'_0)$, which implies $\mathit{visible}_{\mathtt{C_0}}(\mathtt{e}, \mathtt{C}'_0)$ and $\mathit{visible}(\mathtt{C}, \mathtt{C}'_0)$. Hence, we have $\mathit{visible}(\mathtt{C}', \mathtt{C}'_0)$ and $\mathit{visible}_{\mathtt{C_0}}(\mathtt{v}'', \mathtt{C}'_0)$. It is clear that $\mathit{visible}_{\mathtt{C_0}}(E[\mathtt{v}''], \mathtt{C}'_0)$ is true; thus, P' satisfies confinement. □

Theorem 2 (*Confinement*)
If P is well-typed and satisfies confinement, and $P \to^* P'$ then P' satisfies confinement.

Proof
Immediate from Lemma 5 and Lemma 8. □

The Confinement Theorem states that a well-typed program that initially satisfies confinement preserves confinement. Intuitively, this means that that during the execution of a well-typed program, all the objects that are accessed within the body of a method 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, and if this occurs the use of `this` is restricted to the receiver position.

6 Generics and confinement

The lack of support for collections and reusable confined classes was identified early on as a significant issue for practical adoption of confined types (Vitek & Bokowski,

$\mathscr{C}7$ *A generic type or type variable cannot be widened to a type containing a different set of type variables.*

$\mathscr{C}8$ *A method invoked on an expression of type* `T` *must either be defined in a type with the same set of type variables as that in* `T` *or be an anonymous method.* **T1**

Fig. 12. Genericity confinement constraint.

2001). In this section, we extend the confinement property to generic types to allow writing generic classes which are, in and of themselves, not confined, but become confined if instantiated with confined arguments. ConfinedFJ is extended with support for generic types, following FGJ (Igarashi *et al.*, 2001), and renamed ConfinedFGJ. The main departure from ConfinedFJ is that a generic type with confined type parameters is also treated as confined. We not only need to prevent unsafe reference widening for confined types but also for generic types with variable type parameters. Therefore, besides the first six confinement rules already presented, we require the following:

Rules $\mathscr{C}5$ and $\mathscr{C}7$ combined enforces a subtyping relation that prevents unsafe reference widening. Recall that $\mathscr{C}5$ prevents widening for non-generic confined types. Since a generic class can be instantiated with confined type parameters, unsafe reference widening can happen after generic type instantiation. For example consider a class `Vector<X>` and a method that assigns a `Vector` to a variable of `l.Object` type. If the class `Vector<X>` is ever instantiated with a confined type, `C`, then the assignment of a `Vector<C>` to an `l.Object` variable leads to unsafe reference widening. Rule $\mathscr{C}7$ prevents such unsafe widenings. For instance, widening a reference from `Vector<X>` to `Map<X>` is safe if the class is defined as `Vector<X> ◁ Map<X>`.

Rule $\mathscr{C}8$ supplements $\mathscr{C}3$ so that method calls on a receiver object of a generic type with confined type parameters will not leak references to the receiver object to untrusted code.

To see the advantage of confined generic classes, consider a generic linked list class. If we desire to use the class to hold both confined and non-confined objects, it should be defined as follows.

```
class p.List<X ◁ l.Object> ◁ l.Object {
   X val;
   p.List<X> next;
   p.List(X val, p.List<X> next) {
      super(); this.val=val; this.next=next;
   }
}
```

With this definition, lists can be used in several contexts. For instance, it is possible to use the same list class twice within the same package, once with a confined type, thus turning that instantiation of the list type into a confined type, and once with a non-confined type. The following example illustrates this. Classes `A` and `B` reside in package `q`, 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 q.A ◁ l.Object {
   p.List<A> show;
   p.List<B> hide;
   ...
}

conf class q.B ◁ l.Object {
   q.B() { super(); }
}
```

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. This type variable need not be used in the body of the class, it will merely serve as a marker. Reuse is thus obtained by instantiating the class in each of the packages with a 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 ◁ l.Object> ◁ l.Object {
   ...
}
```

The type variable X is not used by the implementation of a.Key, and the class can be confined in any package as long as it is instantiated with a confined type, *e.g.* new a.Key<q.B>(). Type parameters allow reusing several related classes at the same time. For example, suppose the classes a.PublicKey<X> and a.PrivateKey<X> both extend the class a.Key<X>. Then, we may instantiate the three classes with a confined type such as q.B and make them confined in a single package. Also, the widening of references from the type a.PublicKey<X> or a.PrivateKey<X> to a.Key<X> is safe as it will not allow references to leak. This use of type variables is very close to approaches based on ownership types.

The semantics of generic confined types is surprisingly simple. Any type variable will be treated as a confined type by the type system in the sense that unsafe reference widening will be forbidden for expressions of this type. Even though a generic type (a type that contains type variables) may not be confined in any package, unsafe reference widening should not be allowed for expressions of the type either. For example, consider a generic container class.

```
class p.Container<X ◁ l.Object> ◁ l.Object {
   X val;
   p.Container(X val) { this.val = val }
   l.Object get() { return this.val; }
   l.Object get2() { return this; }
}
```

The Container class has a method get() that returns the value of field val and a method get2() that returns the variable 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 l.Object respectively. The following example illustrates the case where X is replaced by confined types when Container is instantiated.

```
class q.A ◁ l.Object {
   p.Container<q.B> f = new p.Container<q.B>(new q.B());
   l.Object reveal() { return f.get(); }
}
```

The class q.A is allowed to access q.B, but at runtime the method reveal() calls get() and thus the type of the expression new q.B() is widened to l.Object.

Motivation for Rule $\mathscr{C}$4. In a generic class, the fields of variable type and the methods of variable return types should not be package-scoped, because otherwise, these fields and methods would not be accessible to other code if this class is instantiated outside its package, which would limit its reuse. If this class is instantiated with confined type parameters, then its public methods may return confined values and its public fields may reference confined values. However, this does not result in any confinement violation because a generic type N with confined type parameters is treated as a confined type and objects of this type are not accessible to code outside the its defining package. Moreover, by Rule $\mathscr{C}$4, the subtypes of N must also be confined so that outside code cannot access these public methods through inheritance either.

For example, if the generic class Vector<X> has a public method get that returns elements of the type X stored in the Vector. However, if we instantiate the vector class with a confined type p.C, then the object of the type Vector<p.C> is confined in p. By Rule $\mathscr{C}$4, any class p.D that extends Vector<p.C> must be confined. Therefore, even if the method get is public and returns values of a confined type, the code outside of p is not able to take advantage of this, since the objects of the type Vector<p.C> and its subtypes are not accessible to code outside of p except maybe to methods inherited by Vector<X>.

Even the methods inherited by Vector<X> cannot exploit the method get. If the inherited methods are defined in a class such as l.Object then it cannot access get. The method get cannot override any methods in l.Object since method overriding rule requires that overriding and overridden methods to have the same type signatures while the return type of get is a variable type X not found in l.Object. If Vector<X> inherits a class such as Map<X>, then the method in Map<X> may have access to the get method of Vector<X>. This is safe however, since there is not unsafe reference widening of the variable this to call a method of Map<X> on an object of the type Vector<X> (likewise, it is safe to call methods of Map<C> on objects of the type Vector<C>).

Motivation for Rule $\mathscr{C}$6. While the rule that ensures that method overriding preserves anonymity is not strictly necessary for ConfinedFJ as anonymity can be inferred (Grothoff *et al.*, 2001), it is however needed for ConfinedFGJ.

To illustrate the need, consider the following example, where the generic class `q.Naive` has a type variable `X` with upper bound `q.A`, a field `f` of the type `X`, and a method `reveal()` that calls method `m()` on `f`. Because `m()` in `q.A` is anonymous, the method body `this.f.m()` in `reveal()` is typable even though the receiver expression `this.f` is implicitly widened to the public type `q.A` in `m()`. In other words, it does not violate Rule $\mathscr{C}8$ to call `m` on a receiver expression `this.f` of type `X` with upper bound `q.A` because the method `m` in `q.A` is anonymous.

```
class q.Naive<X ◁ q.A> ◁ l.Object {
   X f;
   q.Naive (X f) { this.f = f; }
   l.Object reveal() { return this.f.m(); }
}

class q.A ◁ l.Object {
   anon l.Object m() { return new l.Object(); }
}

class q.B ◁ q.A {
   l.Object m() { return this; }
}

conf class q.C ◁ q.B { ... }
```

Suppose that overriding does not preserve the anonymity of methods. The method `m()` in the class `q.B` overrides `m()` of `q.A` but the former is not anonymous since it returns the self-reference `this` and widens it to `l.Object` type. Now consider the expression `new q.Naive<q.C>(new q.C()).reveal()` which instantiates the class `q.Naive` with the confined type `q.C` and calls its `reveal()` method. The expression is typable and its type is `l.Object`. However, the reduction steps of the expression show that it reduces to an object of the type `C`.

```
   new q.Naive<q.C>(new q.C()).reveal()

→ new q.Naive<q.C>(new q.C()).f.m()

→ new q.C().m() → new q.C()
```

What went wrong is that while evaluating the call to `reveal()` on the object `new q.Naive<q.C>(new q.C())`, the method `m()` of `q.B` is called on the confined object `new q.C()`. The method is not anonymous and it widens the reference to the confined object `new q.C()` to the public type `l.Object`. In ConfinedFJ, we could infer the anonymity of a method when it is invoked on a confined type. Here, the anonymity of a method sometimes has to be decided on type variables with concrete upper bounds. Without Rule $\mathscr{C}6$, the fact that the method `m` is anonymous relative to `q.A` does not implies it is anonymous relative to `X` which can be replaced by the subtypes of `q.A` such as `q.B` or `q.C`, Therefore, Rule $\mathscr{C}6$ is needed to ensure the anonymity of methods that are called on confined or generic types even if the methods are overridden in subclasses.

$$
\begin{array}{lcl}
\texttt{N} & ::= & \texttt{C}\langle\overline{\texttt{T}}\rangle \\
\texttt{T} & ::= & \texttt{X} \mid \texttt{N} \\
\texttt{L} & ::= & [\texttt{conf}]\ \texttt{class}\ \texttt{C}\langle\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\rangle\ \triangleleft\ \texttt{N}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \\
\texttt{K} & ::= & \texttt{C}(\overline{\texttt{T}}\ \overline{\texttt{f}})\{\ \texttt{super}(\overline{\texttt{f}});\ \texttt{this}.\overline{\texttt{f}}\ =\ \overline{\texttt{f}};\ \} \\
\texttt{M} & ::= & [\texttt{anon}]\ \texttt{T}\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\{\ \texttt{return}\ \texttt{e};\ \} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid (\texttt{N})\ \texttt{e} \mid \texttt{new}\ \texttt{N}(\overline{\texttt{e}}) \\
\texttt{v} & ::= & \texttt{new}\ \texttt{N}(\overline{\texttt{v}})
\end{array}
$$

Fig. 13. ConfinedFGJ: Syntax.

In ConfinedFJ, Rule $\mathscr{C}6$ only needs to be applied to methods inherited by confined types, in ConfinedFGJ, we also apply the rule to the methods inherited by the generic types since generic types could become confined after instantiation.

6.1 Syntax

The syntax for ConfinedFGJ is shown in Figure 13. For simplicity, we omit generic methods, thus only classes can have type parameters. Metavariables $\texttt{X}, \texttt{Y}$ range over type variables, $\texttt{N}, \texttt{W}$ range over concrete types, and $\texttt{S}, \texttt{T}$ range over both concrete types and type variables. In a class definition $[\texttt{conf}]\ \texttt{class}\ \texttt{C}\langle\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\rangle \triangleleft \texttt{N}\ \{\ldots\}$, the upper bounds for the type variables $\overline{\texttt{X}}$ are $\overline{\texttt{N}}$, which are always non-variable types. The type variable $\texttt{X}$ appearing in a generic class declaration can be instantiated with either public or confined type.

6.2 Dynamic Semantics

The dynamic semantics of ConfinedFGJ in Figure 14 is mostly identical to the ConfinedFJ rules presented in Figure 7.

6.3 Static semantics

The structure of the ConfinedFGJ static semantics is similar to that of the ConfinedFJ static semantics. Figure 15 gives subtyping rules, definitions for well-formed types, and other miscellaneous definitions. The subtyping rules are the same as those in Generic FJ. A generic type may contain type parameters that are confined in different packages. The set $\mathit{confPack}(\texttt{C}\langle\overline{\texttt{T}}\rangle)$ contains the set of packages that $\texttt{C}$ and $\overline{\texttt{T}}$ are confined in. The set $\mathit{Var}(\texttt{T})$ contains the set of type variables in $\texttt{T}$.

The partial order $\preceq$ on types represents the restricted subtyping relation that does not allow unsafe reference widening. As in FGJ, Δ denotes a type environment that maps type variables to their concrete type upper bounds. To have $\Delta \vdash \texttt{S} \preceq \texttt{T}$, we must have $\Delta \vdash \texttt{S} <: \texttt{T}$ and that $\mathit{confPack}(\texttt{S})$ is a subset of $\mathit{confPack}(\texttt{T})$; also, $\texttt{S}$, $\texttt{T}$ must contain the same set of type variables. With the last restriction, the partial

$$\frac{\mathtt{e} = \mathtt{new\ N}(\overline{\mathtt{v}}).\mathtt{f}_i \quad \mathit{fields}(\mathtt{N}) = (\overline{\mathtt{T}}\,\overline{\mathtt{f}})}{P\ .\ \mathtt{v\ m}\ E[\mathtt{e}] \ \rightarrow\ P\ .\ \mathtt{v\ m}\ E[\mathtt{v}_i]} \qquad \text{(GR-Field)}$$

$$\frac{\mathtt{e} = (\mathtt{N}')\ \mathtt{new\ N}(\overline{\mathtt{v}}) \quad \emptyset \vdash \mathtt{N} <: \mathtt{N}'}{P\ .\ \mathtt{v\ m}\ E[\mathtt{e}] \ \rightarrow\ P\ .\ \mathtt{v\ m}\ E[\mathtt{new\ N}(\overline{\mathtt{v}})]} \qquad \text{(GR-Cast)}$$

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

$$\frac{\mathtt{e} = \mathtt{v}'.\mathtt{m}'(\overline{\mathtt{v}})}{P\ .\ \mathtt{v\ m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{v}'' \ \rightarrow\ P\ .\ \mathtt{v\ m}\ E[\mathtt{v}'']} \qquad \text{(GR-Ret)}$$

Fig. 14. ConfinedFGJ: Dynamic semantics.

order on $\mathtt{S},\mathtt{T}$ still holds even if type variables in $\mathtt{S},\mathtt{T}$ are instantiated by confined types. For example, if $\Delta \vdash \mathtt{X} \preceq \mathtt{N}$ then it must be the case that $\mathtt{N} = \mathtt{C}\langle\overline{\mathtt{T}}\rangle$ with $\mathtt{X}$ being the only type variable in $\overline{\mathtt{T}}$. In this case, if $\mathtt{D}$ is confined and $\mathtt{N}' = [{}^{\mathtt{D}}/_{\mathtt{X}}]\mathtt{N}$, then $\Delta \vdash \mathtt{D} \preceq \mathtt{N}'$.

A type $\mathtt{T}$ is visible in the class $\mathtt{C}\langle\overline{\mathtt{T}}\rangle$ if for any package $\mathtt{p}$ that $\mathtt{T}$ is confined in, either $\mathtt{C}$ is defined in $\mathtt{p}$ or one of $\overline{\mathtt{T}}$ is confined in $\mathtt{p}$. If $\mathtt{T}$ is a concrete type that has the form $\mathtt{C}'\langle\overline{\mathtt{T}'}\rangle$, then this definition implies that $\mathtt{C}'$ is visible in $\mathtt{C}$ and the confined types in $\overline{\mathtt{T}'}$ must be visible in $\mathtt{C}$ or come from $\overline{\mathtt{T}}$. Note that this definition gives the appearance that a type variable is visible in any class, however since a type variable is not accessible outside its defining class, $\mathit{visible}(\mathtt{X},\mathtt{N})$ does not apply unless $\mathtt{X}$ is defined in $\mathtt{N}$.

Figure 16 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 instantiated by type arguments. In the rest of the paper, $\mathit{anon}(\mathtt{m},\mathtt{C}\langle\overline{\mathtt{T}}\rangle)$ is equivalent to $\mathit{anon}(\mathtt{m},\mathtt{C})$ and $\mathit{anon}(\mathtt{e},\mathtt{C}\langle\overline{\mathtt{T}}\rangle)$ is equivalent to $\mathit{anon}(\mathtt{e},\mathtt{C})$.

6.3.1 Typing rules

Figure 17 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 some additional constraints to prevent unsafe reference widening.

- Rules GT-Var, GT-New, and GT-UCast are similar to those in ConfinedFJ.
- By Rule GT-Field, an expression $\mathtt{e.f_i}$ is well-typed given the environments Δ, Γ only if $\mathtt{f_i}$ is a field declared in the type $\mathit{bound}_\Delta(\mathtt{T})$, where $\mathtt{T}$ is the type of $\mathtt{e}$ and $\mathit{bound}_\Delta(\mathtt{T})$ refers to the type upper bound of $\mathtt{T}$ in Δ if it is a variable or $\mathtt{T}$ itself if it is a non-variable type.
- By Rule (GT-method), if a method call $\mathtt{e.m}(\overline{\mathtt{e}})$ is well-typed, then the types of the arguments $\overline{\mathtt{e}}$ are safe subtypes of the corresponding parameter types of the

Subtyping:

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

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

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

Well-formed types:

$$\frac{}{\Delta \vdash \mathtt{l.Object}} \qquad \frac{\mathtt{X} \in \mathrm{dom}(\Delta)}{\Delta \vdash \mathtt{X}}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{[conf]\ class\ C}\langle \overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle \lhd \mathtt{N}\,\{\,\dots\,\} \\ \Delta \vdash \overline{\mathtt{T}} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}} \end{array}}{\Delta \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle}$$

Type variables in type:

$$Var(\mathtt{X}) = \{\mathtt{X}\} \qquad Var(\mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \bigcup_{\forall \mathtt{T} \in \overline{\mathtt{T}}} Var(\mathtt{T})$$

Confining packages:

$$confPack(\mathtt{X}) = \emptyset$$

$$confPack(\mathtt{C}) = \begin{cases} \{packof(\mathtt{C})\} & \text{if } conf(\mathtt{C}) \\ \emptyset & \text{otherwise} \end{cases}$$

$$confPack(\mathtt{C}\langle\overline{\mathtt{T}}\rangle) = confPack(\mathtt{C}) \cup \textstyle\bigcup_{\forall \mathtt{T} \in \overline{\mathtt{T}}} confPack(\mathtt{T})$$

Safe subtyping:

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \quad confPack(\mathtt{S}) \subseteq confPack(\mathtt{T}) \quad Var(\mathtt{S}) = Var(\mathtt{T})}{\Delta \vdash \mathtt{S} \preceq \mathtt{T}}$$

Visibility of types:

$$\frac{confPack(\mathtt{T}) \subseteq \{packof(\mathtt{C})\} \cup \bigcup_{\forall \mathtt{T} \in \overline{\mathtt{T}}} confPack(\mathtt{T})}{visible(\mathtt{T}, \mathtt{C}\langle\overline{\mathtt{T}}\rangle)}$$

Fig. 15. ConfinedFGJ: Subtyping rules, well-formed types, and miscellaneous definitions.

method `m` in order to prevent unsafe reference widening through parameter passing; and moreover, either `m` is defined in a type `N` so that the type of `e` is a safe subtype of `N` or `m` is an anonymous method. The latter requirement, which

Bound of type:

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

Field look-up:

$$\frac{}{fields(\mathtt{l.Object}) = ()}$$

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

Method Definition Lookup:

$$\frac{CT(\mathtt{C}) = [\mathtt{conf}]\ \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}}{methods(\mathtt{C}) = \overline{\mathtt{M}}}$$

$$\frac{[\mathtt{anon}]\ \mathtt{U}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} \in methods(\mathtt{C})}{mdef(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \mathtt{C}\langle\overline{\mathtt{T}}\rangle}$$

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

Method Type Lookup:

$$\frac{\begin{array}{c} mdef(\mathtt{m},\ \mathtt{N}) = \mathtt{C}\langle\overline{\mathtt{T}}\rangle \quad CT(\mathtt{C}) = [\mathtt{conf}]\ \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle\ldots \\ [\mathtt{anon}]\ \mathtt{U}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} \in methods(\mathtt{C}) \end{array}}{mtype(\mathtt{m},\ \mathtt{N}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}} \to [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}}$$

Method body look-up:

$$\frac{\begin{array}{c} mdef(\mathtt{m},\ \mathtt{N}) = \mathtt{C}\langle\overline{\mathtt{T}}\rangle \quad CT(\mathtt{C}) = [\mathtt{conf}]\ \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\rangle\ldots \\ [\mathtt{anon}]\ \mathtt{U}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} \in methods(\mathtt{C}) \end{array}}{mbody(\mathtt{m},\ \mathtt{N}) = (\overline{\mathtt{x}},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e})}$$

Valid method overriding:

$$\frac{\begin{array}{c} \text{either } \mathtt{m} \text{ is not defined in } \mathtt{N}_0' \text{ or any of its parents, or} \\ mtype(\mathtt{m},\ \mathtt{N}_0) = \overline{\mathtt{T}} \to \mathtt{T} \quad mtype(\mathtt{m},\ \mathtt{N}_0') = \overline{\mathtt{T}} \to \mathtt{T} \quad anon(\mathtt{m}, \mathtt{N}_0') \Rightarrow anon(\mathtt{m}, \mathtt{N}_0) \end{array}}{override(\mathtt{m}, \mathtt{N}_0, \mathtt{N}_0')}$$

Fig. 16. ConfinedFGJ: Auxiliary functions.

Expression typing:

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

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \mathit{fields}(\mathit{bound}_{\Delta}(\mathtt{T})) = (\overline{\mathtt{T}}\ \overline{\mathtt{f}})}{\Delta;\Gamma \vdash \mathtt{e.f_i} : \mathtt{T_i}} \qquad \text{(GT-Field)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}} \\ \mathit{mdef}(\mathtt{m},\ \mathit{bound}_{\Delta}(\mathtt{T})) = \mathtt{N} \quad \mathit{mtype}(\mathtt{m},\ \mathtt{N}) = \overline{\mathtt{U}} \to \mathtt{U} \\ \Delta \vdash \overline{\mathtt{V}} \preceq \overline{\mathtt{U}} \quad (\Delta \vdash \mathtt{T} \preceq \mathtt{N}) \vee \mathit{anon}(\mathtt{m},\mathtt{N})\end{array}}{\Delta;\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{U}} \qquad \text{(GT-Invk)}$$

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

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \Delta \vdash \mathtt{N} \quad \mathtt{T} \preceq \mathtt{N}}{\Delta;\Gamma \vdash (\mathtt{N})\ \mathtt{e} : \mathtt{N}} \qquad \text{(GT-UCast)}$$

Method typing:

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

Class typing:

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

Expression visibility:

$$\frac{\mathit{visible}(\Gamma(\mathtt{x}), \mathtt{N})}{\Delta;\Gamma \vdash \mathit{visible}(\mathtt{x}, \mathtt{N})} \qquad \frac{\Delta;\Gamma \vdash \mathtt{e.f_i} : \mathtt{N'} \quad \mathit{visible}(\mathtt{N'}, \mathtt{N}) \quad \Delta;\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{N})}{\Delta;\Gamma \vdash \mathit{visible}(\mathtt{e.f_i}, \mathtt{N})}$$

$$\frac{\mathit{visible}(\mathtt{N'}, \mathtt{N}) \quad \Delta;\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{N})}{\Delta;\Gamma \vdash \mathit{visible}((\mathtt{N'})\ \mathtt{e}, \mathtt{N})} \qquad \frac{\mathit{visible}(\mathtt{N'}, \mathtt{N}) \quad \forall \mathtt{i},\ \Delta;\Gamma \vdash \mathit{visible}(\mathtt{e_i}, \mathtt{N})}{\Delta;\Gamma \vdash \mathit{visible}(\mathtt{new\ N'}(\overline{\mathtt{e}}), \mathtt{N})}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{N'} \quad \mathit{visible}(\mathtt{N'}, \mathtt{N}) \quad \Delta;\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{N}) \quad \forall \mathtt{i},\ \Delta;\Gamma \vdash \mathit{visible}(\mathtt{e_i}, \mathtt{N})}{\Delta;\Gamma \vdash \mathit{visible}(\mathtt{e.m}(\overline{\mathtt{e}}), \mathtt{N})}$$

Fig. 17. ConfinedFGJ: Typing rules.

corresponds to Rules $\mathscr{C}3$ and $\mathscr{C}8$, prevents a receiver object of confined type from being stored in fields or variables of non-confined types. The defining type of `m` is determined by searching the inheritance hierarchy upward from the type $bound_\Delta(\mathtt{T})$, where `T` is the type of `e`.

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 classes. As in the case of ConfinedFJ, the visibility constraints on method bodies model confinement rules $\mathscr{C}2$ and $\mathscr{C}1$. That is, a confined type cannot be used in the classes outside the package of the confined type, and fields of confined types and methods that return values of confined types are not accessible outside the defining packages of the confined types. The class typing rule GT-Class is similar to the one in ConfinedFJ and it models Rule $\mathscr{C}4$ so that if a class `C` extends a confined type `N`, then `C` must be confined as well.

6.4 Properties

In this section, we prove some results similar to those for ConfinedFJ. In Confined FGJ, a program without free type variables should have the same confinement properties as a program in ConfinedFJ. Theorem 3 shows that the execution of a well-typed generic program always preserves confinement.

The definition of well-typed programs states that all frames be well-typed under the assumption that return values have the proper type. Confinement satisfaction is defined to mean that every the expression being evaluated must be visible from the enclosing method's class.

Definition 3 (*Well-typed*)
A program P is well-typed iff $\vdash P$ as defined below.

$$\frac{\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{N}}{\vdash \mathit{nil}\ .\ \mathtt{v\ m\ e}} \qquad \frac{\vdash P\ .\ \mathtt{v\ m}\ E[\mathtt{e}] \quad \emptyset;\emptyset \vdash \mathtt{e} : \mathtt{N} \quad \emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{N}' \quad \mathtt{N}' \preceq \mathtt{N}}{\vdash P\ .\ \mathtt{v\ m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{e}'}$$

Definition 4 (*Confinement Satisfaction*)
A program $P = \mathtt{v}_1\ \mathtt{m}_1\ \mathtt{e}_1 \ldots \mathtt{v}_n\ \mathtt{m}_n\ \mathtt{e}_n$ satisfies confinement iff for all $\mathtt{i} \in [1,\mathtt{n}]$, we have $visible_{\mathtt{N}_i}(\mathtt{e}_i, \mathtt{N}'_i)$, where $\mathtt{v}_i = \mathtt{new}\ \mathtt{N}_i(\overline{\mathtt{u}})$ and $mdef(\mathtt{m}_i,\ \mathtt{N}_i) = \mathtt{N}'_i$.

The definition of $visible_{\mathtt{N}_0}(\mathtt{e},\mathtt{N})$ is similar to that of $visible_{\mathtt{C}_0}(\mathtt{e},\mathtt{C})$. The difference is that if the type of `e` is $\mathtt{N}'$, then $visible_{\mathtt{N}_0}(\mathtt{e},\mathtt{N})$ implies $visible(\mathtt{N}',\mathtt{N})$. The latter means that if $\mathtt{N}'$ is a type confined in the package `p`, and $\mathtt{N} = \mathtt{C}\langle\overline{\mathtt{N}}\rangle$, then either `C` is defined in `P` or there exists $\mathtt{N}'' \in \overline{\mathtt{N}}$ such that $\mathtt{N}''$ is confined in `p`.

Next, we prove some helper lemmas used in proving that subject reduction preserves typing (Lemma 18) and confinement (Lemma 20).

In particular, the proof of Lemma 18 depends on Lemma 15, which shows that the body of a method invoked on a well-formed type is well-typed. To prove Lemma 15, we need to show that type variable substitution preserves safe subtyping, well-formed types, and expression typing, Also, the proof of Lemma 20 depends on Lemma 14, which shows that type variable substitution preserves static expression visibility. To

$$\frac{\emptyset;\emptyset \vdash \mathtt{e.f_i} : \mathtt{N'} \quad \mathit{visible}(\mathtt{N'},\mathtt{N}) \quad (\mathtt{e} = \mathtt{new}\ \mathtt{N_0}(\overline{\mathtt{u}}) \ \vee\ \mathit{visible}_{\mathtt{N_0}}(\mathtt{e},\mathtt{N}))}{\mathit{visible}_{\mathtt{N_0}}(\mathtt{e.f_i},\mathtt{N})}$$

$$\frac{\mathit{visible}(\mathtt{N'},\mathtt{N}) \quad \mathit{visible}_{\mathtt{N_0}}(\mathtt{e},\mathtt{N})}{\mathit{visible}_{\mathtt{N_0}}((\mathtt{N'})\ \mathtt{e},\mathtt{N})} \qquad \frac{\mathit{visible}(\mathtt{N'},\mathtt{N}) \quad \forall \mathtt{i},\ \mathit{visible}_{\mathtt{N_0}}(\mathtt{e_i},\mathtt{N})}{\mathit{visible}_{\mathtt{N_0}}(\mathtt{new}\ \mathtt{N'}(\overline{\mathtt{v}}\ \overline{\mathtt{e}}),\mathtt{N})}$$

$$\frac{\emptyset;\emptyset \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{N'} \quad \mathit{visible}(\mathtt{N'},\mathtt{N}) \quad (\mathtt{e} = \mathtt{new}\ \mathtt{N_0}(\overline{\mathtt{u}}) \ \vee\ \mathit{visible}_{\mathtt{N_0}}(\mathtt{e},\mathtt{N})) \quad \forall \mathtt{i},\ \mathit{visible}_{\mathtt{N_0}}(\mathtt{e_i},\mathtt{N})}{\mathit{visible}_{\mathtt{N_0}}(\mathtt{e.m}(\overline{\mathtt{e}}),\mathtt{N})}$$

T1

Fig. 18. ConfinedFGJ: Runtime expression visibility.

prove Lemma 14, we need to show that type variable substitution preserves type visibility and expression typing.

Suppose $CT(\mathtt{C}) = \ldots \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N'}}\rangle \ldots$, $\emptyset \vdash \mathtt{C}\langle\overline{\mathtt{N}}\rangle$, and $\Delta = \overline{\mathtt{X}} <: \overline{\mathtt{N'}}$. We show that the substitution of type variables $\overline{\mathtt{X}}$ by $\overline{\mathtt{N}}$ preserves subtyping.

Lemma 9
If $\Delta \vdash \mathtt{S} <: \mathtt{T}$, then $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{S} <: [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}$.

Proof
The proof follows that of FGJ (Igarashi *et al.*, 2001). □

The lemma below is related to Rule $\mathscr{C}7$, which imposes additional restriction on safe subtyping for generic types so that safe-subtyping still holds when there is type-variable substitution.

Suppose $CT(\mathtt{C}) = \ldots \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N'}}\rangle \ldots$, $\emptyset \vdash \mathtt{C}\langle\overline{\mathtt{N}}\rangle$, and $\Delta = \overline{\mathtt{X}} <: \overline{\mathtt{N'}}$. We show that the substitution of type variables $\overline{\mathtt{X}}$ by $\overline{\mathtt{N}}$ preserves safe subtyping.

Lemma 10
If $\Delta \vdash \mathtt{S},\mathtt{T}$ and $\Delta \vdash \mathtt{S} \preceq \mathtt{T}$, then $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{S} \preceq [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}$.

Proof
From $\Delta \vdash \mathtt{S} \preceq \mathtt{T}$, we have $\Delta \vdash \mathtt{S} <: \mathtt{T}$, $\mathit{Var}(\mathtt{S}) = \mathit{Var}(\mathtt{T})$, and $\mathit{confPack}(\mathtt{S}) \subseteq \mathit{confPack}(\mathtt{T})$. By Lemma 9, we have $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{S} <: [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}$. From $\Delta \vdash \mathtt{S},\mathtt{T}$ and $\mathrm{dom}(\Delta) = \overline{\mathtt{X}}$, all type variables in $\mathtt{S},\mathtt{T}$ are replaced by types in the substitution $[\overline{\mathtt{N}}/\overline{\mathtt{X}}]$, which implies $\mathit{Var}([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{S}) = \mathit{Var}([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}) = \emptyset$. Since $\mathit{Var}(\mathtt{S}) = \mathit{Var}(\mathtt{T})$, the same set of types replace variables in $\mathtt{S}$ and $\mathtt{T}$. Thus, $\mathit{confPack}([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{S}) \subseteq \mathit{confPack}([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T})$. □

Suppose $CT(\mathtt{C}) = \ldots \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N'}}\rangle \ldots$, $\emptyset \vdash \mathtt{C}\langle\overline{\mathtt{N}}\rangle$, and $\Delta = \overline{\mathtt{X}} <: \overline{\mathtt{N'}}$. We show that the substitution of type variables $\overline{\mathtt{X}}$ by $\overline{\mathtt{N}}$ preserves well-formed types.

Lemma 11
If $\Delta \vdash \mathtt{T}$, then $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}$.

Proof
The proof follows that of FGJ (Igarashi *et al.*, 2001). □

Suppose $CT(\mathtt{C}) = \ldots \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N'}}\rangle \ldots$, $\emptyset \vdash \mathtt{C}\langle\overline{\mathtt{N}}\rangle$, and $\Delta = \overline{\mathtt{X}} <: \overline{\mathtt{N'}}$. We show that the substitution of type variables $\overline{\mathtt{X}}$ by $\overline{\mathtt{N}}$ preserves expression typing.

Lemma 12
If $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$ then $\emptyset;\ [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{e} : [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}$.

Proof
We prove by induction on the derivation of $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$. There are five cases depending on the last rule used in the type derivation.

1. Suppose $\mathtt{e} = \mathtt{x}$. In this case, $\Delta;\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})$ and $\emptyset;[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma \vdash \mathtt{x} : [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma(\mathtt{x})$.
2. Suppose $\mathtt{e} = \mathtt{e}_0.\mathtt{f}_\mathtt{i}$. In this case, if $\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0$ and *fields*(*bound*$_\Delta(\mathtt{T}_0)) = (\overline{\mathtt{T}\ \mathtt{f}})$, then $\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_\mathtt{i} : \mathtt{T}_\mathtt{i}$. By induction hypothesis, we have that $\emptyset;[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{e}_0 : [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_0$; and by the definition of *fields*, $\exists\overline{\mathtt{S}\ g}$ such that *fields* $([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_0) = (\overline{\mathtt{S}\ g},\ [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\overline{\mathtt{T}\ f})$. Therefore, by Rule GT-Field, we have $\emptyset;[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}](\mathtt{e}_0.\mathtt{f}_\mathtt{i}) : [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_\mathtt{i}$.
3. Suppose $\mathtt{e} = \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$, In this case, $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{U}$ if $\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0$, $\Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{V}}$, *mdef*$(\mathtt{m},$ *bound*$_\Delta(\mathtt{T}_0)) = \mathtt{N}_0$, *mtype*$(\mathtt{m},\ \mathtt{N}_0) = \overline{\mathtt{U}} \to \mathtt{U}$, $\Delta \vdash \overline{\mathtt{V}} \preceq \overline{\mathtt{U}}$, and $\Delta \vdash \mathtt{T}_0 \preceq \mathtt{N}_0 \ \vee$ *anon*$(\mathtt{m},\mathtt{N}_0)$. By induction hypothesis, we have $\emptyset;[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{e}_0 : [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_0$, $\emptyset;[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\overline{\mathtt{e}} : [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}$, and by Lemma 10, we have $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\overline{\mathtt{V}} \preceq [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}$. From $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_0 <: [\overline{\mathtt{N}}/\overline{\mathtt{X}}]($*bound*$_\Delta(\mathtt{T}_0))$, and by induction on the recursive definition of *mdef*, it can be shown that *mdef*$(\mathtt{m},\ [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_0) = \mathtt{N}'_0$, where $\emptyset \vdash \mathtt{N}'_0 <: [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0$, and *mtype*$(\mathtt{m},\ [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0) = [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\overline{\mathtt{U}} \to [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{U}$. In particular, if $\mathtt{T}_0$ is a non-variable type, then $\mathtt{N}'_0 = [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0$ (from the definition of *mdef*). If $\mathtt{T}_0$ is a variable, then $[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_0 <: \Delta(\mathtt{T}_0)$ and by the definition of *mdef*, $\mathtt{m}$ has to be defined in $[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0$ or a subclass of $[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0$. If $\Delta \vdash \mathtt{T}_0 \preceq \mathtt{N}_0$, then from Lemma 10, we have $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_0 \preceq [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0$. Since *Var*$([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_0) = \emptyset$, we have *confPack*$([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_0) \subseteq$ *confPack*$([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0)$. From $\emptyset \vdash \mathtt{N}'_0 <: [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0$ and Rule GT-Class, we have *confPack*$(\mathtt{N}'_0) \subseteq$ *confPack*$([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0)$. Thus, $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_0 \preceq \mathtt{N}'_0$. By definition, *anon*$(\mathtt{m},\mathtt{N}_0)$ implies *anon*$(\mathtt{m},[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0)$. By Rule GT-Method and $\emptyset \vdash \mathtt{N}'_0 <: [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0$, we have that *anon*$(\mathtt{m},[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}_0)$ implies *anon*$(\mathtt{m},\mathtt{N}'_0)$. Thus, we have $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}_0 \preceq \mathtt{N}'_0 \ \vee$ *anon*$(\mathtt{m},\mathtt{N}'_0)$. From Rule GT-Invk, we conclude that $\emptyset;[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}](\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})) : [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{U}$.
4. Suppose $\mathtt{e} = \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}})$. In this case, if $\Delta \vdash \mathtt{N}$, *fields*$(\mathtt{N}) = (\overline{\mathtt{T}\ \mathtt{f}})$, $\Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}$, and $\Delta \vdash \overline{\mathtt{S}} \preceq \overline{\mathtt{T}}$, then $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{N}$. By Lemma 11, we have $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}$. It can be shown that *fields*$([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}) = ([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\overline{\mathtt{T}\ \mathtt{f}})$. By induction hypothesis, we have $\emptyset;[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\overline{\mathtt{e}} : [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}$. By Lemma 10, we have $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\overline{\mathtt{S}} \preceq [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}$. Thus, by Rule GT-New, we have $\emptyset;[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}](\mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}})) : [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}$.
5. Suppose $\mathtt{e} = (\mathtt{N})\ \mathtt{e}'$. In this case, $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{N}$, if $\Delta;\Gamma \vdash \mathtt{e}' : \mathtt{T}$, $\Delta \vdash \mathtt{N}$, and $\Delta \vdash \mathtt{T} \preceq \mathtt{N}$. By Lemma 11, we have $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}$. By induction hypothesis, we have $\emptyset;[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{e} : [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T}$. By Lemma 10, we have $\emptyset \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T} \preceq [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}$. Therefore, by Rule GT-Cast, we have $\emptyset;[\overline{\mathtt{N}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{N}}/\overline{\mathtt{X}}]((\mathtt{N})\ \mathtt{e}') : [\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{N}$.

□

Suppose $CT(\mathtt{C}) = \ldots \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}} \lhd \overline{\mathtt{N}'}\rangle \ldots$, $\emptyset \vdash \mathtt{C}\langle\overline{\mathtt{N}}\rangle$, and $\Delta = \overline{\mathtt{X}} <: \overline{\mathtt{N}'}$. We show that the substitution of type variables $\overline{\mathtt{X}}$ by $\overline{\mathtt{N}}$ preserves type visibility.

Lemma 13
If $\Delta \vdash \mathtt{T}$, *visible*$(\mathtt{T},\mathtt{C}\langle\overline{\mathtt{X}}\rangle)$ then *visible*$([\overline{\mathtt{N}}/\overline{\mathtt{X}}]\mathtt{T},\mathtt{C}\langle\overline{\mathtt{N}}\rangle)$.

Proof
Since $\mathit{confPack}(\mathtt{T}) \subseteq \{\mathit{packof}(\mathtt{C})\}$, we have $\mathit{confPack}([^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{T}) \subseteq \mathit{confPack}(\mathtt{C}\langle\overline{\mathtt{N}}\rangle)$. Thus, $\mathit{visible}([^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{T}, \mathtt{C}\langle\overline{\mathtt{N}}\rangle)$. □

Suppose $CT(\mathtt{C}) = \ldots \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N'}}\rangle \ldots$, $\emptyset \vdash \mathtt{C}\langle\overline{\mathtt{N}}\rangle$, and $\Delta = \overline{\mathtt{X}} <: \overline{\mathtt{N'}}$. We show that the substitution of type variables $\overline{\mathtt{X}}$ by $\overline{\mathtt{N}}$ preserves static expression visibility.

Lemma 14
If $\Delta;\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}\langle\overline{\mathtt{X}}\rangle)$, then $\emptyset;\Gamma \vdash \mathit{visible}([^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{e}, \mathtt{C}\langle\overline{\mathtt{N}}\rangle)$.

Proof
The proof is straightforward by induction on the derivation of $\Delta;\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})$ and by Lemma 12 and 13. □

We now show that the return expression of a well-typed method is well-typed after type variable substitution.

Lemma 15
If $\emptyset \vdash \mathtt{N_0}$, $\mathit{mtype}(\mathtt{m},\ \mathtt{N_0}) = \overline{\mathtt{N'}} \rightarrow \mathtt{N'}$, $\mathit{mbody}(\mathtt{m},\ \mathtt{N_0}) = (\overline{\mathtt{x}},\ \mathtt{e})$, and $\mathit{mdef}(\mathtt{m},\ \mathtt{N_0}) = \mathtt{N'_0}$, then $\exists\mathtt{N}$ such that $\emptyset;\ \overline{\mathtt{x}} : \overline{\mathtt{N'}}, \mathtt{this} : \mathtt{N'_0} \vdash \mathtt{e} : \mathtt{N}$ and $\emptyset \vdash \mathtt{N} \preceq \mathtt{N'}$.

Proof
Since $\emptyset \vdash \mathtt{N_0}$, from the definition of $\mathit{mdef}(\mathtt{m},\ \mathtt{N_0}) = \mathtt{N'_0}$, Rule T-CLASS, and Lemma 11, we can show by induction $\emptyset \vdash \mathtt{N'_0}$.

If $\mathtt{N'_0} = \mathtt{C}\langle\overline{\mathtt{N}}\rangle$, $CT(\mathtt{C}) = \ldots \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}} \triangleleft \overline{\mathtt{W}}\rangle \ldots$, $\mathit{mtype}(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{X}}\rangle) = \overline{\mathtt{U}} \rightarrow \mathtt{U}$, and $\mathit{mbody}(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{X}}\rangle) = (\overline{\mathtt{x}},\ \mathtt{e_0})$, then there exists $\mathtt{U'}$ such that $\Delta;\ \overline{\mathtt{x}} : \overline{\mathtt{U}}, \mathtt{this} : \mathtt{C}\langle\overline{\mathtt{X}}\rangle \vdash \mathtt{e_0} : \mathtt{U'}$ and $\Delta \vdash \mathtt{U'} \preceq \mathtt{U}$, where $\Delta = \overline{\mathtt{X}} <: \overline{\mathtt{W}}$.

By Lemma 12 and $\emptyset \vdash \mathtt{N'_0}$, we have $\emptyset;\ \overline{\mathtt{x}} : [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\overline{\mathtt{U}}, \mathtt{this} : [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{C}\langle\overline{\mathtt{X}}\rangle \vdash [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{e_0} : [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{U'}$.

By Lemma 10 and $\emptyset \vdash \mathtt{N'_0}$, we have $\emptyset \vdash [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{U'} \preceq [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{U}$.

Since $\mathit{mtype}(\mathtt{m},\ \mathtt{N_0}) = \overline{\mathtt{N'}} \rightarrow \mathtt{N'}$, we have $\overline{\mathtt{N'}} = [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\overline{\mathtt{U}}$, $\mathtt{N'} = [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{U}$. Also, since $\mathit{mbody}(\mathtt{m},\ \mathtt{N_0}) = (\overline{\mathtt{x}},\ \mathtt{e})$ we have $\mathtt{e} = [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{e_0}$. Since $\mathtt{N'_0} = \mathtt{C}\langle\overline{\mathtt{N}}\rangle = [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{C}\langle\overline{\mathtt{X}}\rangle$, we have $\emptyset;\ \overline{\mathtt{x}} : \overline{\mathtt{N'}}, \mathtt{this} : \mathtt{N'_0} \vdash \mathtt{e} : [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{U'}$ and $\emptyset \vdash [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{U'} \preceq \mathtt{N'}$. Let $\mathtt{N} = [^{\overline{\mathtt{N}}}/_{\overline{\mathtt{X}}}]\mathtt{U'}$. Then, we have $\emptyset;\ \overline{\mathtt{x}} : \overline{\mathtt{N'}}, \mathtt{this} : \mathtt{N'_0} \vdash \mathtt{e} : \mathtt{N}$ and $\emptyset \vdash \mathtt{N} \preceq \mathtt{N'}$. □

Lemmas 16 and 17 show that term substitution preserves typing and the latter applies to the bodies of anonymous methods.

Lemma 16 (*Term Substitution*)
If $\emptyset;\ \overline{\mathtt{x}} : \overline{\mathtt{N'}} \vdash \mathtt{e} : \mathtt{N'}$, $\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{N}}$, and $\emptyset \vdash \overline{\mathtt{N}} \preceq \overline{\mathtt{N'}}$, then $\emptyset \vdash [^{\overline{\mathtt{v}}}/_{\overline{\mathtt{x}}}]\mathtt{e} : \mathtt{N}$ where $\emptyset \vdash \mathtt{N} \preceq \mathtt{N'}$.

Proof
The proof is similar to that of Lemma 3. □

Lemma 17 (*Substitution*)
If $\emptyset;\ \overline{\mathtt{x}} : \overline{\mathtt{N'}}, \mathtt{this} : \mathtt{N'_0} \vdash \mathtt{e} : \mathtt{N'}$, $\emptyset;\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{N}}$, $\emptyset \vdash \overline{\mathtt{N}} \preceq \overline{\mathtt{N'}}$, $\emptyset;\emptyset \vdash \mathtt{new}\ \mathtt{N_0}(\overline{\mathtt{u}}) : \mathtt{N_0}$, $\emptyset \vdash \mathtt{N_0} <: \mathtt{N'_0}$, and $\mathit{anon}(\mathtt{e}, \mathtt{N'_0})$, then $\emptyset;\emptyset \vdash [^{\overline{\mathtt{v}}}/_{\overline{\mathtt{x}}}, ^{\mathtt{new}\ \mathtt{N_0}(\overline{\mathtt{u}})}/_{\mathtt{this}}]\mathtt{e} : \mathtt{N}$ where $\emptyset \vdash \mathtt{N} \preceq \mathtt{N'}$.

Proof
The proof is similar to that of Lemma 4. □

We now show that subject reduction preserves typing and well-typed program can make progress.

Lemma 18 (*Subject reduction*)
If P is well-typed and $P \rightarrow P'$, then P' is well-typed.

Proof
Similar to Lemma 5, we prove by a case analysis of the reduction rule used.

If $P' = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{e}''$, then to prove P' is well-typed, we need to show that $P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]$ is well-typed, $\mathtt{e}''$ is well-typed and its type is a safe subtype of the type of $\mathtt{e}$. In particular, if $P = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{e}'$ then it is sufficient to show that $\emptyset;\emptyset \vdash \mathtt{e}'' : \mathtt{N}''$ and $\emptyset \vdash \mathtt{N}'' \preceq \mathtt{N}'$, where $\mathtt{N}'$ is the type of $\mathtt{e}'$.

1. If the reduction from P to P' is by Rule GR-Field, then P has the form $P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]$, where $\mathtt{e} = \mathtt{new}\ \mathtt{N}_0(\overline{\mathtt{v}}).\mathtt{f}_\mathtt{i}$, and $P' = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}']$, where $\mathtt{e}' = \mathtt{v}_\mathtt{i}$. Since P is well-typed, if $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{N}_\mathtt{i}$, then from Rule GT-Field, $\mathtt{new}\ \mathtt{N}_0(\overline{\mathtt{v}})$ is well-typed and if $\emptyset;\emptyset \vdash \mathtt{v}_\mathtt{i} : \mathtt{C}'_\mathtt{i}$, then $\mathtt{N}'_\mathtt{i} \preceq \mathtt{N}_\mathtt{i}$ by Rule GT-New. By induction on the type derivation of $E[\mathtt{e}]$, we can show that if $\emptyset;\emptyset \vdash E[\mathtt{e}] : \mathtt{N}$, then $\exists \mathtt{N}'$ such that $\emptyset;\emptyset \vdash E[\mathtt{e}'] : \mathtt{N}'$ and $\emptyset \vdash \mathtt{N}' \preceq \mathtt{N}$. Therefore, P' is well-typed.
2. If the reduction is by Rule GR-Cast, then P has the form $P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]$, where $\mathtt{e} = (\mathtt{N})\ \mathtt{new}\ \mathtt{N}'(\overline{\mathtt{v}})$, and $P' = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}']$, where $\mathtt{e}' = \mathtt{new}\ \mathtt{N}'(\overline{\mathtt{u}})$, and from Rule GT-UCast, $\emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{N}'$ and $\emptyset \vdash \mathtt{N}' \preceq \mathtt{N}$. Thus, similar to the previous case we can show that P' is well-typed.
3. If the reduction is by Rule R-Invk, then P has the form $P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]$, where $\mathtt{e} = \mathtt{v}'.\mathtt{m}'(\overline{\mathtt{v}'})$, $\mathtt{v}' = \mathtt{new}\ \mathtt{N}_0(\overline{\mathtt{u}})$, $\mathit{mbody}(\mathtt{m},\ \mathtt{N}_0) = (\overline{\mathtt{x}},\ \mathtt{e}_0)$, and $P' = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{e}'$, where $\mathtt{e}' = [\overline{\mathtt{v}}/_{\overline{\mathtt{x}}},\ ^{\mathtt{v}'}/_{\mathtt{this}}]\mathtt{e}_0$. If $\mathit{mtype}(\mathtt{m},\ \mathtt{N}_0) = \overline{\mathtt{N}} \rightarrow \mathtt{N}$, $\mathit{mdef}(\mathtt{m},\ \mathtt{N}_0) = \mathtt{N}'_0$, $\emptyset;\overline{\mathtt{x}} : \overline{\mathtt{N}}, \mathtt{this} : \mathtt{N}'_0 \vdash \mathtt{e}_0 : \mathtt{N}'$, and $\emptyset;\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{N}'}$, then $\emptyset \vdash \mathtt{N}' \preceq \mathtt{N}$, $\emptyset \vdash \overline{\mathtt{N}'} \preceq \overline{\mathtt{N}}$, and either $\emptyset \vdash \mathtt{N}_0 \preceq \mathtt{N}'_0$ or $\mathit{anon}(\mathtt{m}, \mathtt{N}_0)$. From Lemma 15, 16, and 17, and Rule R-Invk, $\exists \mathtt{N}''$ such that $\emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{N}''$ and $\emptyset \vdash \mathtt{N}'' \preceq \mathtt{N}'$. Thus, $\emptyset \vdash \mathtt{N}'' \preceq \mathtt{N}$ and P' is well-typed.
4. If the reduction is by Rule GR-Ret, then P is of the form $P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{v}''$ and $P' = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{v}'']$. Since P is well-typed, if $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{W}$ and $\emptyset \vdash \mathtt{v}'' : \mathtt{W}'$, then $\emptyset \vdash \mathtt{W}' \preceq \mathtt{W}$. We can show by induction that if $\emptyset;\emptyset \vdash E[\mathtt{e}] : \mathtt{N}$, then $\emptyset \vdash E[\mathtt{v}''] : \mathtt{N}'$ and $\emptyset \vdash \mathtt{N}' \preceq \mathtt{N}$. Therefore, P' is well-typed.

□

Lemma 19 (*Progress*)
If P is well-typed and not in the form of $\mathit{nil}\ .\ \mathtt{v}\ \mathtt{m}\ \mathtt{v}'$ then $\exists P'$ such that $P \rightarrow P'$.

6.4.1 Confinement property

The following lemma shows that subject reduction of a well-typed program preserves confinement.

Lemma 20
If P is well-typed, satisfies confinement, and $P \rightarrow P'$, then P' satisfies confinement.

Proof
If the reduction from P to P' is by Rule GR-Field, GR-Cast, or GR-Ret, the proof is similar to the that of Lemma 8.

If the reduction is by Rule GR-Invk, $P = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]$, $\mathtt{e} = \mathtt{v}'.\mathtt{m}'(\overline{\mathtt{v}})$, and $\mathtt{v}' = \mathtt{new}\ \mathtt{N}_0(\overline{\mathtt{u}})$. $mbody(\mathtt{m}', \mathtt{N}_0) = (\overline{\mathtt{x}}, \mathtt{e}_0)$, then $P' = P''\ .\ \mathtt{v}\ \mathtt{m}\ E[\mathtt{e}]\ .\ \mathtt{v}'\ \mathtt{m}'\ \mathtt{e}'$, where $\mathtt{e}' = [\overline{\mathtt{v}}/\overline{\mathtt{x}}, \mathtt{v}'/\mathtt{this}]\mathtt{e}_0$.

Suppose $mtype(\mathtt{m}', \mathtt{N}_0) = \overline{\mathtt{N}} \rightarrow \mathtt{N}$, $mdef(\mathtt{m}', \mathtt{N}_0) = \mathtt{N}'_0$, and $\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{N}'}$. From Rule GT-Method and Lemma 14, we have $\emptyset;\Gamma \vdash visible(\mathtt{e}_0, \mathtt{N}'_0)$ and from Lemma 15, we have $\emptyset;\Gamma \vdash \mathtt{e}_0 : \mathtt{N}'$ where $\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{N}}, \mathtt{this} : \mathtt{N}'_0$ and $\emptyset \vdash \mathtt{N}' \preceq \mathtt{N}$. Since P is well-typed, by Rule GT-Invk we have $\emptyset \vdash \overline{\mathtt{N}'} \preceq \overline{\mathtt{N}}$.

If $\emptyset \vdash \mathtt{N}_0 \preceq \mathtt{N}'_0$, then from Lemma 16, we have that for each subterm $\mathtt{e}'_0$ of $\mathtt{e}_0$, if $\emptyset;\Gamma \vdash \mathtt{e}'_0 : \mathtt{W}$, then $\emptyset;\emptyset \vdash [\overline{\mathtt{v}}/\overline{\mathtt{x}}, \mathtt{v}'/\mathtt{this}]\mathtt{e}'_0 : \mathtt{W}'$, $\emptyset \vdash \mathtt{W}' \preceq \mathtt{W}$, and consequently $visible(\mathtt{W}, \mathtt{N}'_0)$ implies $visible(\mathtt{W}', \mathtt{N}'_0)$. Thus, from $\emptyset;\Gamma \vdash visible(\mathtt{e}_0, \mathtt{N}'_0)$, we can show $visible_{\mathtt{N}_0}(\mathtt{e}', \mathtt{N}'_0)$ by induction.

If $\emptyset \vdash \mathtt{N}_0 \not\preceq \mathtt{N}'_0$, then by Rule GT-Invk, $anon(\mathtt{m}', \mathtt{N}_0)$, which means that the variable `this` can occur only in the subterms of $\mathtt{e}_0$ in the form of $\mathtt{this.f}$ or $\mathtt{this.m}'(\overline{\mathtt{e}})$ (where $\mathtt{e_i} \neq \mathtt{this}$, $\forall \mathtt{i}$). Thus, the object $\mathtt{v}'$ can be only in the subterms of $\mathtt{e}'$ in the form of $\mathtt{v}'.\mathtt{f}$ or $\mathtt{v}'.\mathtt{m}'(\overline{\mathtt{e}})$ (where $\mathtt{e_i}$ is not of the form $\mathtt{new}\ \mathtt{N}_0(\overline{\mathtt{u}})$, $\forall \mathtt{i}$). From Lemma 17 and $\emptyset;\Gamma \vdash visible(\mathtt{e}_0, \mathtt{N}'_0)$, we can prove $visible_{\mathtt{N}_0}(\mathtt{e}', \mathtt{N}'_0)$ by simple induction. Thus, P' satisfies confinement. □

Lastly, we show that the execution of a well-typed generic program always preserves confinement.

Theorem 3 (*Confinement*)
If $P = \mathtt{v}\ \mathtt{m}\ \mathtt{e}$ is well-typed, satisfies confinement, and $P \rightarrow^* P'$, then P' satisfies confinement.

Proof
Immediate from Lemma 18 and 20. □

6.5 Example: public-key cryptography

We demonstrate the use of generic confined types with an example (Figure 19 and 20) adapted from Vitek and Bokowski (2001). The implementation of a public-key cryptography package needs to ensure that the random number object used in the generation of key pairs cannot be accessed by clients of the package. Also, the references to the private key object generated for a client of the RSA implementation should not escape the client package. The following examples are written in pseudo-ConfinedFGJ using access modifiers for fields, assignment, default initializers, and a type `void`.

In this example, the implementation of public-key cryptography is divided into two parts: a package `rsa` containing reusable classes and a package `secure` containing code for one particular client of the `rsa` package. The class `rsa.ConfinedRandom` is used to hold the random number generator confined in the package `rsa`. The private key object instantiated from the class `secure.PrivKey` is confined in the package `secure`. The class `rsa.Key` implements public-private key pairs. The confined class `PrivKey` extends the class `Key`. Since the methods `crypt()` and `setValues()` in `Key` are anonymous, they can be reused in the confined subclass. `KeyFactory` is a

```
class rsa.Key ◁ l.Object {
   l.BigDecimal mod;
   l.BigDecimal exp;
   anon l.String crypt(l.String msg) {...}
   anon void setValues(l.BigDecimal m, l.BigDecimal e) {
      mod = m; exp = e;
   }
}

conf class rsa.ConfinedRandom ◁ l.Random { }

class rsa.KeyFactory <X ◁ rsa.Key> ◁ l.Object {
   void genKeyPair(rsa.Key pub, X priv) {...}
}
```

Fig. 19. Package containing RSA algorithm.

```
conf class secure.PrivKey ◁ rsa.Key { }

class secure.Main ◁ l.Object {
   private secure.PrivKey privk = new secure.PrivKey();
   rsa.Key pubk = new rsa.Key();
   void main() {
      (new rsa.KeyFactory<secure.PrivKey>()).genKeyPair(pubk, privk);
      ...
   }
}
```

Fig. 20. Confining a type in a different package.

generic class that generates public-private key pairs using a `ConfinedRandom` object. The type parameter `X` with type upper bound `rsa.Key` can be instantiated with type `secure.PrivKey`. Class `secure.Main` calls the `genKeyPair()` method of the object `new rsa.KeyFactory<secure.PrivKey>()` to get a public-private key pair. The `PrivKey` object can be passed to the method `genKeyPair()` because the object `new rsa.KeyFactory<secure.PrivKey>()` is confined in the package `secure` and the argument `privk` now has type `PrivKey`.

In comparison to the original example in Vitek and Bokowski (2001), generic classes allow more reuse and avoid code redundancy. Without generic class, the `KeyFactory` class cannot be used directly in the package `secure` since there is no way for `KeyFactory` to access the private key objects confined in the package `secure`.

The level of object confinement in Figure 19 and 20 can be improved even further. For example, the fields `mod` and `exp` of `rsa.Key` refer to objects of public type which may be accessible to outside code. Even though direct access to the `exp` and `mod` fields of a `PrivKey` object requires a reference to the object, outside code may still obtain references to the values indirectly. For instance, `exp` and `mod` are generated by `rsa.KeyFactory`, which may pass the references to these objects to code outside the package `secure`. Also, the classes in the package `secure` may inadvertently copy the

```
class rsa.Num<X ◁ l.Object> ◁ l.BigDecimal { }

class secure.Key<X ◁ l.Object> ◁ l.Object {
   rsa.Num<X> mod, exp;
   anon void setValues(rsa.Num<X> m, rsa.Num<X> e) {
      this.mod = m; this.exp = e;
   }
}

class rsa.KeyFactory<X ◁ l.Object> ◁ l.Object {
   void genKeyPair(rsa.Key<l.Object> pub, rsa.Key<X> priv) { ...} }


conf class secure.C ◁ l.Object { }

class secure.Main ◁ l.Object {
   private rsa.Key<secure.C> privk = new rsa.Key<secure.C>();
   rsa.Key<l.Object> pubk = new rsa.Key<l.Object>();
   void main() {
      (new rsa.KeyFactory<secure.C>()).genKeyPair(pubk, privk);
      ...
   }
}
```

Fig. 21. Confining the internal values of the private key object.

internal values of `secure.PrivKey` objects to outside code. To solve this problem, we can define a generic class `rsa.Num<X>` to hold `mod` and `exp`. `Key` is redefined as a generic class `rsa.Key<X>` where the type variable `X` is used to instantiate the type of the fields. We also define a dummy confined class `secure.C` solely for the purpose of instantiating generic classes so that their instances are confined within `secure`. The modified code is shown in Figure 21.

Since public key objects can come from anywhere, they are instances of the type `Key<l.Object>`. Private key objects of the `secure` package are instantiated from `Key<secure.C>` class so that they are confined in the package. Correspondingly, `mod` and `exp` of the private key objects are instances of the type `rsa.Num<secure.C>` which are confined within `secure` as well.

Using generic confined types, we can create objects confined in the package `secure` by calling the method `genKeyPair` of the class `KeyFactory` located in the package `rsa`. Thus, both the private key object and its internal values can be confined. This would be otherwise difficult to do since the class `KeyFactory` must be located in `rsa` in order to access a random number object of the type `ConfinedRandom`.

7 Related work

Reference semantics permeate object-oriented programming languages, and the issue of controlling aliasing has been the focus of numerous papers in recent years (Hogg, 1991; Hogg *et al.*, 1992; Kent & Maung, 1995; Detlefs *et al.*, 1998; Almeida, 1997;

Noble *et al.*, 1998; Genius *et al.*, 1998; Clarke *et al.*, 1998; Müller & Poetzsch-Heffter, 1999; Clarke *et al.*, 2001; Aldrich *et al.*, 2002; Banerjee & Naumann, 2002a; Clarke & Drossopoulou, 2002; Boyapati *et al.*, 2003b). Noble *et al.* (1998) proposed flexible alias protection to control potential aliasing amongst components of an aggregate object (or *owner*). Aliasing mode declarations specify constraints on sharing of references. The mode `rep` protects *representation objects* from exposure. In essence, `rep` objects belong to a single owner object and the model guarantees that all paths that lead to a representation object go through that object's owner. The mode `arg` marks argument objects which do not belong to the current owner, and therefore may be aliased from the outside. Argument objects can have different *roles*, and the model guarantees that an owner cannot introduce aliasing between roles. Clarke *et al.* (1998) first proposed ownership types for representation containment and investigated the properties of object graphs based on dominator trees. Their ownership model enforces strict object encapsulation with arguably limited expressiveness. Later the same authors (Clarke *et al.*, 2001) formalized the ownership model with a simple object calculus and fixed ownership context. Clarke and Drossopoulou (2002) extended the ownership model with dynamic aliases to allow temporary access to the representation objects. They also extended the ownership types with computational effects to support reasoning about object-oriented programs.

Hogg's Islands and Almeida's Balloons have similar aims (Hogg, 1991; Almeida, 1997). An Island or Balloon is an owner object that protects its internal representation from aliasing. The main difference from Noble *et al.* (1998) is that both proposals strive for full encapsulation, that is, all objects reachable from an owner are protected from aliasing. Boyland *et al.* (2001) introduced capabilities as a uniform system to describe restrictions imposed on references.

The *universe* types (Müller & Poetzsch-Heffter, 1999) uses read-only types to handle temporary access to the representation objects of an abstraction. Later, Muelleri and Poetzsch-Heffter (2000a; 2000b) extend the *universe* model with an notion of *type universe* such that all objects of the types declared in one module can own a common representation. The objects in a universe are fully contained and to transfer objects between universes requires cloning operations.

Boyapati *et al.* (2003b) use ownership types for object encapsulation and local reasoning about program correctness. They use inner classes to represent interface object that shares the representation of an owner. Each inner class instance is owned by its outer class instance and thus they can be reasoned together as a module. They also applied ownership types to detect race conditions (Boyapati *et al.*, 2002; Boyapati & Rinard, 2001), to scoped memory in the Real-time Specification for Java (Boyapati *et al.*, 2003c), and to lazy modular upgrades in object-oriented database (Boyapati *et al.*, 2003a).

Banerjee and Naumann (2002a) demonstrated the use of object confinement to achieve representation independence. Their 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 (Banerjee & Naumann, 2002b).

Clarke *et al.* (2003) define a clever variant of confined types for the purpose of ensuring the integrity of components in the Enterprise JavaBeans framework. There are several interesting aspects to their work. They allow confinement to be specified in so called deployment descriptors. Thus the same set classes can be confined in one application and public in another. This is related to our use of generics for confining classes, with the difference that with generics the same class can be confined in several packages within the same application. On the other hand, their approach does not require additional syntax or changes to the existing code provided it already meets confinement invariants. Another interesting aspect of the work is that the unit of confinement is different. Rather than confining types within a package, the authors confine them within a Bean using the following rules (CB1-6): CB1 declares which types are confined ($\mathscr{C}2$ in our case), CB2 prevents confined types from appearing at the Bean boundary or in static variables (roughly equivalent to $\mathscr{C}1$), CB3 prevents widening of confined types (identical to $\mathscr{C}2$), CB4 prevents unconfined types to be cast to confined types, CB5 prevents confined code from accessing unconfined classes which have confined types in their signature, and finally CB6 states that confined classes may extend only one another or `Object` (a stronger version of $\mathscr{C}4$). Rule CB6 precludes confined classes from inheriting code from non-confined classes and thus sidesteps the issue of anonymous methods. The drawback is that a confined class may not inherit from an unconfined one. The paper observes that this has not been a problem in practice. The systems also differ in rules CB2 and CB5 which conspire to prevent the use of static variables to communicate across beans. Rule CB4 is essential as it prevents a form of spoofing in which an unconfined public subclass is used to leak reference to confined fields of the parent.

Type annotations have applications other than restricting object aliases. The work of Foster *et al.* (2002) extends standard type system with flow-sensitive type-qualifiers, which can be used for verifying a class of flow-sensitive properties. They implemented an efficient type-inference algorithm with practical applications such as analyzing locking behavior in the Linux kernel.

8 Conclusion

This paper has formalized the notion of *confined type* (Vitek & Bokowski, 2001) in the context of a minimal object calculus modeled on Featherweight Java. We also illustrated the application of confined types to security. A static type system that mirrors the informal rules of 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.

References

Abadi, M. and Cardelli, L. (1996) *A Theory of Objects.* Springer-Verlag.

Aldrich, J., Kostadinov, V. and Chambers, C. (2002) Alias annotations for program understanding. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA).*

Almeida, P. S. (1997) Balloon types: Controlling sharing of state in data types. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).*

Banerjee, A. and Naumann, D. A. (2002a) Representation independence, confinement and access control. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL).*

Banerjee, A. and Naumann, D. A. (2002b) Secure information flow and pointer confinement in a Java-like language. *Proceedings of the IEEE Computer Security Foundations Workshop.*

Boyapati, C. (2004) *SafeJava: A Unified Type System for Safe Programming.* PhD thesis, MIT.

Boyapati, C. and Rinard, M. (2001) A parameterized type system for race-free Java programs. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA).*

Boyapati, C., Lee, R. and Rinard, M. (2002) Ownership types for safe programming: Preventing data races and deadlocks. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA).*

Boyapati, C., Liskov, B., Shrira, L., Moh, C.-H. and Richman, S. (2003a) Lazy modular upgrades in persistent object store. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA).*

Boyapati, C., Liskov, B. and Shrira, L. (2003b) Ownership types for object encapsulation. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL).*

Boyapati, C., Salcianu, A., Beebee, W. and Rinard, M. (2003c) Ownership types for safe region-based memory management in real-time Java. *Proceedings of the ACM Conference on Programming Language Design and Implementation.*

Boyland, J., Noble, J. and Retert, W. (2001) Capabilities for aliasing: A generalisation of uniqueness and read-only. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).*

Clarke, D., Richmond, M. and Noble, J. (2003) Saving the world from bad Beans: Deployment-time confinement checking. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA).*

Clarke, D. (2001) *Object ownership and containment.* PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia.

Clarke, D. and Drossopoulou, S. (2002) Ownership, encapsulation and the disjointness of type and effect. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA).*

Clarke, D., Potter, J. and Noble, J. (1998) Ownership types for flexible alias protection. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA).*

Clarke, D., Noble, J. and Potter, J. M. (2001) Simple ownership types for object containment. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).*

Detlefs, D. L., Leino, K., Rustan M. and Nelson, G. (1998) *Wrestling with rep exposure.* Technical report SRC-RR-156. Digital Equipment Corporation Systems Research Center.

Foster, J. S., Fähndrich, M. and Aiken, A. (1999) A theory of type qualifiers. *Proceedings of the ACM Conference on Programming Language Design and Implementation.*

Foster, J. S., Terauchi, T. and Aiken, A. (2002) Flow-sensitive type qualifiers. *Proceedings of the ACM Conference on Programming Language Design and Implementation.*

Foster, J. S. (2002) *Type Qualifiers: Lightweight Specifications to Improve Software Quality.* PhD thesis, University of California, Berkeley.

Genius, D., Trapp, M. and Zimmermann, W. (1998) An approach to improve locality using Sandwich Types. *Proceedings of the 2nd Types in Compilation Workshop.*

Gong, L. (1998) Guarding objects. In: Vigna, G. (editor), *Mobile Agents and Security.* LNCS, vol. 576, pp. 1–23. Springer-Verlag.

Gong, L. (1999) *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* Addison-Wesley.

Grossman, D., Morrisett, G. and Zdancewic, S. (2000) Syntactic type abstraction. *ACM Trans. Program. Lang. & Syst.* **22**(6), 1037–1080.

Grothoff, C., Palsberg, J. and Vitek, J. (2001) Encapsulating objects with confined types. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA).*

Hagimont, D., Mossière, J., de Pina, X. R. and Saunier, F. (1996) Hidden software capabilities. *Proceedings of the 16th International Conference on Distributed Computing System.*

Hogg, J. (1991) Islands: Aliasing protection in object-oriented languages. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA).*

Hogg, J., Lea, D., Wills, A., de Champeaux, D. and Holt, R. (1992) The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, **3**(2).

Igarashi, A., Pierce, B. C. and Wadler, P. (2001) Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. & Syst.* **23**(3), 396–450.

Kent, S. J. H. and Maung, I. (1995) Encapsulation and Aggregation. *Proceedings of TOOLS Pacific 1995 – Technology of Object-Oriented Languages and Systems.* Prentice Hall.

Leavens, G. (1991) Modular specification and verification of object-oriented programs. *IEEE Software*, November, 72–80.

Leroy, X. and Rouaix, F. (1998) Security properties of typed applets. *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL).*

Levy, H. (editor) (1984) *Capability Based Computer Systems.* Digital Press.

Müller, P. and Poetzsch-Heffter, A. (1999) Universes: A type system for controlling representation exposure. In: Poetzsch-Heffter, A. and Meyer, J. (editors), *Programming Languages and Fundamentals of Programming.* Fernuniversität Hagen.

Müller, P. and Poetzsch-Heffter, A. (2000a) Modular specification and verification techniques for object-oriented software components. *Foundations of Component-Based Systems*, 137–159.

Müller, P. and Poetzsch-Heffter, A. (2000b) A type system for controlling representation exposure in Java. In: Drossopoulou, S., Eisenbach, S., Jacobs, B., Leavens, G. T., Müller, P. and Poetzsch-Heffter, A. (editors), *Formal Techniques for Java Programs.*

Noble, J., Vitek, J. and Potter, J. (1998) Flexible alias protection. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).*

Potanin, A., Noble, J. and Biddle, R. (2004a) Checking ownership and confinement. *Concurrency & Computation: Practice & Experience*, **16**(7), 671–687.

Potanin, A., Noble, J., Clarke, D. and Biddle, R. (2004b) Featherweight generic confinement. *International Workshop on Foundations of Object-oriented Languages (FOOL).*

Sewell, P. and Vitek, J. (2003) Secure composition of untrusted code: Box π, wrappers, and causality types. *J. Comput. Secur.* **11**(2), 135–187.

Vitek, Ja. and Bokowski, B. (2001) Confined types in Java. *Software–Practice & Exper.* **31**(6), 507–532.

Vitek, J. and Bryce, C. (2001) The JavaSeal mobile agent kernel. *Autonomous Agents & Multi-Agent Systems*, **4**.

Wallach, D., Balfanz, D., Dean, D. and Felton, E. (1997) Extensible Security Architectures for Java. *Proceedings of the 16th Symposium on Operating System Principles.*

Zhao, T., Palsberg, J. and Vitek, J. (2003) Lightweight confinement for Java. *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA).*