JFP **16** (6): 793–811, 2006. © 2006 Cambridge University Press
doi:10.1017/S0956796806006125 First published online 14 September 2006 Printed in the United Kingdom

Featherweight generic confinement

ALEX POTANIN, JAMES NOBLE
Victoria University of Wellington, New Zealand
(*e-mail:* {alex,kjx}@mcs.vuw.ac.nz)

DAVE CLARKE
CWI, Netherlands
(*e-mail:* dave@cwi.nl)

ROBERT BIDDLE
Carleton University, Otttawa, Canada
(*e-mail:* robert_biddle@carleton.ca)

Abstract

Existing approaches to object encapsulation either rely on ad hoc syntactic restrictions or require the use of specialised type systems. Syntactic restrictions are difficult to scale and to prove correct, while specialised type systems require extensive changes to programming languages. We demonstrate that confinement can be enforced cheaply in Featherweight Generic Java, with no essential change to the underlying language or type system. This result demonstrates that polymorphic type parameters can simultaneously act as ownership parameters and should facilitate the adoption of confinement and ownership type systems in general-purpose programming languages.

1 Introduction

Two main approaches to object instance encapsulation are under investigation in the literature. On one hand, programming conventions, such as Islands (Hogg, 1991) and various kinds of Confined Types (Vitek & Bokowski, 2001; Clarke *et al.*, 2003) use tailored restrictions on programs to provide confinement guarantees for programs in existing programming languages. This approach was recently proven to be sound (Zhao *et al.*, 2006). On the other hand, ownership type systems (Clarke *et al.*, 1998), originating from the formalisation of Flexible Alias Protection (Noble *et al.*, 1998), require quite significant modifications to programming languages. In particular, languages like Joe, Universes, Alias Java, and Safe Concurrent Java, depend upon *ownership parameterisation* within the type system (Clarke & Drossopoulou, 2002; Müller & Poetzsch-Heffter, 1999; Aldrich *et al.*, 2002; Boyapati *et al.*, 2003). All of these type systems are distinct, but they only support ownership parameterisation, not type parameters.

This article continues the efforts to provide effective object encapsulation within practical programming languages. The key insight behind this paper is that confinement and ownership type systems can readily be modelled within existing parametric polymorphic type systems: in fact, we demonstrate that confinement systems for object encapsulation within static protection domains can be subsumed completely within a basic generic type system. This is achieved by using a single type parameter space to carry *both* generic and ownership information. As a result, we can enforce confinement in Featherweight Generic Java (Igarashi *et al.*, 2001) "almost for free" – with no change to the underlying language or type system – by additionally enforcing some simple visibility rules and constraints on program structure.

Featherweight Generic Confinement is a minimalist confinement scheme that leverages parametrically polymorphic types to enforce static confinement. This paper is an expanded version of Potanin *et al.* (2004b) presenting the final version of our type system and incorporating *manifest ownership* (Clarke, 2002) to better address compatibility with plain FGJ programs (Potanin *et al.*, 2004). Our main goal is to obtain a simpler formalism with few new concepts. We hope that this approach will facilitate the adoption of confinement and ownership type systems by general-purpose programming languages.

The next section briefly introduces the notion of encapsulation and confinement, in particular the kind of confinement used in Confined Types (Vitek & Bokowski, 2001) – the primary topic of this article. We then present FGJ+c that reuses the type soundness of FGJ to support a simple confinement invariant, ensuring that confined classes may not be accessed outside a static protection domain (effectively a Java package). We present the additional constraints required of programs in FGJ+c, prove a confinement invariant, and conclude with a discussion of our prototype implementation and plans for future work.

2 Confinement

Islands, confinement, and ownership are all essentially forms of object encapsulation. All of these schemes are attempts to establish an encapsulation *boundary* that protects some objects *inside* the boundary from direct access by other objects *outside* that boundary. Where these proposals differ from earlier programming language encapsulation and module systems is that they restrict access to objects at runtime: that is, they constrain values of pointers or references to objects in object-oriented systems, rather than merely accesses to field and method names. These schemes enforce a *confinement invariant* which states that objects outside a particular boundary may not access objects inside that boundary.

The differences between these systems can be observed by noting which objects constitute the insides, the outsides, and the boundaries and how these three sets are expressed (Noble *et al.*, 2003). For example, in Confined Types (Vitek & Bokowski, 2001), the unit of confinement is a Java package: all the instances of public classes within that package form the encapsulation boundary; all the instances of package-scoped classes (known as *confined classes*) are inside the boundary, and instances of classes in any other package are outside the boundary. This means that an instance of a class may access an instance of a public class belonging to any package, but may only access those instances of confined classes belonging to its own package.

What confinement means in practice is that any code written in one confined domain (say one Java package) should, when executed, never *directly* refer to an instance of a class inside the boundary of another confined domain. References stored in object fields must be restricted: a field of a class cannot hold an object that is encapsulated inside a different package. The execution of a class's methods must also be restricted: methods cannot access confined classes of other packages. Note, however, that this prohibition refers only to direct accesses: *indirect* access is permitted – indeed, is encouraged. Public classes (or instances of public classes) thus provide an interface to the private instances in their package.

Zhao *et al.* (2006) have formulated a confinement invariant in terms of the expressions within methods. Basically, if an expression (or any of its subexpressions) can possibly evaluate to some object o, that object must be visible in the context of the method. In some more detail: if d_0 is a subexpression of d, and d_0 evaluates to e (denoted $d_0 \overset{*}{\rightarrow} e$), then any object denoted by e must be visible in the class containing d.

3 Featherweight Generic Java + Confinement

In this section we present *Featherweight Generic Java + Confinement* (FGJ+c hereafter) which embodies our confinement scheme. After outlining the main principles behind FGJ+c, we give a formal presentation of FGJ+c's type system and a proof of its confinement invariant.

3.1 Program structure

The key idea behind Generic Confinement is to use generic type parameters to carry confinement information as well as type information. Following the traditional approach (Clarke, 2002) we require that every pure FGJ+c class has at least one type parameter to carry this confinement information. Following both the confinement and ownership literature, we call this extra parameter the *owner* type parameter. In the case of confinement (as in FGJ+c), the owner corresponds to the package to which an object is confined: in the per-object ownership type systems an object's owner will be another object.

We use the *last* type parameter to record an object's owner to promote upwards compatibility and because our implementation will allow confinement parameters to be elided. All FGJ+c classes descend from a new class `CObject<O>` (for confinable object) that has just one parameter; all of its subclasses must invariantly preserve this owner. The preservation of owner over subtyping lies at the foundation of Generic Confinement.

Figure 1 shows a (functional) stack implementation in FGJ+c. Note that all class names are prefixed by a package identifier, thus `uStack` is a `Stack` in package `u`. This is a convention to indicate the package within which each class is defined. We assume that each package is identified by a single lower case letter that prefixes every class inside a package. Names of classes that belong to a default package start with a capital letter. Note also that classes which extend class `World` are used to indicate

```
// Package u (from the word util).

// Class uNode serves as a "null object" for the purposes of implementing a uStack.
class uNode<T extends Any, Owner extends World> extends CObject<Owner> {
  uNode() { super(); }
}

// Class uStackNode is a simple stack node used by uStack.
class uStackNode<T extends Any, Owner extends World> extends uNode<T, Owner> {
  T element; uNode<T, Owner> nextNode;
  uStackNode(T element, uNode<T, Owner> nextNode) {
    super(); this.element = element; this.nextNode = nextNode;
  }
}

// Class uStack implements a simple functional stack.
class uStack<T extends Any, Owner extends World> extends CObject<Owner> {
  uNode<T, Owner> root;
  uStack(uNode<T, Owner> root) {
    super(); this.root = root;
  }

  uStack<T, Owner> push(T element) {
    return new uStack<T, Owner>(new uStackNode<T, Owner>(element, this.root));
  }

  uStack<T, Owner> pop() {
    return new uStack<T, Owner>(((uStackNode<T, Owner>) this.root).nextNode);
  }

  T top() {
    return ((uStackNode<T, Owner>) this.root).element;
  }
}
```

Fig. 1. FGJ+c Stack Example. This FGJ+c code demonstrates a possible implementation of a functional stack inside package u.

ownership. For each package we use lower case letter (e.g. u) for its name and the same upper case letter (e.g. U) for the owner class corresponding to the package.

Each uStack has two type parameters, the first being the type of items to be stored in the stack, and the second being the ownership of that stack instance. This illustrates that FGJ+c provides both *type polymorphism* (stack can hold different item types) and *ownership polymorphism* (stack can be confined to different domains). The item type parameter of the stack is bound by class Any – this allows any subclass of CObject<O> to be used in its place for *any* owner parameter. Please observe that we cannot make uNode owned by U, since it has to be created outside of uStack. This is easily fixed by extending the type system with an extra field initialisation capability.

Figure 2 presents an example of utilising our stack. Package s contains two classes. sPassword stores a secret ID, and sPasswordManager stores a stack of passwords utilising uStack. The stack and passwords stored inside sPasswordManager are confined to package s because their full types include the owner class S that can only be written inside package s: uStack<sPassword<S>, S>. If we try to access the contents of sPasswordManager's stack in a different package (e.g. by calling getSecretPassword in package m) we won't be able to assign the result to anything or cast it to an appropriate type to make use of it. Because the owner parameter S is preserved over the subtyping hierarchy there is no way around this restriction.

```
// Package s (from the word secret).

class sPassword<Owner extends World> extends CObject<Owner> {
  int secretID;
  sPassword(int secretID) {
    super(); this.secretID = secretID;
  }
}

class sPasswordManager<Owner extends World> extends CObject<Owner> {
  sPasswordManager() { super(); }

  uStack<sPassword<S>, S> createStack() {
    return new uStack<sPassword<S>, S>(new uNode<sPassword<S>, S>())
  }

  uStack<sPassword<S>, S> addSecretPassword(int secretID) {
    return this.createStack().push(new sPassword(secretID));
  }

  // This method can only be called inside the s package, since outside of
  // this package we cannot write the type of the return element due to owner S
  // being involved.
  sPassword<S> getSecretPassword() {
    return this.addSecretPassword(7).top();
  }
}

// Package m (from the word main).

class mMain<Owner extends World> extends CObject<Owner> {
  mMain() { super(); }

  sPasswordManager<M> createPasswordManager() {
    return new sPasswordManager<M>();
  }

  void addSecretPassword() {
    this.createPasswordManager().addSecretPassword(42);
  }

  void getSecretPassword() {
    // We cannot perform a call to the following method or assign the result
    // to anything, including the super class CObject<S>, since we are not
    // allowed to use owner S outside s package.
    this.createPasswordManager().getSecretPassword()
  }
}
```

Fig. 2. FGJ+c confinement violation example. This FGJ+c code demonstrates how confinement inside package `s` can be enforced using owner classes. Note that although `void` is not part of FGJ, it simplifies the presentation of the idea in this figure.

3.2 Packages and owner classes

In FGJ+c confinement domains are static Java packages, but we need to represent them within the FGJ type system so that we can bind the owner parameters. For this reason, we use parameter-less FGJ classes that form a separate class hierarchy extending `World` to represent these domains. Because the classes that represent domains (again like Java packages) are not actually part of the program, they should not be instantiated during the execution of an FGJ+c program. We call these types *owner classes.*

Confinement in FGJ+c is enforced simply by requiring that any concrete owner (other than `World`) can only appear within the body of classes within its own

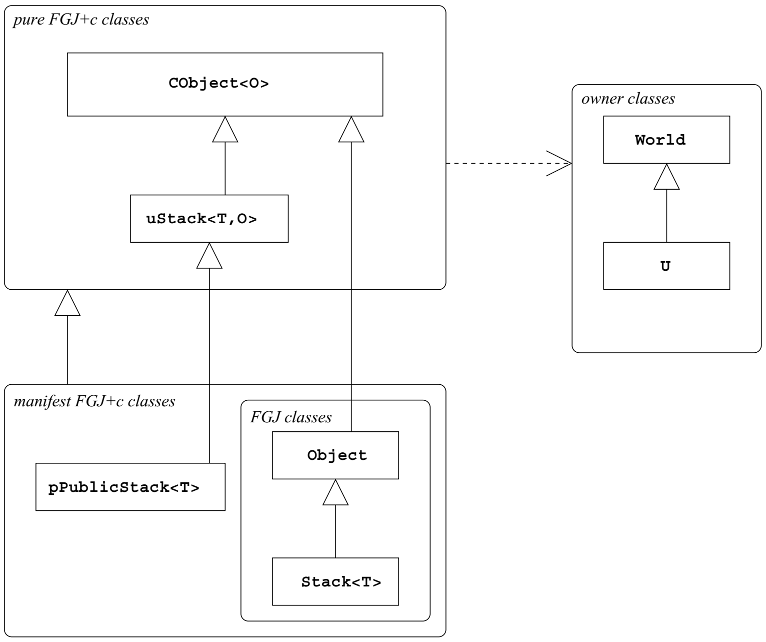

Fig. 3. FGJ+c Classes and Owner Classes. *Pure* FGJ+c classes have an explicit owner type parameter. *Manifest* FGJ+c classes have an owner fixed when subclassing a pure FGJ+c class. *Owner* classes lie outside the FGJ+c class hierarchy because they cannot be instantiated in FGJ+c programs; pure FGJ+c classes use them to bind their owner type parameters (as shown by the dashed arrow on the diagram).

package. In other words, owner `S` can appear within the definition of classes such as `sPassword` but owner `U` cannot. Note that class names themselves are not restricted *per se*; this is why a name like `uStack` can appear in package `s`.

3.3 Manifest ownership

We adopt the concept of *manifest ownership* (Clarke, 2002) to allow classes without explicit owner type parameters (e.g., all classes in a *vanilla* (standard) FGJ program). Somewhere in the subclass hierarchy, the owner is fixed and all the objects of that class have the same owner. For example, here is how `Object` fits into the FGJ+c class hierarchy:

```
class Object extends CObject<World> { ... }
```

With this definition `Object` and every FGJ class under it has a default owner parameter `World` (thus making them publically accessible).

Figure 3 shows the relationships between owner classes and program classes in FGJ+c. Owner classes inherit from the class `World`, and there is one owner class corresponding to each FGJ+c package. The owner class hierarchy lies outside the `CObject` hierarchy of pure and manifest FGJ+c classes – these are the only owners that can occur in a FGJ+c program. Manifest FGJ+c classes have an owner class corresponding to their owner, which is not written out explicitly, but rather is found among its superclasses in the FGJ+c class hierarchy. Vanilla FGJ classes form a subset of manifest FGJ+c classes. The FGJ+c class hierarchy has two different roots: one for all the program classes (`CObject<O>`) and one for uninstantiable owner classes (`World`).

To demonstrate manifest ownership, consider creating a public stack as follows[1]:

```
class pPublicStack<T extends Object> extends uStack<T, World>
  { ... }
```

In this case, the *owner* of class `PublicStack` is `World`, and thus all of its instances are owned by `World`, while any use of class `pPublicStack` requires no owner type parameter.

Manifest ownership allows the following familiar declaration of a public `Stack` object, which is indistinguishable from that of FGJ:

```
class Stack<T extends Object> extends Object { ... }
```

The important difference is that in FGJ+c, class `Stack` has an owner `World` coming from `Object`'s super class `CObject<World>`. Figure 3 shows the examples in this subsection as a class diagram.

Manifest ownership also allows the definition of fully confined classes, that is, classes whose instances can never be used outside their defining package. Consider the definition of the `Link` class in package `l`:

```
class lLink<T extends Any> extends CObject<L> { ... }
```

By the virtue of manifest ownership, `Link`'s owner is fixed to be `L`. `L` owner class is only visible within package `l`, thus ensuring all instances of `Link` will be confined within that package. Because we are declaring a class inside package `l` we cannot "fix" the owner to anything other than `L`. The following example would be invalid:

```
class qC extends CObject<L> { ... }
```

That is, it is not possible to write owner `L` inside a different package `q`.

3.4 `Any` ***type bound***

FGJ requires every type variable to be bound. To increase the expressiveness of our system we introduce a very limited form of anonymous type parameters. The type `Any` may only appear as the bound of a formal generic type parameter. Any class with any ownership may be passed as an actual parameter if a formal parameter is bound by `Any`. For example, in figure 1, `uStack`'s first type parameter is bound by `Any`. We add the `Any` type bound to FGJ+c by introducing the empty class `Any` above `CObject<O>` in the hierarchy. FGJ+c's rules prevent the type or class being

[1] Manifest ownership is not restricted to fixing the owner to be `World` – it can be any owner class.

Syntax:

$$\begin{array}{l}
\texttt{T ::= X | N} \\
\texttt{N ::= C<}\overline{\texttt{T}}\texttt{>} \\
\texttt{L ::= class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \{\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\} \\
\texttt{K ::= C(}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{) } \{ \texttt{ super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{; } \} \\
\texttt{M ::= <}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) } \{ \texttt{ return e; } \} \\
\texttt{e ::= x | e.f | e.m<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{) | new N(}\overline{\texttt{e}}\texttt{) | (}\overline{\texttt{N}}\texttt{)e}
\end{array}$$

`X` ranges over the type variables and `N` ranges over the nonvariable types.
Δ type environment: a mapping from type variables to nonvariable types.
Γ type environment: a mapping from variables to types.
CT class table: a mapping from class names `C` to class declarations `L`.

Fig. 4. **FGJ+c Syntax (Identical to FGJ Syntax from Igarashi et al).**

used anywhere except as a type bound. The importance of not allowing other uses of `Any` lies in the fact that `Any` doesn't have an owner and thus we cannot allow it as an FGJ+c type if we want preservation of owners through subtyping – an essential property of FGJ+c.

This type bound does not allow any constraints on owners or types of actual parameters. More flexible schemes (e.g. wildcards or variance (Igarashi & Viroli, 2002)) will remove these restrictions, allowing a single type parameter to be independently bounded for type and ownership. Lacking variance or multiple inheritance, we are unable to express such flexible bounds in a system built on top of pure FGJ. However, this technique will be applicable in other systems.

4 FGJ+c Definition

FGJ+c can be considered a strict subset of FGJ, that is, every FGJ+c program is an FGJ program, if we allow for a different root of the class hierarchy. FGJ+c adds some extra restrictions that leverage FGJ's proven type soundness to provide confinement. Every FGJ+c program must meet the FGJ rules (Igarashi *et al.*, 2001) along with additional rules presented in the figures of this section. For reference, figure 4 shows the FGJ syntax from Igarashi *et al.* (2001). To simplify our presentation, we assume that owner classes are syntactically distinguishable. Owners have the syntax:

```
O ::= OVar | OCon
```

where `O` ranges over all owners, `OVar` ranges over owner variables, and `OCon` ranges over concrete owners such as `World` and the owner classes corresponding to packages. Pure FGJ+c types and classes are written to include an owner class as their last type parameter or argument, which can be distinguished using the following syntax:

$$\begin{array}{l}
\texttt{N ::= C<}\overline{\texttt{T}}\texttt{, O>} \\
\texttt{L ::= class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{, OVar} \triangleleft \texttt{OCon>} \triangleleft \texttt{N} \{\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\}.
\end{array}$$

FGJ+c Judgements:

$\mathit{owner}_\Delta(\mathtt{T})$	Determines owner of type `T`.
$\Delta \vdash \mathtt{T\ OK{+}c}$	Type `T` is OK.
$\mathit{visible}_\Delta(\mathtt{O},\mathtt{D})$	Owner `O` is visible in class `D`.
$\mathit{visible}_\Delta(\mathtt{T},\mathtt{D})$	Type `T` is visible in class `D`.
$\Delta;\Gamma \vdash \mathit{visible}(\mathtt{e},\mathtt{D})$	Expression `e` is visible in class `D`.
$\mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \mathtt{FGJ{+}c\ IN\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}$	Method `m` definition is OK.
$\mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{FGJ{+}c}$	Class `C` definition is OK.

Fig. 5. **FGJ+c Judgements.**

Bound of type (from FGJ):

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

FGJ+c Owner Lookup Function:

$$\begin{array}{lcl}
\mathit{owner}_\Delta(\mathtt{X}) & = & \mathit{owner}_\Delta(\mathit{bound}_\Delta(\mathtt{X})) \\
\mathit{owner}_\Delta(\mathtt{C<}\overline{\mathtt{T}},\ \mathtt{O>}) & = & \mathtt{O} \\
\mathit{owner}_\Delta(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) & = & \mathit{owner}_\Delta(\mathtt{N}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]) \\
 & & \text{where } \mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\{\overline{\mathtt{T'}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}
\end{array} \qquad \text{(FGJ+c-Owner)}$$

FGJ+c Types:

$$\frac{\Delta \vdash \mathtt{T} <: \mathtt{CObject<O>} \qquad \Delta \vdash \mathtt{O} <: \mathtt{World}}{\Delta \vdash \mathtt{T\ OK{+}c}} \qquad \frac{\Delta \vdash \mathtt{X} <: \mathtt{Any}}{\Delta \vdash \mathtt{X\ OK{+}c}} \qquad \text{(FGJ+c-Type)}$$

Fig. 6. **FGJ Bound, FGJ+c Owner Lookup Function, and FGJ+c Type Rule.**

4.1 FGJ+c programs

Any FGJ+c program is an FGJ program that meets the following requirement: all classes must satisfy either FGJ+c-Class (for generic classes) or FGJ+m-Class (for manifest classes). A corresponding rule (FGJ+c-Type) ensures that the only types used in FGJ+c programs are subtypes of `CObject<O>` for some owner `O`.

Figure 5 shows the type judgements used in FGJ+c rules and figures 6, 7 and 8 give the rules used to constrain FGJ programs. These rules deal with three concerns: firstly, they ensure that every type has an owner (as the owner contains information to determine the visibility of a type, figure 6); secondly, they determine which types

Owner Visibility:

$$visible_\Delta(\mathtt{O},\ \mathtt{D}) \quad = \quad \mathtt{O} \in owners_\Delta(\mathtt{D}) \cup \{\pi_{\mathtt{D}}, \mathtt{World}\}^\dagger \qquad \text{(V-OWNER)}$$

where

$$owners_\Delta(\mathtt{D}) \quad = \quad \begin{cases} \{owner_\Delta(\mathtt{N'}) \mid \mathtt{N'} \in \overline{\mathtt{N}}, \mathtt{N}\}, \\ \quad \text{if } CT(\mathtt{D}) = \mathtt{class\ D<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\{\ldots\} \\ \{\mathtt{OVar}\} \cup \{owner_\Delta(\mathtt{N'}) \mid \mathtt{N'} \in \overline{\mathtt{N}}\}, \\ \quad \text{if } CT(\mathtt{D}) = \mathtt{class\ D<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}},\ \mathtt{OVar} \triangleleft \mathtt{OCon>} \triangleleft \mathtt{N}\{\ldots\} \end{cases}$$

Type Visibility:

$$\frac{visible_\Delta(owner_\Delta(\mathtt{T}),\ \mathtt{D})}{visible_\Delta(\mathtt{T},\ \mathtt{D})} \qquad \frac{bound_\Delta(\mathtt{x}) = \mathtt{Any}}{visible_\Delta(\mathtt{x},\ \mathtt{D})} \qquad \text{(V-TYPE)}$$

Term Visibility:

$$\frac{\Delta;\Gamma \vdash \mathtt{x} : \mathtt{T} \qquad visible_\Delta(\mathtt{T},\ \mathtt{D})}{\Delta;\Gamma \vdash visible(\mathtt{x},\ \mathtt{D})} \qquad \text{(V-VAR)}$$

$$\frac{\Delta;\Gamma \vdash visible(\mathtt{e},\ \mathtt{D}) \qquad \Delta;\Gamma \vdash \mathtt{e.f_i} : \mathtt{T} \qquad visible_\Delta(\mathtt{T},\ \mathtt{D})}{\Delta;\Gamma \vdash visible(\mathtt{e.f_i},\ \mathtt{D})} \qquad \text{(V-FIELD)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{T} \qquad visible_\Delta(\mathtt{T},\ \mathtt{D}) \\ \Delta;\Gamma \vdash visible(\mathtt{e},\ \mathtt{D}) \qquad \Delta;\Gamma \vdash visible(\overline{\mathtt{e}},\ \mathtt{D})\end{array}}{\Delta;\Gamma \vdash visible(\mathtt{e.m}(\overline{\mathtt{e}}),\ \mathtt{D})} \qquad \text{(V-INVK)}$$

$$\frac{\Delta;\Gamma \vdash visible(\overline{\mathtt{e}},\ \mathtt{D}) \qquad visible_\Delta(\mathtt{N},\ \mathtt{D})}{\Delta;\Gamma \vdash visible(\mathtt{new\ N}(\overline{\mathtt{e}}),\ \mathtt{D})} \qquad \text{(V-NEW)}$$

$$\frac{\Delta;\Gamma \vdash visible(\mathtt{e},\ \mathtt{D}) \qquad visible_\Delta(\mathtt{N},\ \mathtt{D})}{\Delta;\Gamma \vdash visible((\mathtt{N})\ \mathtt{e},\ \mathtt{D})} \qquad \text{(V-CAST)}$$

† $\pi_{\mathtt{D}}$ is the owner class corresponding to the package to which D belongs.

Fig. 7. **FGJ+c Visibility Rules.**

are visible in a given class (figure 7); and finally, they propagate and check the desired constraints for fields and methods (figure 8).

FGJ+c Owner Lookup Function. This function in figure 6 is used by the majority of the rules to look up an owner class corresponding to a given type. This is either an explicit owner used as the last type parameter, or in the case of a manifest FGJ+c class, the owner of its superclass.

FGJ+c Methods:

$$\frac{\begin{array}{c} \Delta = \{\overline{\mathtt{X}} \mathrel{<:} \overline{\mathtt{N}},\ \overline{\mathtt{Y}} \mathrel{<:} \overline{\mathtt{P}}\} \qquad \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}, \overline{\mathtt{P}}\ \mathtt{OK{+}c} \\ visible_\Delta(\overline{\mathtt{T}},\ \mathtt{C}) \qquad visible_\Delta(\mathtt{T},\ \mathtt{C}) \qquad visible_\Delta(\overline{\mathtt{P}},\ \mathtt{C}) \\ \Delta;\ \overline{\mathtt{x}} : \overline{\mathtt{T}},\ \mathtt{this} : \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash visible(\mathtt{e_0},\ \mathtt{C}) \end{array}}{\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>\ T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)}\{\ \mathtt{return\ e_0;}\ \}\ \mathtt{FGJ{+}c\ IN\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}} \quad \text{(FGJ+c-Method)}$$

FGJ+c Classes:

$$\frac{\begin{array}{c} \mathtt{N} = \mathtt{C'<}\overline{\mathtt{T'}}\mathtt{, OVar>} \qquad \Delta = \overline{\mathtt{X}} \mathrel{<:} \overline{\mathtt{N}},\ \mathtt{OVar} \mathrel{<:} \mathtt{OCon} \\ visible_\Delta(\mathtt{OCon},\ \mathtt{C}) \qquad \Delta \vdash \mathtt{N}, \overline{\mathtt{T}}\ \mathtt{OK{+}c} \\ \forall \mathtt{N'} \in \overline{\mathtt{N}} : \mathtt{N'} = \mathtt{Any}\ \mathtt{or}\ \Delta \vdash \mathtt{N'}\ \mathtt{OK{+}c} \\ \overline{\mathtt{M}}\ \mathtt{FGJ{+}c\ IN\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}},\ \mathtt{OVar} \lhd \mathtt{OCon>} \\ visible_\Delta(\overline{\mathtt{T}},\ \mathtt{C}) \qquad visible_\Delta(\overline{\mathtt{N}},\ \mathtt{C}) \end{array}}{\mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}},\ \mathtt{OVar} \lhd \mathtt{OCon>} \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{FGJ{+}c}} \quad \text{(FGJ+c-Class)}$$

$$\frac{\begin{array}{c} \Delta = \overline{\mathtt{X}} \mathrel{<:} \overline{\mathtt{N}} \qquad visible_\Delta(\mathtt{N},\ \mathtt{C}) \\ \Delta \vdash \mathtt{N}, \overline{\mathtt{T}}\ \mathtt{OK{+}c} \qquad \overline{\mathtt{M}}\ \mathtt{FGJ{+}c\ IN\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \\ \forall \mathtt{N'} \in \overline{\mathtt{N}} : \mathtt{N'} = \mathtt{Any}\ \mathtt{or}\ \Delta \vdash \mathtt{N'}\ \mathtt{OK{+}c} \\ visible_\Delta(\overline{\mathtt{T}},\ \mathtt{C}) \qquad visible_\Delta(\overline{\mathtt{N}},\ \mathtt{C}) \end{array}}{\mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{FGJ{+}c}} \quad \text{(FGJ+m-Class)}$$

Fig. 8. **FGJ+c Method and Class Rules.** FGJ+c program can only contain FGJ+c valid classes. Additionally any FGJ+c program or class has to meet all of the FGJ rules.

FGJ+c Types. The first rule in figure 6 states that the only types allowed are the subtypes of `CObject<O>`. These types can either be pure FGJ+c (have an explicit owner parameter) or manifest FGJ+c (have an implicit owner fixed in a superclass hierarchy). Note that the type variables will be classified as valid FGJ+c types by the virtue of their bounds (using FGJ's type well-formedness rules and the restriction that all the nonvariable types are valid in FGJ+c). In addition, a root type `CObject<O>` is also allowed. The second rule admits the type variables bound by `Any`.

4.2 Visibility

The visibility rules shown in figure 7 form the foundation of FGJ+c: they determine which owners, types, and terms are visible, and thus usable, within a given class. There are three sets of rules: those that determine *owner* visibility, *type* visibility, and *term* visibility. We utilise a function $\pi_{\mathtt{D}}$ that returns the owner class corresponding to the package to which `D` belongs.

The *owner* visibility predicate simply checks that a given owner is either the owner of the current class, corresponds to the class's package (*e.g.* $\pi_{\mathtt{D}}$ inside class `D`), is a public owner (denoted `World`), or is the owner of one of the type parameters – roughly following Zhao *et al.* (2006). The owner variable of a class – and any of its type parameters – are always visible because Generic Confinement views passing type parameters as granting permission to access the actual argument types (see 5.1). Method type parameters do not grant this permission.

The *type* visibility predicate allows the use of types when their owner parameter is visible. As a special case, a type variable bound by `Any` is considered to be always visible within the class in which it is declared. *Term* visibility rules are defined inductively on the structure of FGJ terms. For each subexpression, the rules determine whether the type of that subexpression is visible according to the type visibility rules.

Consider the following code example, where `P` refers to an owner class marking instances confined to package `p`.

```
pList<qFoo<R>, S>
```

This describes a list declared in package `p` storing items declared in package `q` that are confined to package `r`, while list itself is confined to package `s`. This list is allowed access to classes confined to `p` (since the code performing the access is already inside package `p`), `s` (since the list instance is confined in that package during the execution) and to instances confined to package `r` because one of the list's type parameters is confined in that package (*i.e.*, is owned by `R`). Any classes confined to `q` or any other package cannot be accessed inside the list.

Visibility is the key to Generic Confinement. It specifies the subset of valid FGJ programs that are considered valid FGJ+c programs. Any FGJ class that contains expressions violating visibility by accessing the types private to a different confinement domain is declared invalid by failing one of the FGJ+c visibility rules.

4.3 Propagation of constraints in classes and methods

Figure 8 shows how visibility constraints are propagated through classes to their fields and methods and eventually to expressions. These rules say that *every* type appearing in the program, even as a subexpression, *must* be visible in the current class. For classes, (FGJ+c-Class) and (FGJ+m-Class), we check that all the field types are visible and that the bounds on the type variables are also visible to the current class. The difference between the two class rules lies in the presence of an explicit owner parameter in the class being declared. For pure FGJ+c classes, an owner parameter is present and we require that its immediate superclass has the same owner parameter. For manifest FGJ+c classes, the owner parameter is no longer explicit, so we need to check that our superclass is visible with respect to the current class. Finally, `Any` is allowed as nonvariable bound for non-owner type parameters.

The rules for classes (FGJ+c-Class) and (FGJ+m-Class) and types (FGJ+c-Type) combine together to ensure that every class has an owner (or that its bound does, in the case of type variables) *and* that this owner is preserved through subtyping. `Any` is treated as a special case of a type bound allowing any type with any owner if required. This does not break visibility rules since in generic confinement having an additional owner as part of one of class's type parameters gives you permission to access classes confined to that owner. We address this issue further in our discussion in section 5. In addition, we ensure that the all types instantiated in programs are a subtype of `CObject<O>` for some owner class `O` and thus also prevent owner classes from being instantiated. We also get the restriction that only owner classes can be used as owners.

For methods (FGJ+C-METHOD), we check that the argument and return types are visible, that the method body satisfies the visibility constraint, that all subexpressions use visible types, and again that the type variables of the method have bounds which are visible.

An alternative design would be to use visibility checks throughout the existing FGJ rules (as in CFJ by Zhao *et al.* (2006)), rather than build the appropriate checks into the (FGJ+C-CLASS) and (FGJ+M-CLASS) rules. The advantage of our separate approach is that we can reuse all of the FGJ rules (and proofs) and that we demonstrate that FGJ+c doesn't need a type system stronger than FGJ.

These rules guarantee that FGJ+c programs do not break confinement, allowing us to prove a confinement invariant.

4.4 Confinement invariant

The confinement invariant states that types that are not visible within the current package are not reachable. We assume that we only deal with programs for which all classes are valid FGJ+c classes as given by the rules in figure 8. We prove that during execution, an expression cannot result in an instance of a class that is not *visible* within the current context. This result relies on the fact that the owner parameter is preserved in the class hierarchy.

Lemma (Ownership Invariance). *If* $\Delta \vdash \mathtt{S} <: \mathtt{T}$ *and* $\Delta \vdash \mathtt{T} <: \mathtt{CObject<O>}$*, then* $\mathit{owner}_\Delta(\mathtt{S}) = \mathit{owner}_\Delta(\mathtt{T}) = \mathtt{O}$.

Proof By induction on the depth of the subtype hierarchy. By FGJ+C-CLASS and FGJ+M-CLASS a FGJ+c class has the same owner parameter as its superclass. □

Theorem (Confinement Invariant). *Let* $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$ *be a subexpression appearing in the body of a method of a well-formed FGJ+c class* `C`*. Then: If* $\mathtt{e} \overset{*}{\rightarrow} \mathtt{new\ D<}\overline{\mathtt{T_D}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}$*, then* $\mathit{visible}_\Delta(\mathtt{D<}\overline{\mathtt{T_D}}\mathtt{>}, \mathtt{C})$.

Proof Because the class is a well-formed FGJ+c class, its methods are well-formed FGJ+c methods. This and the FGJ's subformula property (rule inversion) imply that, for appropriate Δ and Γ, both $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$ and $\Delta;\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})$ hold. From this we can derive $\mathit{visible}_\Delta(\mathtt{T}, \mathtt{C})$, and hence $\mathit{visible}_\Delta(\mathit{owner}_\Delta(\mathtt{T}), \mathtt{C})$ or $\mathit{bound}_\Delta(\mathtt{T}) = \mathtt{Any}$. By FGJ's subject reduction property, there is a $\mathtt{T}'$ such that $\Delta;\Gamma \vdash \mathtt{new\ D<}\overline{\mathtt{T_D}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{T}'$, where $\Delta \vdash \mathtt{T}' <: \mathtt{T}$. Furthermore, we have that $\Delta;\Gamma \vdash \mathtt{new\ D<}\overline{\mathtt{T_D}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{D<}\overline{\mathtt{T_D}}\mathtt{>}$, and hence clearly $\Delta \vdash \mathtt{D<}\overline{\mathtt{T_D}}\mathtt{>} <: \mathtt{T}'$, and $\Delta \vdash \mathtt{D<}\overline{\mathtt{T_D}}\mathtt{>} <: \mathtt{T}$. By the Ownership Invariance lemma, either $\mathit{owner}_\Delta(\mathtt{D<}\overline{\mathtt{T_D}}\mathtt{>}) = \mathit{owner}_\Delta(\mathtt{T})$ or $\mathit{bound}_\Delta(\mathtt{T}) = \mathtt{Any}$, from which we deduce $\mathit{visible}_\Delta(\mathit{owner}_\Delta(\mathtt{D<}\overline{\mathtt{T_D}}\mathtt{>}), \mathtt{C})$, and hence $\mathit{visible}_\Delta(\mathtt{D<}\overline{\mathtt{T_D}}\mathtt{>}, \mathtt{C})$. □

5 Discussion

On one hand, every FGJ+c program is a valid FGJ program – it type checks and executes following FGJ's evaluation rules, with no accesses to the confined instances of classes. On the other hand, every FGJ program can be mapped to a manifest FGJ+c classes by an FGJ's `Object` extending `CObject<World>`. Thus all FGJ programs are FGJ+c. The special treatment that the owner classes receive when interpreted by the additional FGJ+c rules allows us to guarantee a confinement invariant. In this section we delve into a few of the more interesting aspects of FGJ+c.

5.1 Generic confinement

Generic confinement raises two issues that were also discussed by Zhao *et al.* (2006). First, as with ownership type systems (Clarke, 2002), instantiating a class with an actual owner parameter can be seen as giving the instances of that class permission to access other objects owned by the actual owner parameter – in our case, confined within the package corresponding to the actual owner parameter. This is made explicit in the owner visibility rule, which explicitly checks these permissions, that is, which ensures an owner is visible if it is the owner of any of the (actual) type parameters.

Second, if we consider the following FGJ+c expression evaluated using the FGJ rules:

```
class pFoo {
 ...
 Bar<Q> m() {
  return (new uMap<Integer<World>, Bar<Q>, P>).get(new Integer(42));
 }
}
```

then we can see that although this code could be located *anywhere* (in this case, inside method `m` of an unrelated class `Foo` in some package `p`), *inside* the evaluation of the `get` method objects private to the `uMap`'s package can be accessed (as can objects owned by `World` or by the owner of the `Foo` class's package `P`) and can appear as intermediate results as the expression is evaluated. This breaks neither our confinement invariant nor that of Zhao *et al.* (2006), since the only objects that the `Foo` instance executing method `m` can access *directly* are the final results of evaluating whole subexpressions, such as the constructor call or the `get` invocation. The `Map` constructor or `get` method may well create or reference other objects that are private to the `Map`, but `Foo` itself will not have permission to access these other objects directly, and our system prevents such accesses.

An important observation is that we still need to ensure owner preservation over subtyping in FGJ+c. If, for example, our language were to support full variance (Igarashi & Viroli, 2002) rather than basic wildcards, then it would be much harder for us to ensure that subtypes like: `uStack<S> <: uStack<World>` were disallowed. Hence a full exploration of the interaction of variance with Generic Confinement poses an open question.

5.2 Towards ownership

FGJ is a functional subset of Java that omits its imperative aspects such as assignments, field updates, local variables, etc. We are extending FGJ with imperative features following Pierce (2002). We have found that we can formulate a containment invariant comparable to that of ownership types with little substantial additions to the visibility and preservation of owners over subtyping already present in FGJ+c. FGJ is a sufficient platform for reasoning about confinement: the advantage of formulating confinement as a small addition to FGJ is a clean system like FGJ+c.

Imperative FGJ with confinement and ownership (controlling access on a per instance rather than per class level) evolves into a different type system requiring a full soundness proof of all of the things that can be avoided when trying to concentrate on Generic Confinement alone.

6 Related work

Object encapsulation has been recognised as a means for addressing aliasing, security, concurrency, and memory management, with the merit of smoothly aligning with the way many object-oriented programs are designed. Two complementary threads of research have evolved. On one hand are expressive but weighty type systems based on ownership types (Clarke *et al.*, 1998; Aldrich & Chambers, 2004; Boyapati & Rinard, 2001). On the other hand are lightweight but limited systems based on confined types (Vitek & Bokowski, 2001).

Systems based on ownership types differ among themselves essentially in only one characteristic, which Clarke and Wrigstad distinguish as *shallow* vs. *deep* ownership (Clarke & Wrigstad, 2003). Deep ownership types permit only a single object as entry point to the collection of objects it owns, whereas shallow ownership types permit multiple entry points into the confined collection.

Clarke & Drossopoulou (2002) and Boyapati *et al.* (2003) describe how to exploit the useful properties of deep ownership, but there is also a general concern about whether it might be too restrictive in practice. Ownership types require additional annotations to use them, raising issues about their role in programming. Some authors argue that, with appropriate defaults, this need not be a problem in practice (Aldrich *et al.*, 2002; Boyapati *et al.*, 2003).

Confined type systems have achieved their more conservative goals while keeping the amount of annotations low. Vitek & Bokowski's (2001) original system, which had security as its application, required certain classes to be annotated as confined to indicate classes confined within the present package, and certain methods to be annotated as anonymous, to indicate that such methods do not reveal "`this`". Grothoff *et al.* (2001) show how type inference can be used to avoid the need for annotation, making a system that can provide per-package encapsulation in practical programs. Clarke *et al.* (2003) apply these ideas in the context of Enterprise Java Beans, and by exploiting special architecture specific constraints, provide per-object encapsulation without annotations or inference.

Recent work by Zhao *et al.* (2006) has formalised Vitek and Bokowski's approach to per-package confinement, with an operational semantics and a static type system based on Featherweight Java. This work also proposes and formalises a notion of generic confined types, allowing, for example, a collection to be confined or not, depending upon the specifications of the contained elements.

Our approach is essentially the opposite. Rather than starting from a language without generic types, and then adding a special form of genericity to support confinement, we start from a language with generic types (GJ, or rather its formal core FGJ) and then ensure per-package confinement. This approach has led to a simpler formal system, requiring few new concepts. We do not need to distinguish anonymous methods, because "`this`" is parameterised to record its ownership.

The key to our approach is preserving the owner in types and using visibility rules to determine where it can and cannot be used. In the approach adopted by Zhao *et al.* (2006) it is necessary to (ultimately) restrict the subtype relation and the operations which occur when using a type whose ownership isn't known. The advantage of Zhao *et al.*'s type system is that it allows a newly declared class to become confined to a package even though all of its superclasses were declared public. This is permitted as long as all the superclasses meet certain *anonymity* conditions on their methods: the type system is then required to deal with such anonymity constraints.

While FGJ+c cannot make a subclass of a manifest ownership public class confined to a package, it is fully capable of making a confined subclass of a pure class, while the pure class can also be instantiated with a public `World` owner. If a program's classes were written as pure then it is not going to be a problem. It is only in the setting of manifest public owners that such subclassing is not going to be possible.

Banerjee and Naumann prove a per-object representation independence result for Java (Banerjee & Naumann, 2004). They adopt a confinement discipline resembling ownership types, except that they apply the confinement only at the point they wish to reason about. They require that confined classes extend a special class called Rep, and that the boundary classes extend a special class called Own. Neither Rep nor Own can be forgotten from a type. Banerjee and Naumann's results demonstrate that confinement can be used to derive principles reasoning about programs.

A bit further afield, we find that the implementation of the State Monad in Haskell (Launchbury & Peyton Jones, 1995) adopts similar mechanisms. In the State Monad, a type variable is assigned to the encapsulated state, and an appropriate quantification over the type (via rank-2 polymorphism) ensures that the state doesn't escape and thus behaves correctly. Interestingly, this design resembles an encoding of existential types in terms of universal types, while Clarke's thesis formalises the confinement provided by ownership types as existential over owners (Clarke, 2002).

Finally, recent work in phantom types (Hinze, 2003; Fluet & Pucella, 2002; Leijen & Meijer, 1999), where "phantom" type parameters only purpose is to enforce well-formedness constraints, directly aligns with our approach.

7 Implementation

We have implemented an extension to the Java 5 implementation of the Java Compiler (Sun Microsystems, 2005) that we call OGJ (for "Oh! Gee! Java!" (Noble & Biddle, 2003)). OGJ programs are essentially Java 5 programs with the addition of owner parameters that can be `World`, `Class`, `Package`, or `This`. These are real Java interfaces defined in a package `ogj.ownership`. Any OGJ program will compile as long as the (blank) definitions of these interfaces are present in the class path, making OGJ backwards compatible with generic Java compilers.

On the other hand, if the program is compiled using our extension to Java 5, these four interfaces are treated specially so that any class parameterised by `World` and `Package` behaves in a similar way to FGJ+c classes parameterised by `World` and the package owner classes. Thus, a class defined as follows:

```
package my.util;

import ogj.ownership.*;

public class Link<Item, Owner extends Package> { ... }
```

will be guaranteed to have all of its instances confined within the `my.util` package as long as the code is compiled using our OGJ compiler extension. While in FGJ+c we use a separate owner class for each owner parameter corresponding to a package, OGJ makes appropriate replacements of every occurrence of parameter `Package` with an appropriate owner class. This reduces programmer load and hides the owner classes from view.

Furthermore, we have also implemented full support for per-class confinement similar to Class Universes (Müller & Poetzsch-Heffter, 1999) (objects can only be used by the *class* within which they are declared) and for per-object ownership. In this paper, we have formalised the part of OGJ that supports confinement. The OGJ compiler implements the first publicly available language that has support for both ownership and genericity (Potanin, 2005).

8 Conclusion

Generic confinement unifies two notions (genericity and confinement) that previously appeared to be unrelated. In particular, we demonstrated that the FGJ type system, combined with a series of visibility rules, is strong enough to provide a confinement invariant comparable to that of Vitek and Bokowski's Confined Types. This result shows that confinement and generic type information can be expressed within the same system and carried around the program as binding to the same parameters. We proved this is possible for confinement, and plan to extend this work to more discriminating systems such as ownership types. This may provide a lightweight route for ownership types to become applicable in practice, with genericity carrying ownership into popular object-oriented programming languages.

Acknowledgements

This work is supported in part by the Royal Society of New Zealand Marsden Fund and a Claude McCarthy Fellowship. Thank you to the anonymous reviewers for their invaluable comments that greatly improved this article and last but not least to Phil Wadler for encouraging us to submit this paper.

References

Aldrich, Jonathan, & Chambers, Craig. (2004). Ownership Domains: Separating Aliasing Policy from Mechanism. *Pages 1–25 of: Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, vol. 3086. Oslo, Norway: Springer-Verlag, Berlin, Heidelberg, Germany.

Aldrich, Jonathan, Kostadinov, Valentin, & Chambers, Craig. (2002). Alias Annotations for Program Understanding. *Pages 311–330 of: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Seattle, WA, USA: ACM Press, New York, NY, USA.

Banerjee, Anindya, & Naumann, David A. (2004). Ownership Confinement Ensures Representation Independence for Object-Oriented Programs. *Journal of the ACM (JACM)*, **52**(6), 894–960.

Boyapati, Chandrasekhar, & Rinard, Martin. (2001). A Parameterized Type System for Race-Free Java Programs. *Pages 56–69 of: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Tampa Bay, FL, USA: ACM Press, New York, NY, USA.

Boyapati, Chandrasekhar, Liskov, Barbara, & Shrira, Liuba. (2003). Ownership Types for Object Encapsulation. *Pages 213–223 of: Proceedings of ACM Symposium on Principles of Programming Languages (POPL).* New Orleans, LA, USA: ACM Press, New York, NY, USA. Invited talk by Barbara Liskov.

Clarke, Dave. (2002). *Object Ownership and Containment*. Ph.D. thesis, School of CSE, UNSW, Australia.

Clarke, Dave, & Drossopoulou, Sophia. (2002). Ownership, Encapsulation, and the Disjointness of Type and Effect. *Pages 292–310 of: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Seattle, WA, USA: ACM Press, New York, NY, USA.

Clarke, Dave, Richmond, Michael, & Noble, James. (2003). Saving the World from Bad Beans: Deployment-Time Confinement Checking. *Pages 374–387 of: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Anaheim, CA: ACM Press, New York, NY, USA.

Clarke, David, & Wrigstad, Tobias. (2003). External Uniqueness is Unique Enough. *Pages 176–200 of: Proceedings of European Conference on Object-Oriented Programming (ECOOP).* Lecture Notes in Computer Science (LNCS), vol. 2473. Darmstadt, Germany: Springer-Verlag, Berlin, Heidelberg, Germany.

Clarke, David, Potter, John, & Noble, James. (1998). Ownership Types for Flexible Alias Protection. *Pages 48–64 of: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* Vancouver, Canada: ACM Press, New York, NY, USA.

Fluet, Matthew, & Pucella, Riccardo. 2002 (Aug.). Phantom Types and Subtyping. *Pages 448–460 of: International Conference on Theoretical Computer Science (TCS).*

Hinze, Ralf. (2003). *The Fun of Programming*. Palgrave Macmillan. Editors: Jeremy Gibbons and Oege de Moor. Chap. Fun with Phantom Types, pages 245–262.

Hogg, John. (1991). Islands: Aliasing Protection in Object-Oriented Languages. *Pages 271–285 of: Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, vol. 26. Phoenix, AZ, USA: ACM Press, New York, NY, USA.

Igarashi, Atsushi, & Viroli, Mirko. (2002). On variance-based subtyping for parametric types. *Pages 441–469 of: Proceedings of European Conference on Object-Oriented Programming (ECOOP).* Malaga, Spain: Springer-Verlag, Berlin, Heidelberg, Germany. To appear in ACM Transactions on Programming Languages and Systems.

Igarashi, Atsushi, Pierce, Benjamin C., & Wadler, Philip. (2001). Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **23**(3), 396–450.

Launchbury, John, & Peyton Jones, Simon L. (1995). State in Haskell. *Lisp and Symbolic Computation*, **8**(4), 293–341.

Leijen, Daan, & Meijer, Erik. (1999). Domain-Specific Embedded Compilers. *Pages 109–122 of: Proceedings of the 2nd Conference on Domain-Specific Languages*. Berkeley, CA, USA: USENIX Association.

Müller, P., & Poetzsch-Heffter, A. (1999). *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen. Poetzsch-Heffter, A. and Meyer, J. (editors). Chap. Universes: a Type System for Controlling Representation Exposure.

Noble, James, & Biddle, Robert. 2003 (May). *Oh! Gee! Java! — Ownership Types (almost) for Free*. Tech. rept. VUW-CS-TR-03/9. School of Mathematics, Statistics, and Computer Science, Victoria University of Wellington, New Zealand.

Noble, James, Vitek, Jan, & Potter, John. (1998). Flexible Alias Protection. *Pages 158–185 of:* Jul, Eric (ed), *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science (LNCS), vol. 1445. Springer-Verlag, Berlin, Heidelberg, Germany.

Noble, James, Biddle, Robert, Tempero, Ewan, Potanin, Alex, & Clarke, Dave. (2003). Towards a Model of Encapsulation. Clarke, Dave (ed), *Proceedings of International Workshop on Aliasing, Confinement, and Ownership (IWACO)*. UU-CS-2003, no. 030. Utrecht University.

Potanin, Alex. (2005). *Ownership Generic Java Download*. `http://www.mcs.vuw.ac.nz/~alex/ogj/`.

Potanin, Alex, Noble, James, Clarke, Dave, & Biddle, Robert. (2004). Defaulting Generic Java to Ownership. *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming (FTfJP)*. Oslo, Norway: Springer-Verlag, Berlin, Heidelberg, Germany.

Sun Microsystems. (2005). *Java Development Kit*. Available at: `http://java.sun.com/j2se/`.

Vitek, Jan, & Bokowski, Boris. (2001). Confined Types in Java. *Software Practice & Experience*, **31**(6), 507–532.

Zhao, Tian, Palsberg, Jens, & Vitek, Jan. (2006). Type-Based Confinement. *Journal of Functional Programming*, **16**(1), 83–128.