Featherweight Generic Ownership

Alex Potanin, James Noble, Dave Clarke[1], and Robert Biddle[2]
{alex, kjx}@mcs.vuw.ac.nz, dave@cwi.nl, and
robert_biddle@carleton.ca

Victoria University of Wellington, New Zealand
[1] Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands
[2] Human-Oriented Technology Laboratory, Carleton University, Canada

Abstract. Object ownership is an approach to controlling aliasing in programming languages. Proposals for adding ownership to programming languages do not directly support type genericity. We present Featherweight Generic Ownership — the first system to unify ownership and type polymorphism. Our type system extends Featherweight Generic Java with locations to support ownership and confinement invariants, as well as having full soundness proof. We hope that our work will help bring full support for object encapsulation to the mainstream programming world.

1 Introduction

Object ownership ensures that objects cannot be leaked beyond an object or collection of objects which *own* them. There are two main approaches to object encapsulation in the literature: enforcing coding conventions within an existing programming language, or significantly modifying a language to allow ownership support. The first approach is taken by Islands [12] and various kinds of Confined Types [21, 7]. Programs must be written to follow a set of specific conventions that can be checked to see if they provide containment guarantees [11]. The soundness of this approach has been recently proven [22]. Support for generics is added on top of such collections of restrictions for enforcing encapsulation [22].

The second approach is taken by languages such as Joe, Universes, Alias Java, and Safe Java [6, 15, 1, 4]. Ownership parameterisation is added to the syntax and expressed explicitly within the type system of these languages. All of these type systems are distinct: they all employ ownership parameterisation, but none has support for type genericity.

This paper continues efforts to provide effective object encapsulation within practical programming languages. We present Featherweight Generic Ownership (FGO) that uses a single parameter space to carry *both* generic type and ownership type information. We build on the ideas of Featherweight Generic Confinement [18] (that showed a way to achieve confinement guarantees, but not object ownership) to demonstrate a full support for shallow ownership and type genericity in a simple and unified manner. We show that starting with a type system with full support for type polymorphism allows us to achieve ownership guarantees more straightforwardly than starting with non type polymorphic type system.

Outline. We first present the ideas behind FGO in section 2. In section 3 we present FGO type system and delve into the details that allow it to support ownership and confinement by carrying the ownership information as part of a type. We then state and prove the soundness of FGO, as well as ownership and confinement invariants in section 4. Finally, we conclude our presentation in section 5.

2 FGO Basics

The key idea behind FGO is to use generic type parameters to carry ownership information as well as type information. Following the approach of Ownership Types [8], we require that every pure FGO class has at least one type parameter to carry this ownership information. We use the *last* type parameter to record an object's owner, to promote upwards compatibility and because our implementation will allow ownership parameters to be elided [17]. All FGO classes descend from a new parameterised root `Object<O>` that has just one parameter; all of its subclasses must invariantly preserve this owner, either by preserving the parameter, or by using manifest ownership.

The following declaration of a class called `Main` shows that the class is declared with an owner parameter `Owner` that is bound to `Object`'s owner parameter.

```
class m.Main<Owner extends World> extends Object<Owner> {
  m.Main() { super(); }
  p.OwnedStack<Object<World>, World> publicStack() {
    return new p.OwnedStack<Object<World>, World>;
  }
  p.OwnedStack<m.Main<World>, M> confinedStack() {
    return new p.OwnedStack<m.Main<World>, M>;
  }
  p.OwnedStack<m.Main<M>, This> privateStack() {
    return new p.OwnedStack<m.Main<M>, This>;
  }
}
```

Note that all class names are prefixed by a package identifier, thus `m.Main`. This is a convention to indicate the package within which each class is defined. Note also that classes which extend class `World` are used to indicate ownership. We use lower case letters such as `m` to denote packages and upper case letters such as `M` to denote owner classes corresponding to these packages. Finally we use an owner class `This` to mark instances that are going to be private to a particular instance of a class.

Within the `Main` class, three methods return `OwnedStack` objects; one of these is public, another one is confined to package `m`, and the last one is private to a particular instance of class `Main`. The public stack stores `Object<World>` instances that are accessible from anywhere (`World` is instantiating `Object`'s owner parameter); the confined stack stores `Main` instances that are also globally accessible, however the confined stack has owner `M`, meaning that it is only accessible within package `m`; the private stack stores instances of `Main` accessible inside package `m` only, while the actual stack is only accessible by the particular instance of `Main` that created it. In each case, the stack's second parameter describes its ownership. These three stacks illustrate that FGO provides both *type polymorphism* (the stacks hold different item types) and *ownership polymorphism* (the stacks belong in different protection contexts).

The version of ownership supported by FGO is usually referred to as *shallow ownership* [9]: the access to objects is controlled without enforcing an object graph property such as *owners as dominators* [5]. This is similar to the kind of ownership supported by Alias Java [1]. `This` owner class marks types that are accessible only by a particular object.

At the same time, FGO supports package-level ownership, more commonly known as *confinement* [21, 22]. Package owner classes (e.g. `M` in the example above) are used to mark types that are confined to particular packages.

FGO supports both confinement and shallow ownership, and our current work is extending it to also support *deep ownership* — the kind of ownership enforcing owner's position in program's object graph as dominators. Deep ownership is provided (with a variety of extensions to make it less restrictive and thus more usable in practice) by languages such as Joe [6] and Safe Java [3].

We adopt the concept of *manifest ownership* [5] to allow classes without explicit owner type parameters. We distinguish two different categories of FGO classes by calling the classes with explicit owner parameters *pure FGO classes*, and the ones with an implicit owner parameter — *manifest FGO classes*. A manifest FGO class does not have an explicit owner parameter, rather the class's owner is fixed and all the objects of that class have the same owner. To demonstrate manifest ownership, consider the following alternative to creating a public stack:

```
class p.PublicStack extends p.OwnedStack<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 `PublicStack` requires no owner type parameter.

Manifest ownership allows us to fit all of the FGJ's classes into FGO class hierarchy by simply making FGJ's root class into an FGO manifest class [1]:

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

With this definition `Object` and every FGJ class under it has a default owner parameter `World` (thus making them publically accessible). For example, it becomes possible to have the following familiar declaration of a public `Stack` object, which is indistinguishable from that of FGJ:

```
class Stack extends Object { ... }
```

The important difference is that in FGO, class `Stack` has an owner `World` coming from `Object`'s super class `Object<World>`.

[1] To avoid name conflict, FGO can choose a different name for its root, such as `OObject<World>` for *owned* object.

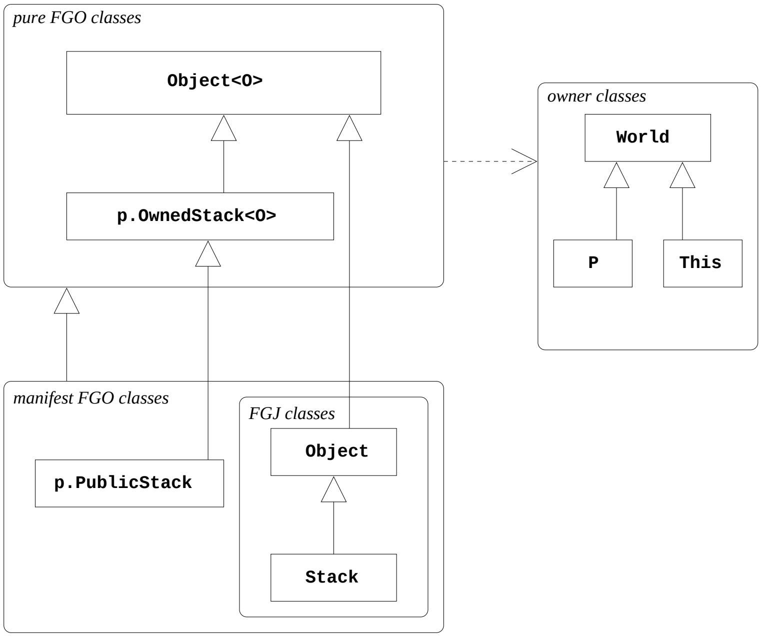

Fig. 1. FGO Classes and Owner Classes. *Pure* FGO classes have an explicit owner type parameter. *Manifest* FGO classes have an owner fixed when subclassing a pure FGO class. *Owner* classes lie outside the FGO class hierarchy because they cannot be instantiated in FGO programs; pure FGO classes use them to bind their owner type parameters (as shown by the dashed arrow on the diagram).

Figure 1 shows the relationship between owner classes and program classes in FGO. Owner classes inherit from the class `World`, and there is one owner class corresponding to each FGO package, as well as a special owner class `This` described above. The owner class hierarchy lies outside the `Object<World>` hierarchy of pure and manifest FGO classes. Manifest FGO classes have an owner class corresponding to their owner, which is not written out explicitly but rather is found among its superclasses in the FGO class hierarchy. Vanilla FGJ classes form a subset of manifest FGO classes.

3 FGO Type System

Featherweight Generic Ownership (FGO) builds on Featherweight Generic Java (a type system describing a functional subset of Java [13]), Featherweight Generic Confinement (a type system describing a way to provide confinement and type polymorphism in a unified manner, utilising manifest ownership [19]), and Featherweight Domains Java (a type system describing a way to support ownership domains with locations and store to describe the references [2]). FGO presents a provably sound type system providing ownership, confinement, and type polymorphism in a unified manner. FGO also

Syntax:

$\mathtt{T} ::= \mathtt{X} \mid \mathtt{N}$	$\mathtt{e} ::= \mathtt{e_s} \mid l \mid l > \mathtt{e}$
$\mathtt{N} ::= \mathtt{C{<}\overline{T}{>}}$	$\mathtt{S} ::= \{l\ :\ \mathtt{N}(\overline{v})\}$
$\mathtt{L} ::= \mathtt{class\ C{<}\overline{X} \lhd \overline{N}{>} \lhd\ N\{\overline{T}\ \overline{f}; K\ \overline{M}\}}$	$v,\ l \in \mathit{locations}$
$\mathtt{K} ::= \mathtt{C}(\overline{\mathtt{T_{cp}}\ \mathtt{f_{cp}}})\ \{\ \mathtt{super}(\overline{\mathtt{f_{cp}}});\ \mathtt{this.\overline{f}} = \mathit{initfield}_{\overline{\mathtt{T}}}(\overline{\mathtt{T_{cp}}\ \mathtt{f_{cp}}})\ \}$	$\Gamma ::= \{\mathtt{x}\ :\ \mathtt{T}\}$
$\mathtt{M} ::= \mathtt{{<}\overline{X} \lhd \overline{N}{>}T\ m(\overline{T}\ \overline{x})\{return\ e;\}}$	$\Sigma ::= \{l\ :\ \mathtt{T}\}$
$\mathtt{e_s} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m{<}\overline{T}{>}(\overline{e})} \mid \mathtt{new\ N(\overline{e})} \mid \mathtt{(\overline{N})\ e}$	$\Delta ::= \{\mathtt{X}\ :\ \mathtt{N}\}$

Functions:

$initfield_{\mathtt{T}}(\overline{\mathtt{T_{cp}}\ \mathtt{f_{cp}}})$	Initialises a field of type T using constructor parameters.
$\pi_{\mathtt{C}}$	Package owner class corresponding to class C.
$this_l(\mathtt{e})$	Provides replacement for This keyword in method or field type which are executed on expression e while at location *l*.
$owner_{\Delta}(\mathtt{T})$	Determines owner of type T.
$visible_{\Delta}(\mathtt{O}, \mathtt{C})$	Owner O is visible in class C.
$visible_{\Delta}(\mathtt{T}, \mathtt{C})$	Type T is visible in class C.

Judgements:

$\Delta; \Sigma \vdash \mathtt{T}\ \mathsf{OK}$	Type T is OK.
$\Delta; \Sigma \vdash \mathtt{T} \mathrel{\mathtt{<:}} \mathtt{U}$	Type T is a subtype of type U.
$\Delta; \Gamma; \Sigma; P \vdash \mathtt{e} : \mathtt{T}$	Expression e is well typed.
$\Delta; \Gamma; \Sigma; P \vdash \mathit{visible}(\mathtt{e})$	Expression e is visible with respect to permission *P* (a type at static time).
$\Delta \vdash \Sigma\ \mathsf{OK}$ and $\Delta; \Sigma \vdash S$	Store (heap) is well-formed.
$\mathtt{{<}\overline{Y} \lhd \overline{P}{>}\ T\ m(\overline{T}\ \overline{x})\{return\ e_0;\}\ FGO\ IN\ C{<}\overline{X} \lhd \overline{N}{>}}$	Method m definition is OK.
$\mathtt{class\ C{<}\overline{X} \lhd \overline{N}{>} \lhd N\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}\ FGO}$	Class C definition is OK.

Fig. 2. FGO Syntax, Functions, and Judgements

builds on the confinement and genericity ideas of Confined Featherweight Java [23], ownership typing of Joe [6], the concept of phantom types [10], and utilising type polymorphism to its full potential [14].

In this section we briefly present the important points of FGO type system as well as revealing a full set of rules in the figures. We assume that the reader is familiar with Featherweight Generic Java (FGJ) [13].

3.1 Syntax and Judgements

Figure 2 presents FGO syntax, useful functions, and judgements used throughout the type system. The syntax is very close to FGJ, except for locations (l, v) and store (S). Every time an object is created, a unique location l is associated with it and stored in store S. This location is then used to track which object's field or method is being accessed during the execution. Additionally, we had to change the constructor syntax to allow initialisation of fields inside the class, rather than only outside. This allows field's owners to be private to the instance creating them.

FGO also introduces a set of *visibility* functions and rules that allow us to look at the types involved and decide based on their *owner* information whether they are "visible" with respect to each other. $\pi_{\mathtt{C}}$ gives an owner class corresponding to a package where

Initialise Field:

$$\begin{aligned} initfield_{\mathtt{T}}(\overline{\mathtt{T}_{\mathtt{cp}}\ \mathtt{f}_{\mathtt{cp}}}) &= \mathtt{f}', \qquad \mathtt{T}'\ \mathtt{f}' \in \overline{\mathtt{T}_{\mathtt{cp}}\ \mathtt{f}_{\mathtt{cp}}}\ and\ \mathtt{T}' = \mathtt{T} \\ &= \mathtt{new}\ \mathtt{N}(\overline{\mathtt{f}}),\ \overline{\mathtt{f}} \subseteq \overline{\mathtt{f}_{\mathtt{cp}}}\ and\ \mathtt{N} = \mathtt{T} \end{aligned} \qquad \text{(FGO-Init-Field)}$$

Constructor Parameters Lookup:

$$cparams(\mathtt{Object<O>}) = \bullet \qquad \text{(CP-Object)}$$

$$\frac{\begin{array}{c}\mathtt{K} ::= \mathtt{C}(\overline{\mathtt{T}_{\mathtt{cp}}}\ \overline{\mathtt{f}_{\mathtt{cp}}})\ \{\ \mathtt{super}(\overline{\mathtt{f}_{\mathtt{cp}}});\ \mathtt{this}.\overline{\mathtt{f}} = initfield_{\overline{\mathtt{T}}}(\overline{\mathtt{f}_{\mathtt{cp}}})\ \} \\ \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad cparams([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}\end{array}}{cparams(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{T}_{\mathtt{cp}}}\ \overline{\mathtt{f}_{\mathtt{cp}}}} \qquad \text{(CP-Class)}$$

Owner Lookup:

$$\begin{aligned} owner_{\Delta}(\mathtt{X}) &= owner_{\Delta}(bound_{\Delta}(\mathtt{X})) \\ owner_{\Delta}(\mathtt{C<}\overline{\mathtt{T}},\ \mathtt{O>}) &= \mathtt{O} \\ owner_{\Delta}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) &= owner_{\Delta}(\mathtt{N}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]), \text{ where} \\ & \quad CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\{\overline{\mathtt{T}'}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \end{aligned} \qquad \text{(FGO-Owner)}$$

Fig. 3. FGO-Specific Auxiliary Functions

class `C` is declared. The function $this$ allows us to preserve unique owners per each object location to be able to ensure that objects with `This` owner are private to a correct instance location.

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 `This`, as well as the $\pi_{\mathtt{C}}$ owner classes corresponding to packages. Pure FGO 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:

$$\begin{aligned} \mathtt{N}_{pure} &::= \mathtt{C<}\overline{\mathtt{T}},\ \mathtt{O>} \\ \mathtt{L}_{pure} &::= \mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}},\ \mathtt{OVar} \triangleleft \mathtt{OCon>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}. \end{aligned}$$

We use *CT* (class table) to denote a mapping from class names `C` to class declarations `L` and *P* (permission) to denote a current location to be able to correctly type `this`.

3.2 Auxiliary Functions

The extra FGO auxiliary functions are the *initfield*, *cparams*, and *owner* functions shown in figure 3. The other auxiliary functions are presented in figure 4 are the same as those of FGJ, given the new root class.

The *initfield* function is part of the FGO syntax to allow two kinds of field initialisation — either using the explicit constructor parameter (as is in FGJ) or by creating a

Bound of Type:

$$bound_{\Delta}(\texttt{X}) = \Delta(\texttt{X})$$
$$bound_{\Delta}(\texttt{N}) = \texttt{N}$$

Subclassing:

$$\texttt{C} \unlhd \texttt{C} \qquad \frac{\texttt{C} \unlhd \texttt{D} \quad \texttt{D} \unlhd \texttt{E}}{\texttt{C} \unlhd \texttt{E}} \qquad \frac{\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{D<}\overline{\texttt{T}}\texttt{>}\{\ldots\}}{\texttt{C} \unlhd \texttt{D}}$$

Field Lookup:

$$fields(\texttt{Object<O>}) \;=\; \bullet \qquad \text{(F-OBJECT)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad fields([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{fields(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{g}},\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \overline{\texttt{f}}} \qquad \text{(F-CLASS)}$$

Method Type Lookup:

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \overline{\texttt{N}}\ \{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \\ \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{> U m(}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)}\{\ \texttt{return e};\} \in \overline{\texttt{M}}\end{array}}{mtype(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}} \to \texttt{U})} \qquad \text{(MT-CLASS)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \lhd \texttt{N>} \lhd \overline{\texttt{N}}\ \{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{mtype(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mtype(\texttt{m},\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})} \qquad \text{(MT-SUPER)}$$

Method Body Lookup:

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \overline{\texttt{N}}\ \{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \\ \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{> U m(}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)}\{\ \texttt{return}\ \texttt{e}_0;\} \in \overline{\texttt{M}}\end{array}}{mbody(\texttt{m<}\overline{\texttt{V}}\texttt{>},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{x}}.[\overline{\texttt{T}}/\overline{\texttt{X}},\ \overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{e}_0} \qquad \text{(MB-CLASS)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \lhd \texttt{N>} \lhd \overline{\texttt{N}}\ \{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{mbody(\texttt{m<}\overline{\texttt{V}}\texttt{>},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mbody(\texttt{m<}\overline{\texttt{V}}\texttt{>},\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})} \qquad \text{(MB-SUPER)}$$

Constructor Parameters Lookup:

$$cparams(\texttt{Object<O>}) \;=\; \bullet \qquad \text{(CP-OBJECT)}$$

$$\frac{\begin{array}{c}\texttt{K} ::= \texttt{C}(\overline{\texttt{T}_{\texttt{cp}}}\ \overline{\texttt{f}_{\texttt{cp}}})\ \{\ \texttt{super}(\overline{\texttt{f}_{\texttt{cp}}});\ \texttt{this}.\overline{\texttt{f}} = initfield_{\overline{\texttt{T}}}(\overline{\texttt{f}_{\texttt{cp}}})\ \} \\ \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad cparams([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}) = \overline{\texttt{U}}\ \overline{\texttt{g}}\end{array}}{cparams(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{g}},\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{T}_{\texttt{cp}}}\ \overline{\texttt{f}_{\texttt{cp}}}} \qquad \text{(CP-CLASS)}$$

Fig. 4. Other FGO Auxiliary Functions

new instance of an object stored in a field inside the constructor, using the constructor parameters (marked $_{\texttt{cp}}$) if necessary to initialise a new object. This allows FGO classes to initialise fields with owner `This` which are inaccessible outside the object. The function *cparams* works in the same way as the field lookup function, except that it looks up constructor parameters for a given class.

The *owner* function is given a type for which it establishes the nonvariable bound and checks if the corresponding class type instance has an owner. In case of a manifest class, a superclass making the owner fixed is found by traversing the class hierarchy. This owner is then returned by the function.

3.3 *This* Function

The *this* function, presented in figure 5, forms the foundation for object ownership in FGO. It is used both to verify the validity of class declarations inside a FGO program and to ensure that during reduction, every `This` owner is appropriately converted into location-unique $\mathtt{This}_l$ owner class.

This Function:

$$\begin{array}{ll} \mathit{this}_{\mathtt{C}}(\mathtt{this}) & = \mathtt{This} \\ \mathit{this}_{l}(l') & = \mathtt{This}_{l} \quad \text{(FGO-THIS)} \\ \mathit{this}_{P}(\ldots) & = \perp \end{array}$$

Fig. 5. FGO This Function

The first use of *this* is during the validation of class declaration. Every expression containing a method call or a field access is checked to see if any of the types involved contain `This` as an owner class. If they do, then the method call or field access is only allowed on `this`, not on another object. At this time, the permission P given to *this* is the type of the current FGO class being validated.

The second use of *this* is during reduction, when locations become part of the typing rules. In this case, the permission P given to *this* is the current location l. Any occurrence of the owner class `This` is now replaced by a more appropriate location-specific $\mathtt{This}_l$, just like in reduction rule R-METHOD.

In either case, if the use of owner class `This` violates the *this* constraint, the expression type becomes undefined ($\perp$). FGO type soundness guarantees that any validated FGO class will not incur an invalid access to a location-specific instance (marked by $\mathtt{This}_l$ owner) during reduction.

3.4 Subtyping and Type Well-formedness

Figure 6 shows FGO's subtyping rules. They are generally taken verbatim from FGJ, except for the addition of subtyping for owner classes.

`World` forms the top of owner class hierarchy, which any package owner class extends directly from it. `This` also lies directly below `World`, while location-specific version $\mathtt{This}_l$ extends `This`. The latter is important for the subject reduction proof. Finally, FGO's type well-formedness rules shown in figure 7 are the same as FGJ, except that the root of class hierarchy is parameterised.

3.5 Visibility

Another important addition that FGO makes to FGJ are a set of visibility rules similar to the ones used by Featherweight Generic Confinement [19]. While a subset of our visibility rules for terms is similar to the ones used by ConfinedFJ [23], the owner and type visibility form the essential foundation of FGO.

Subtyping:

$$\frac{}{\Delta;\Sigma \vdash \texttt{T} \mathrel{\texttt{<:}} \texttt{T}} \qquad \frac{}{\Delta;\Sigma \vdash \texttt{X} \mathrel{\texttt{<:}} \Delta(\texttt{X})} \qquad \text{(S-REFL) and (S-VAR)}$$

$$\frac{\Delta;\Sigma \vdash \texttt{S} \mathrel{\texttt{<:}} \texttt{T} \qquad \Delta;\Sigma \vdash \texttt{T} \mathrel{\texttt{<:}} \texttt{U}}{\Delta;\Sigma \vdash \texttt{S} \mathrel{\texttt{<:}} \texttt{U}} \qquad \text{(S-TRANS)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \{\ldots\}}{\Delta;\Sigma \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \mathrel{\texttt{<:}} [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}} \qquad \text{(S-CLASS)}$$

$$\frac{}{\Delta;\Sigma \vdash \texttt{World} \mathrel{\texttt{<:}} \texttt{World}} \qquad \frac{}{\Delta;\Sigma \vdash \texttt{This} \mathrel{\texttt{<:}} \texttt{World}} \qquad \text{(S-OWNER)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \{\ldots\}}{\Delta;\Sigma \vdash \pi_{\texttt{C}} \mathrel{\texttt{<:}} \texttt{World}} \qquad \frac{l \in dom(\Sigma)}{\Delta;\Sigma \vdash \texttt{This}_l \mathrel{\texttt{<:}} \texttt{This}}$$

Valid Downcast:

$$\frac{dcast(\texttt{C},\ \texttt{D}) \qquad dcast(\texttt{D},\ \texttt{E})}{dcast(\texttt{C},\ \texttt{E})} \qquad \frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{D<}\overline{\texttt{T}}\texttt{>} \{\ldots\} \\ \overline{\texttt{X}} = FV(\overline{\texttt{T}})\end{array}}{dcast(\texttt{C},\ \texttt{D})}$$

$FV(\overline{\texttt{T}})$ is the set of type variables in $\overline{\texttt{T}}$.

Valid Method Overriding:

$$\frac{\begin{array}{c} mtype(\texttt{m},\ \texttt{N}) = \texttt{<}\overline{\texttt{Z}} \triangleleft \overline{\texttt{Q}}\texttt{>}\overline{\texttt{U}} \rightarrow \texttt{U}_0 \\ \Rightarrow\ \overline{\texttt{P}}, \overline{\texttt{T}} = [\overline{\texttt{Y}}/\overline{\texttt{Z}}](\overline{\texttt{Q}}, \overline{\texttt{U}}) \text{ and } \overline{\texttt{Y}} \mathrel{\texttt{<:}} \overline{\texttt{P}} \vdash \texttt{T}_0 \mathrel{\texttt{<:}} [\overline{\texttt{Y}}/\overline{\texttt{Z}}]\texttt{U}_0 \end{array}}{override(\texttt{m},\ \texttt{N},\ \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0)}$$

Fig. 6. FGO Subtyping Rules

Well-Formed Types:

$$\frac{\texttt{X} \in dom(\Delta)}{\Delta;\Sigma \vdash \texttt{X}\ \texttt{OK}} \qquad \frac{\Delta;\Sigma \vdash \texttt{O} \mathrel{\texttt{<:}} \texttt{World}}{\Delta;\Sigma \vdash \texttt{Object<O>}\ \texttt{OK}} \qquad \text{(WF-VAR) and (WF-OBJECT)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \{\ldots\} \\ \Delta;\Sigma \vdash \texttt{N} \mathrel{\texttt{<:}} \texttt{Object<O>} \qquad \Delta;\Sigma \vdash \texttt{O} \mathrel{\texttt{<:}} \texttt{World} \\ \Delta;\Sigma \vdash \overline{\texttt{T}}\ \texttt{OK} \qquad \Delta;\Sigma \vdash \overline{\texttt{T}} \mathrel{\texttt{<:}} [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}}\end{array}}{\Delta;\Sigma \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ \texttt{OK}} \qquad \text{(WF-CLASS)}$$

Fig. 7. FGO Type Well-Formedness Rules

Figure 8 shows owner visibility rule that checks if an owner `O` is visible inside a nonvariable type `C`. This is the case if the owner is `World`, belongs to the same package as `C`, or is an owner used by one of the generic parameters used when instantiating the type of `C`. The last part is important, since by supplying a class with a generic parameter with a completely unrelated owner context, we *allow* the class to have access to that unrelated owner context so that it can use its own type parameter. This, for example,

Owner Visibility:

$$\mathit{visible}_{\triangle}(\mathtt{O},\ \mathtt{C}) = \mathtt{O} \in \mathit{owners}(\mathtt{C}) \cup \{\mathtt{This}, \pi_{\mathtt{C}}, \mathtt{World}\} \qquad \text{(V-OWNER)}$$

$$\text{where } \mathit{owners}(\mathtt{C}) = \begin{cases} \{\mathit{owner}_{\triangle}(\mathtt{N}') \mid \mathtt{N}' \in \overline{\mathtt{N}}, \mathtt{N}\}, \\ \qquad \text{if } \mathit{CT}(\mathtt{C}) = \mathtt{class\ C{<}}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{{>}} \triangleleft \mathtt{N}\{\ldots\} \\ \{\mathtt{OVar}\} \cup \{\mathit{owner}_{\triangle}(\mathtt{N}') \mid \mathtt{N}' \in \overline{\mathtt{N}}\}, \\ \qquad \text{if } \mathit{CT}(\mathtt{C}) = \mathtt{class\ C{<}}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}},\ \mathtt{OVar} \triangleleft \mathtt{OCon{>}} \triangleleft \mathtt{N}\{\ldots\} \end{cases}$$

Fig. 8. FGO Owner Visibility

can allow a type polymorphic class to have private access to more than one package. This issue was discussed in ConfinedFJ [23].

Finally, we allow a complete visibility of `This` owner on its own and rely on *this* function described in an earlier subsection to stop illegal uses of `This`.

FGO visibility rules are shown in figure 9. Type visibility simply checks the owner of a given type for visibility. Term visibility recursively checks all the types involved in the five possible expressions of FGO to make sure that they are visible in a given nonvariable type `C`. Since these checks are performed on class declarations, locations are not present in these expressions.

Type Visibility:

$$\mathit{visible}_{\triangle}(\mathtt{T},\ \mathtt{C}) = \mathit{visible}_{\triangle}(\mathit{owner}_{\triangle}(\mathtt{T}),\ \mathtt{C}) \qquad \text{(V-TYPE)}$$

Term Visibility:

$$\frac{\Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathtt{x} : \mathtt{T} \qquad \mathit{visible}_{\triangle}(\mathtt{T},\ \mathtt{C})}{\Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathit{visible}(\mathtt{x})} \qquad \text{(V-VAR)}$$

$$\frac{\Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathit{visible}(\mathtt{e}) \qquad \Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathtt{e.f_i} : \mathtt{T} \qquad \mathit{visible}_{\triangle}(\mathtt{T},\ \mathtt{C})}{\Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathit{visible}(\mathtt{e.f_i})} \qquad \text{(V-FIELD)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{T} \qquad \mathit{visible}_{\triangle}(\mathtt{T},\ \mathtt{C}) \\ \Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathit{visible}(\mathtt{e}) \qquad \Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathit{visible}(\overline{\mathtt{e}})\end{array}}{\Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathit{visible}(\mathtt{e.m}(\overline{\mathtt{e}}))} \qquad \text{(V-METHOD)}$$

$$\frac{\Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathit{visible}(\overline{\mathtt{e}}) \qquad \mathit{visible}_{\triangle}(\mathtt{N},\ \mathtt{C})}{\Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathit{visible}(\mathtt{new\ N}(\overline{\mathtt{e}}))} \qquad \text{(V-NEW)}$$

$$\frac{\Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathit{visible}(\mathtt{e}) \qquad \mathit{visible}_{\triangle}(\mathtt{N},\ \mathtt{C})}{\Delta; \Gamma; \Sigma; \mathtt{C} \vdash \mathit{visible}((\mathtt{N})\ \mathtt{e})} \qquad \text{(V-CAST)}$$

Fig. 9. FGO Visibility Rules

The expression typing rules are standard FGJ rules with support for locations, similar to the rules of Featherweight Domains Java [2].

Expression Typing:

$$\frac{}{\Delta;\Gamma;\Sigma;P \vdash \mathtt{x}\ :\Gamma(\mathtt{x})} \qquad \frac{}{\Delta;\Gamma;\Sigma;P \vdash l : \Sigma(l)}$$

(T-VAR) AND (T-LOC)

$$\frac{\begin{array}{c}\Delta;\Gamma;\Sigma;P \vdash \mathtt{e}_0\ :\ \mathtt{T}_0 \\ \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\,\overline{\mathtt{f}} \qquad \mathtt{T} = [\mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\mathtt{T}_i \qquad \Delta;\Sigma \vdash \mathtt{T}\ \mathsf{OK}\end{array}}{\Delta;\Gamma;\Sigma;P \vdash \mathtt{e}_0.\mathtt{f}_i\ :\ \mathtt{T}}$$

(T-FIELD)

$$\frac{\begin{array}{c}\Delta;\Gamma;\Sigma;P \vdash \mathtt{e}_0\ :\ \mathtt{T}_0 \qquad \Delta;\Sigma \vdash \mathtt{T}\ \mathsf{OK} \qquad \Delta;\Sigma \vdash \overline{\mathtt{V}}\ \mathsf{OK} \\ \mathit{mtype}(\mathtt{m},\ \mathit{bound}_\Delta(\mathtt{T}_0)) = \mathtt{<}\overline{\mathtt{Y}} \vartriangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U} \\ \Delta;\Sigma \vdash \overline{\mathtt{V}}\,\mathtt{<:}\,[\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ \mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\overline{\mathtt{P}} \qquad \Delta;\Gamma;\Sigma;P \vdash \overline{\mathtt{e}}\ :\ \overline{\mathtt{S}} \\ \Delta;\Sigma \vdash \overline{\mathtt{S}}\,\mathtt{<:}\,[\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ \mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\overline{\mathtt{U}} \qquad \mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ \mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\mathtt{U}\end{array}}{\Delta;\Gamma;\Sigma;P \vdash \mathtt{e}_0.\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}(\overline{\mathtt{e}})\ :\ \mathtt{T}}$$

(T-METHOD)

$$\frac{\Delta;\Sigma \vdash \mathtt{N}\ \mathsf{OK} \quad \mathit{cparams}(\mathtt{N}) = \overline{\mathtt{T}}\,\overline{\mathtt{f}} \quad \Delta;\Gamma;\Sigma;P \vdash \overline{\mathtt{e}}\ :\ \overline{\mathtt{S}} \quad \Delta;\Sigma \vdash \overline{\mathtt{S}}\,\mathtt{<:}\,\overline{\mathtt{T}}}{\Delta;\Gamma;\Sigma;P \vdash \mathtt{new}\ \mathtt{N}(\overline{\mathtt{e}})\ :\ \mathtt{N}}$$

(T-NEW)

$$\frac{\Delta;\Sigma \vdash \mathtt{N}\ \mathsf{OK} \qquad \Delta;\Gamma;\Sigma;P \vdash \mathtt{e}_0\ :\ \mathtt{T}_0 \qquad \Delta;\Sigma \vdash \mathit{bound}_\Delta(\mathtt{T}_0)\,\mathtt{<:}\,\mathtt{N}}{\Delta;\Gamma;\Sigma;P \vdash (\mathtt{N})\mathtt{e}_0\ :\ \mathtt{N}}$$

(T-UCAST)

$$\frac{\begin{array}{c}\Delta;\Sigma \vdash \mathtt{N}\ \mathsf{OK} \qquad \Delta;\Gamma;\Sigma;P \vdash \mathtt{e}_0\ :\ \mathtt{T}_0 \qquad \Delta;\Sigma \vdash \mathtt{N}\,\mathtt{<:}\,\mathit{bound}_\Delta(\mathtt{T}_0) \\ \mathtt{N} = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \qquad \mathit{bound}_\Delta(\mathtt{T}_0) = \mathtt{D<}\overline{\mathtt{U}}\mathtt{>} \qquad \mathit{dcast}(\mathtt{C},\ \mathtt{D})\end{array}}{\Delta;\Gamma;\Sigma;P \vdash (\mathtt{N})\mathtt{e}_0\ :\ \mathtt{N}}$$

(T-DCAST)

$$\frac{\begin{array}{c}\Delta;\Sigma \vdash \mathtt{N}\ \mathsf{OK} \qquad \Delta;\Gamma;\Sigma;P \vdash \mathtt{e}_0\ :\ \mathtt{T}_0 \qquad \mathtt{N} = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \\ \mathit{bound}_\Delta(\mathtt{T}_0) = \mathtt{D<}\overline{\mathtt{U}}\mathtt{>} \qquad \mathtt{C} \ntrianglelefteq \mathtt{D} \qquad \mathtt{D} \ntrianglelefteq \mathtt{C} \qquad \mathit{stupid\ warning}\end{array}}{\Delta;\Gamma;\Sigma;P \vdash (\mathtt{N})\mathtt{e}_0\ :\ \mathtt{N}}$$

(T-SCAST)

$$\frac{\Delta;\Gamma;\Sigma;l \vdash \mathtt{e}\ :\ \mathtt{T}}{\Delta;\Gamma;\Sigma;P \vdash l\ \mathtt{>}\ \mathtt{e} :\ \mathtt{T}}$$

(T-CONTEXT)

Fig. 10. FGO Expression Typing Rules

3.6 Store, Method, and Class Typing

The store typing rules is standard and is similar to other type systems using locations [6, 2]. The mapping Σ contains the types for each location and the FGO-STORE-WF rule in figure 11 ensures that every one of the types is well-formed. The mapping S contains the type instantiations with locations for each field. The main FGO-STORE rule ensures that not only the types are well-formed, but also that each field location is valid and is a correct subtype of the declared field type. It is interesting to note that our type system doesn't have any explicit containment constraints in the store rule — the benefit of ownership information being part of the type — the subtyping ensures that all the containment constraints are not broken.

Finally, the class and method declarations are checked to contain well-formed, visible types and expressions. There are rules for pure and manifest FGO classes distin-

Store Well-Formedness:

$$\frac{\forall l \in dom(\Sigma) : \Delta; \Sigma \vdash\ \Sigma(l)\ \mathsf{OK}}{\Delta \vdash \Sigma\ \mathsf{OK}} \qquad \text{(FGO-}\textsc{Store-WF}\text{)}$$

Store Typing:

$$\frac{\begin{array}{c} dom(S) = dom(\Sigma) \qquad S[l] = \mathtt{N}(\overline{v}) \Longleftrightarrow \Sigma[l] = \mathtt{N} \qquad \Delta \vdash \Sigma\ \mathsf{OK} \\ (S[l,i] = l') \wedge (fields(\Sigma[l]) = \overline{\mathtt{T}}\ \mathtt{f}) \Longrightarrow \Delta; \Sigma \vdash \Sigma[l']\ \mathtt{<:}\ \mathtt{T}_i \\ (S[l,\ i] = l') \Longrightarrow \Delta; \Sigma \vdash\ \Sigma(l')\ \mathsf{OK} \end{array}}{\Delta; \Sigma \vdash S} \qquad \text{(FGO-}\textsc{Store}\text{)}$$

Fig. 11. FGO Store Typing Rules

Method Typing:

$$\frac{\begin{array}{c} \Delta; \Sigma \vdash \overline{\mathtt{X}}\ \mathtt{<:}\ \overline{\mathtt{N}},\ \overline{\mathtt{Y}}\ \mathtt{<:}\ \overline{\mathtt{P}} \Rightarrow \overline{\mathtt{T}}, \mathtt{T}, \overline{\mathtt{N}}, \overline{\mathtt{P}}\ \mathsf{OK} \qquad \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}\ \triangleleft\ \mathtt{N}\ \{\ldots\} \\ visible_{\Delta}(\overline{\mathtt{T}},\ \mathtt{C}) \qquad visible_{\Delta}(\mathtt{T},\ \mathtt{C}) \qquad visible_{\Delta}(\overline{\mathtt{P}},\ \mathtt{C}) \\ \Delta,\ \overline{\mathtt{x}}\ :\overline{\mathtt{T}},\ \mathtt{this}\ :\ \mathtt{C<}\overline{\mathtt{X}}\mathtt{>}\ ; \Gamma; \Sigma; \mathtt{C<}\overline{\mathtt{X}}\ \triangleleft\ \overline{\mathtt{N}}\mathtt{>} \vdash\ visible(\mathtt{e}_0) \\ \Delta,\ \overline{\mathtt{x}}\ :\overline{\mathtt{T}},\ \mathtt{this}\ :\ \mathtt{C<}\overline{\mathtt{X}}\mathtt{>}\ ; \Gamma; \Sigma; \mathtt{C<}\overline{\mathtt{X}}\ \triangleleft\ \overline{\mathtt{N}}\mathtt{>} \vdash \mathtt{e}_0 : \mathtt{S} \\ \Delta; \Sigma \vdash \mathtt{S}\ \mathtt{<:}\ \mathtt{T} \qquad override(\mathtt{m},\ \mathtt{N},\ \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \rightarrow \mathtt{T}) \end{array}}{\mathtt{<}\overline{\mathtt{Y}}\ \triangleleft\ \overline{\mathtt{P}}\mathtt{>}\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \mathtt{FGO\ IN\ C<}\overline{\mathtt{X}}\ \triangleleft\ \overline{\mathtt{N}}\mathtt{>}}$$

Class Typing (Manifest and Pure):

$$\frac{\begin{array}{c} cparams(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \mathtt{K} = \mathtt{C}(\overline{\mathtt{U}}\ \overline{\mathtt{g}},\ \overline{\mathtt{T}_{\mathtt{cp}}}\ \overline{\mathtt{f}_{\mathtt{cp}}})\{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}}\ =\ initfield_{\overline{\mathtt{T}}}\overline{\mathtt{f}_{\mathtt{cp}}};\} \\ visible_{\Delta}(\mathtt{N},\ \mathtt{C}) \qquad visible_{\Delta}(\overline{\mathtt{T}},\ \mathtt{C}) \qquad visible_{\Delta}(\overline{\mathtt{N}},\ \mathtt{C}) \\ \Delta; \Sigma \vdash \overline{\mathtt{X}}\ \mathtt{<:}\ \overline{\mathtt{N}} \Rightarrow \mathtt{N}, \overline{\mathtt{T}}, \overline{\mathtt{N}}\ \mathsf{OK} \qquad \overline{\mathtt{M}}\ \mathtt{FGO\ IN\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \end{array}}{\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{FGO}}$$

$$\frac{\begin{array}{c} cparams(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \mathtt{K} = \mathtt{C}(\overline{\mathtt{U}}\ \overline{\mathtt{g}},\ \overline{\mathtt{T}_{\mathtt{cp}}}\ \overline{\mathtt{f}_{\mathtt{cp}}})\{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}}\ =\ initfield_{\overline{\mathtt{T}}}\overline{\mathtt{f}_{\mathtt{cp}}};\} \\ visible_{\Delta}(\mathtt{OCon},\ \mathtt{C}) \qquad visible_{\Delta}(\overline{\mathtt{T}},\ \mathtt{C}) \qquad visible_{\Delta}(\overline{\mathtt{N}},\ \mathtt{C}) \\ \mathtt{N} = \mathtt{C}'\mathtt{<}\overline{\mathtt{T}'}, \mathtt{OVar>} \qquad \Delta; \Sigma \vdash \overline{\mathtt{X}}\ \mathtt{<:}\ \overline{\mathtt{N}},\ \mathtt{OVar}\ \mathtt{<:}\ \mathtt{OCon} \Rightarrow \mathtt{N}, \overline{\mathtt{T}}, \overline{\mathtt{N}}\ \mathsf{OK} \\ \overline{\mathtt{M}}\ \mathtt{FGO\ IN\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}},\ \mathtt{OVar} \triangleleft \mathtt{OCon>} \end{array}}{\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}},\ \mathtt{OVar} \triangleleft \mathtt{OCon>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{FGO}}$$

Fig. 12. FGO Methods and Classes Rules

guished only by the fact that the owner may be explicit or must be looked up from a superclass.

3.7 Reduction and Congruence Rules

Figure 13 and 14 provide reduction and congruence rules that are standard FGJ rules with location-specific additions. Again, these are similar to the ones used by Featherweight Domains Java [2].

Reduction Rules:

$$\frac{l \notin dom(S) \qquad S' = S[l \mapsto \texttt{N}(\overline{v})]}{\texttt{new}\ \texttt{N}(\overline{v}), S \longrightarrow l, S'} \qquad \text{(R-New)}$$

$$\frac{S[l] = \texttt{N}(\overline{v}) \qquad fields(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{l.\texttt{f}_i, S \longrightarrow v_i, S} \qquad \text{(R-Field)}$$

$$\frac{S[l] = \texttt{N}(\overline{v}) \qquad mbody(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{N}) = \overline{\texttt{x}}.\texttt{e}_0}{l.\texttt{m<}\overline{\texttt{V}}\texttt{>}(\overline{v}), S \longrightarrow l \texttt{>} [\overline{v}/\overline{\texttt{x}}, l/\texttt{this}, \texttt{This}_l/\texttt{This}]\texttt{e}_0, S} \qquad \text{(R-Method)}$$

$$\frac{S[l] = \texttt{N}(\overline{v}) \qquad \texttt{N <: P}}{(\texttt{P})l, S \longrightarrow l, S} \qquad \text{(R-Cast)}$$

$$\frac{}{l \texttt{>} v, S \longrightarrow v, S} \qquad \text{(R-Context)}$$

Fig. 13. FGO Reduction Rules

Congruence Rules:

$$\frac{\texttt{e}, S \longrightarrow \texttt{e}', S'}{\texttt{e.f}, S \longrightarrow \texttt{e}'\texttt{.f}, S'} \qquad \text{(RC-Field)}$$

$$\frac{\texttt{e}_0, S \longrightarrow \texttt{e}_0', S'}{\texttt{e}_0\texttt{.m<}\overline{\texttt{T}}\texttt{>}(\overline{\texttt{e}}), S \longrightarrow \texttt{e}_0'\texttt{.m<}\overline{\texttt{T}}\texttt{>}(\overline{\texttt{e}}), S'} \qquad \text{(RC-Inv-Recv)}$$

$$\frac{\texttt{e}_i, S \longrightarrow \texttt{e}_i', S'}{\texttt{e}_0\texttt{.m<}\overline{\texttt{T}}\texttt{>}(\ldots, \texttt{e}_i, \ldots), S \longrightarrow \texttt{e}_0\texttt{.m<}\overline{\texttt{T}}\texttt{>}(\ldots, \texttt{e}_i', \ldots), S'} \qquad \text{(RC-Inv-Arg)}$$

$$\frac{\texttt{e}_i, S \longrightarrow \texttt{e}_i', S'}{\texttt{new}\ \texttt{N}(\ldots, \texttt{e}_i, \ldots), S \longrightarrow \texttt{new}\ \texttt{N}(\ldots, \texttt{e}_i', \ldots), S'} \qquad \text{(RC-New-Arg)}$$

$$\frac{\texttt{e}, S \longrightarrow \texttt{e}', S'}{(\texttt{N})\texttt{e}, S \longrightarrow (\texttt{N})\texttt{e}', S'} \qquad \text{(RC-Cast)}$$

$$\frac{\texttt{e}, S \longrightarrow \texttt{e}', S'}{l\texttt{>e}, S \longrightarrow l\texttt{>e}', S'} \qquad \text{(RC-Loc)}$$

Fig. 14. FGO Congruence Rules

4 FGO Theorems

Theorem (Preservation). *If* $\Delta; \Gamma; \Sigma; P \vdash \texttt{e} : \texttt{T}$ *and* $\Delta; \Sigma \vdash S$ *and* $\texttt{e}, S \rightarrow \texttt{e}', S'$*, then* $\exists \Sigma' \supseteq \Sigma$ *and* $\exists \texttt{T}' \texttt{<:} \texttt{T}$ *such that* $\Delta; \Gamma; \Sigma; P \vdash \texttt{e}' : \texttt{T}'$ *and* $\Delta; \Sigma' \vdash S'$.

Proof. Using structural induction on five reduction rules in figure 13.

$$\boxed{\dfrac{l \notin dom(S) \qquad S' = S[l \mapsto \mathtt{N}(\overline{v})]}{\mathtt{new}\ \mathtt{N}(\overline{v}), S \rightarrow l, S'}\ \text{(R-New)}}$$

Expression: By T-New we have $\mathtt{e} = \mathtt{new}\ \mathtt{N}(\overline{v})\colon \mathtt{T}$ where $\mathtt{T} = \mathtt{N}$. By T-Loc $\mathtt{e}' = l\colon \mathtt{T}'$ where $\mathtt{T}' = \Sigma'(l)$. By FGO-Store, $\Sigma'(l) = \mathtt{N}$ if and only if $S'(l) = \mathtt{N}(\overline{v})$, but the latter holds by definition in R-New. Therefore $\mathtt{T}' = \mathtt{N} \mathrel{\mathtt{<:}} \mathtt{N} = \mathtt{T}$ as required.

Store: Define $\Sigma' = \Sigma[l \rightarrow \mathtt{N}]$. We show that $\Delta; \Sigma' \vdash S'$ using FGO-Store rule. By definition of Σ' and S': $dom(S') = dom(S) \cup \{l\} = dom(\Sigma) \cup \{l\} = dom(\Sigma')$ and $S'[l] = \mathtt{N}(\overline{v})$ and $\Sigma'[l] = \mathtt{N}$. By T-New $\Delta; \Sigma \vdash \mathtt{N}\ \mathsf{OK}$ and by definition $\Sigma'(l) = \mathtt{N}$, and hence $\Delta; \Sigma \vdash \Sigma'(l)\ \mathsf{OK}$ and by FGO-Store-WF $\Delta \vdash \Sigma'\ \mathsf{OK}$.

Consider any field i in $fields(\Sigma'(l)) = fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$. By T-New and $initfield_{\mathtt{T}}$ definition (where $\overline{\mathtt{e}} = \overline{v}$ and $\overline{\mathtt{S}} = \Sigma(\overline{v})$) $\Delta; \Sigma \vdash \Sigma(v_i) \mathrel{\mathtt{<:}} \mathtt{T}_i$ and $\Sigma'(v_i) = \Sigma(v_i)$ by definition. But $v_i = S'[l, i]$ by definition of S' and hence: $(S[l, i] = l') \wedge (fields(\Sigma'[l]) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}) \Rightarrow \Delta; \Sigma' \vdash \Sigma'[l'] \mathrel{\mathtt{<:}} \mathtt{T}_i$.

Finally, by FGO-Store-WF, $\Delta; \Sigma' \vdash \Sigma'(l')\ \mathsf{OK}$ since if $l' \neq l$, then $l' \in \Sigma$ and $\Delta; \Sigma \vdash S$ makes $\Sigma(l')\ \mathsf{OK}$ and $\Sigma'(l') = \Sigma(l)$; and for $l' = l$, we already established that $\Sigma'(l)\ \mathsf{OK}$. Therefore $\Delta; \Sigma' \vdash S'$ as required.

$$\boxed{\dfrac{S[l] = \mathtt{N}(\overline{v}) \qquad fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{l.\mathtt{f}_i, S \rightarrow v_i, S}\ \text{(R-Field)}}$$

Expression: By T-Field we have $\mathtt{e} = l.\mathtt{f}_i : \mathtt{T}$ where $\mathtt{T} = [this_P(\mathtt{e}_0)/\mathtt{This}]\mathtt{T}_i$. By T-Loc, $\mathtt{e}' = v_i : \mathtt{T}'$ where $\mathtt{T}' = \Sigma'(v_i)$. By FGO-Store and since $\Sigma' = \Sigma$, we have $\Sigma'(v_i) \mathrel{\mathtt{<:}} \mathtt{T}_i$. But by S-Owner $\Delta; \Sigma \vdash \mathtt{This}_l \mathrel{\mathtt{<:}} \mathtt{This}$ and *this* function at dynamic time will provide us with $\mathtt{This}_l$ for some location l. Therefore $\mathtt{T}' = \mathtt{T}_i \mathrel{\mathtt{<:}} [this_P(\mathtt{e}_0)/\mathtt{This}]\mathtt{T}_i = \mathtt{T}$ as required.

Store: Define $\Sigma' = \Sigma \supseteq \Sigma$, to show that $\Delta; \Gamma; \Sigma' \vdash S'$, we need $\Delta; \Gamma; \Sigma \vdash S$, which already holds.

$$\boxed{\dfrac{S[l] = \mathtt{N}(\overline{v}) \qquad mbody(\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}, \mathtt{N}) = \overline{\mathtt{x}}.\mathtt{e}_0}{l.\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}(\overline{v}), S \rightarrow l\ \texttt{>}\ [\overline{v}/\overline{\mathtt{x}}, l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]\mathtt{e}_0, S}\ \text{(R-Method)}}$$

Expression: By T-Method we have $\mathtt{e} = l.\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}(\overline{v}) : \mathtt{T}$ where $\mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}, this_P(l)/\mathtt{This}]\mathtt{U}$ and $mtype(\mathtt{m}, bound_\Delta(\Sigma(l))) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U}$ and for some class $\mathtt{C}$ such that $\mathtt{N} \mathrel{\mathtt{<:}} \mathtt{C}$ method $\mathtt{m}$ is declared in it (by MT-Class). By method typing rule, $\mathtt{e}_0 : \mathtt{U}$ and hence by T-Context we have $\Delta; \Gamma; \Sigma; l \vdash [\overline{v}/\overline{\mathtt{x}}, l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]\mathtt{e}_0 : [l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]\mathtt{U} = \mathtt{T}'$. Finally, since $P = l$, our *this* function expands $this_P(l)$ into $[l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]$ substitution and $\mathtt{T}' \mathrel{\mathtt{<:}} \mathtt{T}$ as required.

Store: Define $\Sigma' = \Sigma \supseteq \Sigma$, to show that $\Delta; \Gamma; \Sigma' \vdash S'$, we need $\Delta; \Gamma; \Sigma \vdash S$, which already holds.

$$\boxed{\dfrac{S[l] = \mathtt{N}(\overline{v}) \qquad \mathtt{N} \mathrel{\mathtt{<:}} \mathtt{P}}{(\mathtt{P})l, S \rightarrow l, S}\ \text{(R-Cast)}}$$

Expression: By one of T-UCast or T-DCast or T-SCast we have $\texttt{e} = (\texttt{P})l : \texttt{T}$ where $\texttt{T} = \texttt{P}$. By T-Loc $\texttt{e}' = l : \texttt{T}'$ where $\texttt{T}' = \Sigma(l) = \texttt{N}$ by FGO-Store since $S[l] = \texttt{N}(\overline{v})$. $\texttt{T}' = \texttt{N} \texttt{<:} \texttt{P} = \texttt{T}$ as required.

Store: Define $\Sigma' = \Sigma \supseteq \Sigma$, to show that $\Delta; \Gamma; \Sigma' \vdash S'$, we need $\Delta; \Gamma; \Sigma \vdash S$, which already holds.

$$\boxed{\frac{}{l > v, S \to v, S}\ \text{(R-Context)}}$$

Expression: Given that $\texttt{e} = l > v : \texttt{T}$ if and only if $v : \texttt{T}$ by T-Context, we have $\texttt{e}' = v : \texttt{T}$ as required.

Store: Define $\Sigma' = \Sigma \supseteq \Sigma$, to show that $\Delta; \Gamma; \Sigma' \vdash S'$, we need $\Delta; \Gamma; \Sigma \vdash S$, which already holds. ■

Theorem (Progress). *Suppose* e *is a well-typed expression. Then either* e *is a value (location or a variable) or there is a reduction rule that contains* e *on the left hand side.*

Proof. Based on all the possible expression types, either e is a variable or location (T-Var and T-Loc) or one of the reduction rules applies. We need to check that each of the reduction rules is satisfied. The only rules that require additional conditions are R-Field and R-Method - in the case of R-Cast the program gets stuck if the downcast is impossible (similar to FGJ).

In case of R-Field, well-typedness of N ensures that $fields(\texttt{N})$ is well defined and $\texttt{f}_i$ appears in it. In case of R-Method, the fact the $mtype$ looks up the type for m, ensures that $mbody$ will succeed too and will have the same number of arguments (since MT-Class and MB-Class are defined in the same way).■

Type soundness is then immediate from preservation and progress theorems.

Lemma (Ownership Invariance). *If* $\Delta; \Sigma \vdash \texttt{S <: T}$ *and* $\Delta; \Sigma \vdash \texttt{T <: CObject<O>}$, *then* $owner_\Delta(\texttt{S}) = owner_\Delta(\texttt{T}) = \texttt{O}$.

Proof. By induction on the depth of the subtype hierarchy. By FGO class typing rules a FGO class has the same owner parameter as its superclass. ■

Theorem (Confinement Invariant). *Let* e *be a subexpression appearing in the body of a method of a well-formed FGO class* C. *Then: If* $\texttt{e} \to^* \texttt{new D<}\overline{\texttt{T}_\texttt{D}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$, *then* $visible_\Delta(\texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{>, C})$.

Proof. Because the class is a well-formed FGO class, its methods are well-formed FGO methods. This, plus the standard subformula property, implies that, for appropriate $\Delta; \Gamma; \Sigma; P$, both $\Delta; \Gamma; \Sigma; P \vdash \texttt{e} : \texttt{T}$ and $visible_\Delta(\texttt{e, C})$ hold. From this we can derive $visible_\Delta(\texttt{T, C})$, and hence $visible_\Delta(owner_\Delta(\texttt{T}), \texttt{C})$. By FGO's subject reduction property, there is a $\texttt{T}'$ such that $\Delta; \Gamma; \Sigma; P \vdash \texttt{new D<}\overline{\texttt{T}_\texttt{D}}\texttt{>(}\overline{\texttt{e}}\texttt{)} : \texttt{T}'$, where $\Delta; \Sigma \vdash \texttt{T}' \texttt{<: T}$. Furthermore, we have that $\Delta; \Gamma; \Sigma; P \vdash \texttt{new D<}\overline{\texttt{T}_\texttt{D}}\texttt{>(}\overline{\texttt{e}}\texttt{)} : \texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{>}$, and hence clearly $\Delta; \Sigma \vdash \texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{> <: T}'$, and $\Delta \vdash \texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{> <: T}$. By the Ownership Invariance lemma, $owner_\Delta(\texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{>}) = owner_\Delta(\texttt{T})$, from which we deduce $visible_\Delta(owner_\Delta(\texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{>}), \texttt{C})$, and hence $visible_\Delta(\texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{>, C})$. ■

Theorem (Ownership Invariant). *Given any expression of the form* l `>` `e` *during the reduction of any FGO program with variable to nonvariable types mapping* Δ*. If* $\mathtt{e}, S \rightarrow^{*} v, S'$*, then there exist* Σ *and* Σ' *such that* $\Delta; \Sigma \vdash S$ *and* $\Delta; \Sigma' \vdash S'$ *and* $\mathit{visible}_{\Delta}(\Sigma'(v), \Sigma(l))$*.*

Proof (sketch). By FGO subject reduction, $\Sigma'(v)$ `<:` $\Sigma(l)$ and $\Sigma' \supseteq \Sigma$. By ownership invariance, $\mathit{owner}_{\Delta}(\Sigma(l)) = \mathit{owner}_{\Delta}(\Sigma'(l)) = \mathit{owner}_{\Delta}(\Sigma'(v))$. By V-OWNER, $\mathit{visible}_{\Delta}(\Sigma'(v), \Sigma(l))$. ■

5 Discussion and Conclusion

The main contribution of FGO is providing ownership as an integral part of type polymorphism, rather than having two separate parameter spaces for ownership and genericity [3]. This approach turns out to be simpler, with less rules and simpler proofs, due to the existing type polymorphic framework. FGO also turns out to be easier to implement, as we have done by writing Ownership Generic Java compiler [16] on top of Java 5 compiler source [20], since the reuse of the existing type checking for generic types saves huge amounts of programming effort. Finally, we argue that having a simple addition to the language (one additional generic parameter denoting the ownership of a particular instance) makes it easier for the programmers to learn and adopt ownership in their programs.

On the other hand, having two separate parameter spaces for ownership and genericity may make programs more flexible. A number of limitations are present in FGO in its current form. FGO only supports shallow ownership (there is no guarantee that objects in an FGO program will form a dominator tree of the object graph based on their ownership information). FGO doesn't have assignment as part of the type system. For simplicity, we chose to prohibit making a package-confined FGO class private to a particular instance (since $\mathtt{This}_{l}$ owner class is not a subclass of $\pi_{\mathtt{C}}$).

While the addition of assignment and extending the owner hierarchy are technicalities that won't affect the big picture of FGO, other than adding more rules and increasing the size of the type system (e.g. verifying the heap well-formedness requires more work if assignment is present), the addition of deep ownership is slightly harder to deal with. At this stage, to achieve deep ownership, we plan to introduce checks similar to other deep ownership type systems [6, 3] for how owner parameters can be *nested* in a type.

For example, we may want to prohibit types like `p.OwnedStack<m.Main<This>, World>` since it would potentially allow pointers into the objects stored inside the stack from outside of the object who intends to *own* the stack and thus to form a dominator of the `Stack` object in the ownership tree of the program's object graph.

Adding assignment and providing deep ownership support are all ongoing subjects of our work in Featherweight Generic Ownership. We are also exploring the possibilities provided by ownership inference and formulating a set of design patterns utilising ownership support. The system we have presented here, however, is the first type system providing object ownership support and type polymorphism in a unified manner.

Acknowledgments

This work is supported by the RSNZ Marsden Fund and Claude McCarthy Fellowship.

Bibliography

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seattle, WA, USA, November 2002. ACM Press, New York, NY, USA.

[2] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, Oslo, Norway, June 2004. Springer-Verlag, Berlin, Heidelberg, Germany.

[3] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 2004.

[4] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *Proceddings of ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, LA, USA, January 2003. ACM Press, New York, NY, USA. Invited talk by Barbara Liskov.

[5] Dave Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Australia, 2002.

[6] Dave Clarke and Sophia Drossopoulou. Ownership, Encapsulation, and the Disjointness of Type and Effect. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seattle, WA, USA, November 2002. ACM Press, New York, NY, USA.

[7] Dave Clarke, Michael Richmond, and James Noble. Saving the World from Bad Beans: Deployment-Time Confinement Checking. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, California, USA, October 2003. ACM Press, New York, NY, USA.

[8] David Clarke, John Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver, Canada, October 1998. ACM Press, New York, NY, USA.

[9] David Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In Luca Cardelli, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 2473 of *Lecture Notes in Computer Science (LNCS)*, pages 176–200, Darmstadt, Germany, July 2003. Springer-Verlag, Berlin, Heidelberg, Germany.

[10] Matthew Fluet and Riccardo Pucella. Phantom Types and Subtyping. In *Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science (TCS)*, pages 448–460, August 2002.

[11] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating Objects with Confined Types. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tampa Bay, FL, USA, 2001. ACM Press, New York, NY, USA.

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

[13] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, May 2001.

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

[15] P. Müller and A. Poetzsch-Heffter. *Programming Languages and Fundamentals of Programming*, chapter Universes: a Type System for Controlling Representation Exposure. Fernuniversität Hagen, 1999.

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

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

[18] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight Generic Confinement. In *Foundations of Object-Oriented Languages (FOOL11)*, Venice, Italy, January 2004.

[19] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight Generic Confinement. *Journal of Functional Programming*, 2005. Submitted for publication.

[20] Sun Microsystems. Java Development Kit. Available at: `http://java.sun.com/j2se/`, 2005.

[21] Jan Vitek and Boris Bokowski. Confined Types in Java. *Software — Practice & Experience*, 31(6):507–532, May 2001.

[22] Tian Zhao, Jens Palsberg, and Jan Vitek. Lightweight confinement for Featherweight Java. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2003.

[23] Tian Zhao, Jens Palsberg, and Jan Vitek. Type-Based Confinement. *Journal of Functional Programming*, 2005. Accepted for publication.