Generic Ownership

A Practical Approach to Ownership and Confinement in Object-Oriented Programming Languages

by

Alex Potanin

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2007

Abstract

Modern object-oriented programming languages support many techniques that simplify the work of a programmer. Among them is *generic types*: the ability to create generic descriptions of algorithms and object structures that will be automatically specialised by supplying the type information when they are used. At the same time, object-oriented technologies still suffer from *aliasing*: the case of many objects in a program's memory referring to the same object via different references.

Ownership types enforce encapsulation in object-oriented programs by ensuring that objects cannot be referred to from the outside of the object(s) that *own* them. Existing ownership programming languages either do not support generic types or attempt to add them on top of ownership restrictions.

The goal of this work is to bring object ownership into mainstream object-oriented programming languages. This thesis presents Generic Ownership which provides per-object ownership on top of a generic imperative language. Surprisingly, the resulting system not only provides ownership guarantees comparable to the established systems, but also requires few additional language mechanisms to achieve them due to full reuse of generic types.

In this thesis I formalise the core of Generic Ownership, highlighting that the restriction of `this` calls, owner preservation over subtyping, and appropriate owner nesting are the only necessary requirements for ownership. I describe two formalisms: (1) a simple formalism, capturing confinement in a functional setting, and (2) a complete formalism, providing a way for Generic Ownership to support both deep and shallow variations of ownership types.

I support the formal work by describing how the Ownership Generic Java (OGJ) language is implemented as a minimal extension to Java 5. OGJ is the first publicly available language implementation that supports ownership, confinement, and generic types at the same time. I demonstrate OGJ in practice: show how to use OGJ to write programs and provide insights into the implementations of Generic Ownership.

Acknowledgments

I would like to thank everyone who deserves to be thanked and more. First and foremost is my wife Nelly, my parents Arthur and Vera, my sisters Christina and Victoria, my grandparents Elena and Vitaly, and my new family: Betti and Michael. Secondly, my friends Craig, Keith, Donald, Azat, Angela, Rilla, Pippin, Matt, David, Stuart, Bruce, Chris, Simon and all of the helpful members of the ELVIS Research Group. In particular thanks to David Pearce and Craig Anslow for proofreading my thesis in great detail. Last, but not least, I would like to thank my supervisors: James Noble, Robert Biddle, and Dave Clarke. They generally preferred to be on a different continent from me at all times during the writing of this thesis, but their help was nonetheless invaluable. Thanks to the various people and organisations kindly providing scholarships and grants (Claude McCarthy, Faculty of Science, and Marsden in particular). Finally, thanks to Victoria University for employing me.

Contents

List of Figures

Chapter 1

Introduction

"The big lie of object-oriented programming
is that objects provide encapsulation"
John Hogg [Hog91]

Object-oriented programs, when executed, create a complex web of objects in memory that work together by exchanging messages via references between each other. One of the ways that encapsulation in object-oriented languages is compromised is by unexpected changes to objects via aliases. Ownership is one approach to addressing this issue that identifies an *owner* for each object and ensures that changes can only be made through the owner.

Aliasing — both a blessing and a bane — occurs when there is more than one object referring to a single object. Changes made to the single object by one of the referrers may go unnoticed by the other referrers and cause subtle bugs that may compromise the operation of large run-time systems like the Java Virtual Machine [BV99] just as buffer overflows can bring down non-memory managed systems [CWP+03].

There are a number of recent developments that provide a way to control aliasing in object-oriented programming languages [AKC02, Boy04]. Ownership types is one of the proposals to control aliasing [Hog91, Alm97, MPH99, AKC02, Cla02, BLS03, LM04]. Unfortunately, these approaches either lack support for modern object-oriented language features (e.g., generic types) or lack a usable language implementation.

The goal of my thesis is bringing ownership into a modern object-oriented language with full support for advanced features like generic types. I show how Java 5 generic types can be used to provide a working language with ownership support and I provide a publicly available language implementation.

1.1 Motivation

Object ownership ensures that objects cannot be leaked beyond an object or collection of objects which *own* them. Confinement is a variation of ownership that restricts objects to the packages or classes rather than individual instances. There are two main approaches to object ownership: (1) enforcing coding conventions within an existing programming language [Hog91, VB01, CRN03, GPV01, ZPV06], and (2) significantly modifying a language to provide ownership support [MPH99, AKC02, CD02, BLS03]. All of the existing ownership systems either do not consider generic types or add them as a further extension orthogonal to the object ownership concepts.

Examples of the first approach include Islands [Hog91], and various kinds of Confined Types [VB01, CRN03]. Here, programs must be written to follow a set of specific conventions, conformance to which can be checked to see if they provide ownership guarantees [GPV01]. Support for generics is added on top of such collections of restrictions that enforce encapsulation [ZPV06].

Examples of the second approach include languages such as Universes [MPH99], AliasJava [AKC02], Joe [CD02], and SafeJava [BLS03]. Here ownership parameterisation is added to the syntax and expressed explicitly within the type systems of these languages. All of these different languages employ ownership parameterisation, but neither of these languages have support for generic types.

Previous ownership and confinement type systems [Cla02, Boy04, ZPV06] considered adding generic types to an ownership types system as an unrelated issue: without realising the benefits the generic types can bring to ownership types.

Why is it desirable to combine ownership and generic types? Consider for example a *box* as a kind of object. In any typed object-oriented language one is allowed to say: “this is a box” (meaning a box capable of containing anything). In a language with generics, one is allowed to say: “this is a box *of books*”, denoting a box containing books, but not birds. In a language with ownership parameterisation, one is allowed to say: “this is *my* box” or “these are *library* books”. Combining ownership and generics naturally allows us to say: “this is *my* box *of library books*”, not a box of birds, and not my personal books. Ownership works well with generics, both in theory and in practice.

1.2 Contributions

The key contribution of my thesis is the design of a language that combines both ownership and generics. In more detail, my four contributions are:

1. **Generic Ownership** — a new way of combining ownership and generics by using a single parameter space to carry both type and ownership information.

2. **Featherweight Generic Confinement (FGC)** — a formal model showing how Generic Ownership provides confinement (ownership by packages and classes rather than instances) guarantees with only a few simple restrictions to an established Java formalism.

3. **Featherweight Generic Ownership (FGO)** — a formal model demonstrating how ownership, confinement, and generic types can be combined together in one type system using the Generic Ownership approach.

4. **Ownership Generic Java (OGJ)** — a language design that supports Generic Ownership — it is the first language design that supports ownership, confinement, and generics at the same time.

In my thesis I show that ownership systems can be greatly simplified by reusing generic type systems, and formalise this approach within the context of an imperative extension to Featherweight Generic Java (FGJ) [IPW01a].

Generic Ownership shows that a large amount of work done by the non-generic ownership type systems can be done by a standard type generic type system. This means that the concepts essential to ownership are a small number of owner preservation restrictions and careful handling of the `this` variable.

I provide a full formalism supporting ownership and generic types. I prove that the Generic Ownership formalism provides the same ownership guarantees as the other ownership type systems, only it is simpler to formulate and reason about. I provide a language extension requiring no syntax changes to Java to support ownership. Hence, I claim to have provided a very clean way of introducing ownership into a modern programming language like Java (and C#).

1.3 Outline

The remainder of this thesis is structured as follows. Chapter 2 surveys the concepts underlying the core of this thesis and discusses related work. Chapter 3 provides a general overview of Generic Ownership. Chapter 4 shows a clean and simple way of providing a limited form of ownership (confinement) that is completely subsumed by a standard generic type system (Featherweight Generic Java [IPW01a]). Chapter 5 describes the full formal foundations of Generic Ownership in a type system called Featherweight Generic Ownership. Chapter 6 provides an overview of the Ownership Generic Java language, its implementation and applications to real world programming problems. This thesis provides motivational, formal, and implementation support for the introduction of Generic Ownership into modern object-oriented programming languages such as Java and C#. Finally, Chapter 7 puts the work described in this thesis in perspective, discusses potential

future developments for Generic Ownership research and concludes concepts and ideas of this thesis.

Chapter 2

Background

Contents

This chapter describes the aspects of the object-oriented programming languages field that engendered the ideas of generics and ownership. I start with Section 2.1 outlining the basics of object-oriented programming and the idea of an object graph. Section 2.2 presents the concept of aliasing and Section 2.3 discusses encapsulation and provides some insight into the different ways of dealing with problems caused by aliasing. Section 2.4 gives an overview of research in alias management techniques. Finally, Section 2.5 lays out the formal foundations that Generic Ownership builds upon.

2.1 Object-Oriented Programming

As computers grow in computational power, so does the complexity of the software that comes with them. To help manage the growing size of software development projects, researchers developed the concept of *object-oriented programming*. Booch [Boo91] describes object-oriented programming as::

> Object-oriented programming is a method of implementation in which programs are organised as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

Object-oriented programming is extremely popular and is widely adopted by the programming community. Object-oriented programming is shown to improve the understandability, maintainability, and re-usability of software artefacts [WBW89]. In the object-oriented paradigm programs are structured as collections of classes that model entities in the real world, and program execution amounts to a large number of instances of these classes (*objects*) working together to achieve a common goal.

One of the characteristics of object-oriented programs is that there are a large number of objects present inside the program's memory during its execution. An typical Java program contains around 60,000 objects and this number can range anywhere between 3,000 and several million objects [PNB04]. Java programs are becoming larger and larger reaching the heap sizes of many millions of objects [Mit06].

Objects are connected to each other via *references* and objects utilise these references to issue commands or send messages to other objects with which they cooperate. Thus, the contents of a typical program's memory can be thought of as a directed graph of objects, known as an *object graph*.

Software engineering, among many things, is concerned with software reliability. Given that the object graph is the foundation of most modern software products, understanding a graph's behaviour and guaranteeing reliable operation forms one of the essential tasks for researchers.

2.1.1 An Object Graph

An object graph — the object instances in the program and the links between them — is the skeleton of an object-oriented program. Because each node in the graph represents an object, the graph grows and changes as the program runs. A graph contains just a few objects when the program is started, gains more objects as they are created, and loses objects when they are no longer required. The structure of the graph (the links between objects) changes too, as every assignment to an object's non-primitive field makes or changes an edge in the graph.

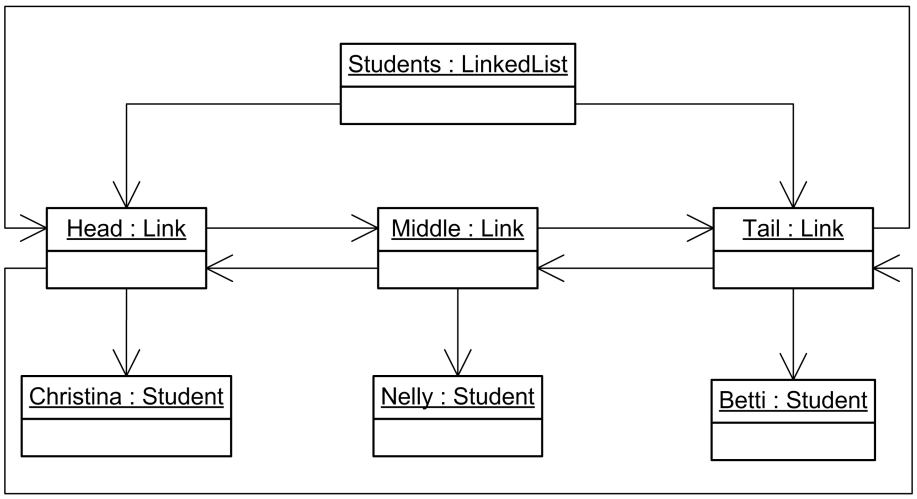

Figure 2.1: The Object Graph of a Circular Doubly Linked List

Figure 2.1 illustrates an object graph of a simple part of a program, in this case, a circular doubly linked list of `Student` objects. The list itself is represented by the `LinkedList` object (an instance of the `LinkedList` class, presumably), which has two references to `Link` objects representing the head and tail of the list. Each `Link` object has two references to other `Link` objects, the previous and the next `Links` in the list, and a third reference to one of the `Student` objects contained in the list. Although the overall structure is clearly a general directed graph with many cycles, rather than a tree

or a directed acyclic graph, some objects (such as the `Student` "Nelly") are accessed uniquely by only a single reference.

The object graphs form the foundation of any object-oriented program. The graphs represent the model of the world created by a software designer when describing the system of classes. The techniques employed by modern object-oriented design (classes, associations, interfaces, inheritance, packages, patterns, UML, CRC Cards, etc.) can be considered to be techniques for defining object graphs by describing the contents of the objects and the structure of the links between objects.

2.1.2 An Object-Oriented Program Example

Figure 2.2 shows a Java implementation of a simple `Map` class that uses neither genericity nor ownership. The map is implemented using a `Vector` containing a number of `Nodes`, each of which stores a key-value pair. The main `Map` class provides methods to insert a new key-value pair into the map and to return the first value associated with a particular key. This example will be used to highlight the advantages of Generic Ownership throughout the chapter and the following chapters.

```
1  public class Map extends Object {
2    private Vector nodes;
3
4    void put(Object key, Object value) {
5      nodes.add(new Node(key, value));
6    }
7
8    Object get(Object k) {
9      Iterator i = nodes.iterator();
10     while (i.hasNext()) {
11       Node mn = (Node) i.next();
12       if (mn.key.equals(k))
13         return mn.value;
14     }
15     return null;
16   }
17 }
18
19 class Node {
20   public Object key; public Object value;
21   Node(Object key, Object value) {
22     this.key = key; this.value = value;
23   }
24 }
```

Figure 2.2: `Map` Class Example (Java)

2.2 Aliasing

An object is *aliased* every time there is more than one pointer referring to that object [HLW+92]. Aliasing can cause a range of difficult problems within object-oriented programs, because one referring object can change the state of the aliased object, implicitly affecting all the other referring objects [NVP98, VB01].

In popular object-oriented programming languages aliasing is endemic and unavoidable: every assignment statement, potentially, causes an extra alias to be created. There is a lot of research that addresses the problems caused by aliasing, including alias protection schemes [MPH99, NVP98] and, in the case of assignment, alternatives to assignment statement in component-based engineering [HW91].

In this section a number of small examples of the problems caused by aliasing illustrate the motivation behind a large amount of research performed in this area.

2.2.1 The Importance of an Object's Identity Being Unique

In the real world distinct objects are clearly unique. For example, if a door comes out of a factory (which might be one of many similar doors) and is installed as the front door of your house, it is very different from an identical door installed in someone else's house. If you paint your door pink, you neighbour's door does not also become pink.

The objects in memory are similar to the real world and their identity should be respected. If an instance of a "Close" button is created for a particular window, one hopes that it is unique to that window and clicking on it does not close some completely unrelated window. Thus, it is important for every object in the object graph to be unique and every object must be treated as such. Having two windows refer to the same button would be no more practical than having two houses have the same door.

While uniqueness of objects is closely related to the real world, having multiple references to the same object from unrelated other objects is common in object-oriented programming. For example, data representation may involve storing the same record in multiple collections. The state of the objects affected by aliasing can be easily changed without their knowledge. While this compromise is desirable in certain software designs, the cases of *unexpected* aliasing in day to day programs still have to be addressed.

2.2.2 An Example of Aliasing

Consider the classes `Point` and `Rectangle` shown in Figure 2.3. Let's say one creates an instance of the `Rectangle` class as shown in Figure 2.3. Consider what will happen if one is to do the following later on in the `main` method:

```
p.x = 400;
```

This statement will not only change the value of the point stored in the local variable `p`, it will also modify the position of the top-right corner of the original `Rectangle`.

This is a very simple example (tested in Java 5) of the problems that can be caused by aliasing, but in the next subsections, I will present a number of examples that are closer to real life. The issue at hand here is that even though a field `topLeft` in class `Rectangle` is declared as `private`, Java's *name-based* encapsulation mechanism (that hides *names* of objects rather than objects themselves) is insufficient to prevent its value from being modified outside of the `Rectangle` object. Two obvious ways to expose private fields in Java are using the supplied constructor parameters to directly initialise the fields (line 5 in the example in Figure 2.3) or provision of getter and setter methods. While a good software engineering practice of getter and setter methods hides the implementation details of the field, it does not prevent unexpected changes to the `Rectangle` object unaccounted for by the designer of the `Rectangle` class. More than just name-based encapsulation is required for the programmer to be able to truly hide the implementation details of the classes.

```
1  class Point {
2    public Integer x; public Integer y;
3
4    public Point(Integer x, Integer y) {
5      this.x = x; this.y = y;
6    }
7  }
8
9  class Rectangle {
10   private Point topleft;
11   private Integer width; private Integer height;
12
13   public Rectangle(Point topleft,
14                    Integer width, Integer height) {
15     this.topleft = topleft;
16     this.width = width; this.height = height;
17   }
18
19   public static void main(String[] args) {
20     Point p = new Point(50, 100);
21     Integer width = 300; Integer height = 200;
22     Rectangle r = new Rectangle(p, width, height);
23   }
24 }
```

Figure 2.3: `Point` and `Rectangle` Aliasing Example

2.2.3 Example: The Elements Inside a Hashtable

Aliasing often arises in data structures. A good example of aliasing in hashtables is discussed in detail by Noble, Potter, and Vitek in their work on Flexible Aliasing Protection [NVP98].

Imagine a typical container like `Map` implemented using a hashtable to maintain a student database. Figure 2.4 (drawn after Figure 3 in [NVP98]) depicts a sample system with a `Hashtable` object having fields named `size` and `table`. Any object inside the system can use this hashtable to store information about the students. Every time a `Student` and, say, their `RawMark` are added to the hashtable, an `Entry` object is created inside the `Array` pointed at by the `table` field. The entry stores the reference to the `Student` object given to it and uses it as a `key` inside the hashtable.

Now, there is no guarantee that nothing else is still pointing at the `Student` object (as shown by the arrows pointing at some of the objects out of nowhere in Figure 2.4). This means that some of the objects that the hashtable relies upon for its state (i.e. the location of the elements in the array is based on their hash code) are aliased.

To clarify the state of the hashtable, consider the Figure 2.5 (drawn after Figure 1 in [NVP98]). A hashtable `a` points at an object with its `size` and an object with its `contents`. The elements inside the hashtable contents are objects named `i`, `j`, etc. Adding another object, say `k`, that is still pointed at by the external object `d` will cause one of the elements to be aliased and `d` will be potentially able to modify the state of the hashtable `contents` as shown by the arrow between `d` and `contents` in the figure.

The problem lies in the fact that, using the design described above, being able to modify parts of the hashtable without its knowledge is not preventable in modern object-oriented programming languages. If the design of a hashtable is such that it requires it to have unmodifiable keys while they are stored in it (lest it loses track of their location due to a changed hash code for the key), there is no way to reliably implement such a design in Java or C#.

One solution to prevent changing keys in a hashtable is to have object immutability support in the language [ZPA+07] — marking keys in a hashtable unmodifiable even by the direct referrers. Unfortunately, this may be too restrictive, since even the owner of the key (say, a hashtable) is no longer going to be allowed to change it.

This example illustrates that something as simple as a hashtable — widely used in many programs written in Java — has a huge potential for aliasing related errors.

2.2.4 Example: Java Applet Security Breach in JDK v1.1.1

An example of aliasing causing major software failure is described in the paper on confined types by Bokowski and Vitek [BV99].

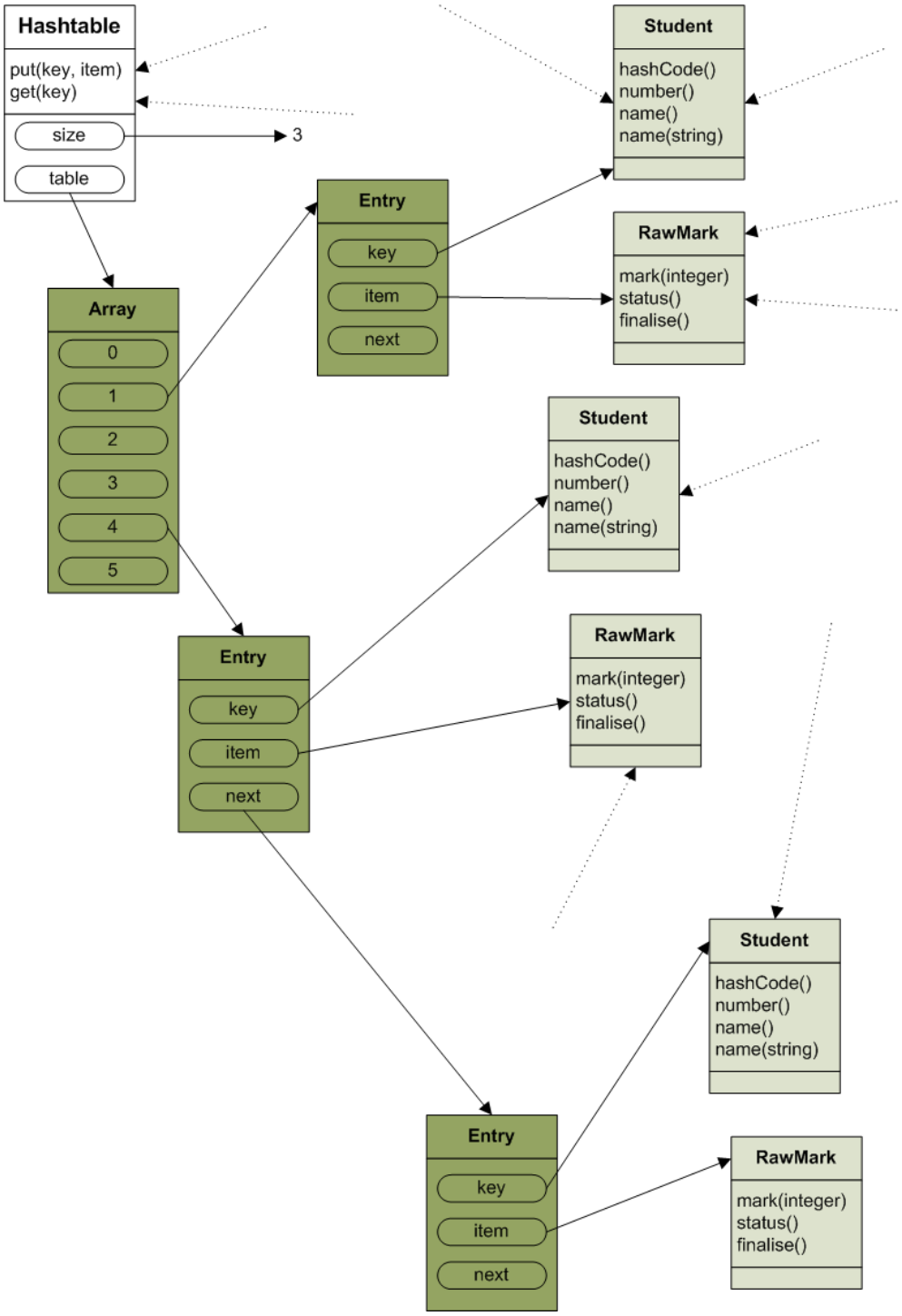

Figure 2.4: Object Diagram of a Program Using a Hashtable

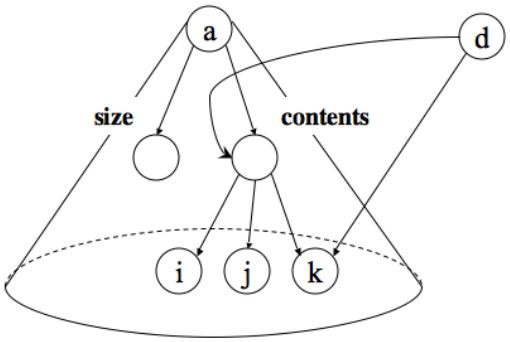

Figure 2.5: An Explanation of Hashtable Aliasing

> In Java, each class object (instance of class `Class`) stores a list of signers, which contains references to objects of type `java.security.Identity`, representing the principals under whose authority the class acts. This list is used by the security architecture to determine the access rights of the class at runtime. A serious security breach was found in the JDK v1.1.1 implementation which allowed untrusted code to acquire extended access rights [Gro97]. The breach is due to a reference to the internal list of signers leaking out of the implementation of the security package into an untrusted applet.

The way the breach can be exploited by a malicious applet is briefly outlined in Figure 2.6. Every class, including the applet's own class, has an array of `Identities` stored for it inside the `java.security` package. This collection of so-called signers defines the access rights of an applet or any other class. There is also a system-wide accessible array of all possible `Identities` accessible through `java.security.IdentityScope`. Therefore, since any applet could potentially obtain the reference to its own array, the applet can modify its own array by adding the rest of the `Identities`.

The reason for this security breach lies in the code for the `getSigners` method in `java .security` package:

```
private Identity[] signers;
...
public Identity[] getSigners() {
```

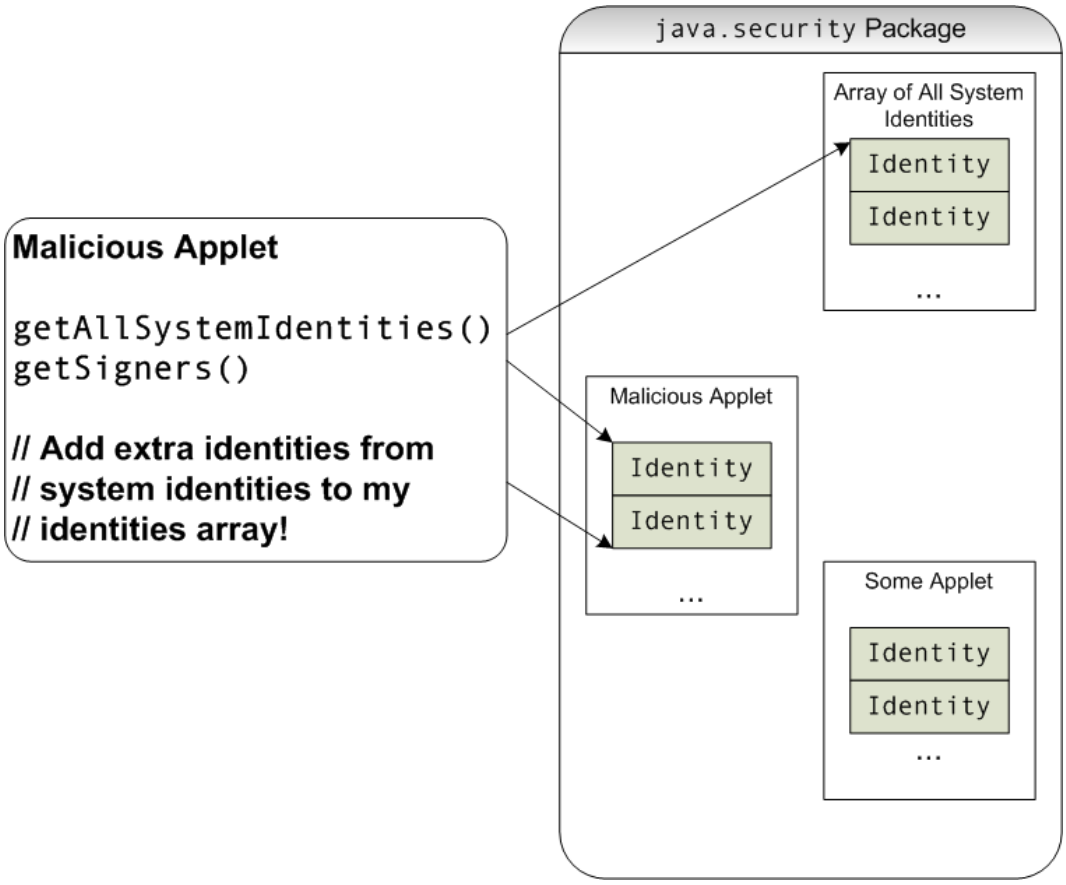

Figure 2.6: Aliased Identity Array Exposes it to Malicious Applets

```
  return signers;
}
```

As one can see, the method's implementation exposes the hidden array for modification. One of the possible fixes to this problem is to return a copy of the array instead:

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

While this fixes the hole exposing the reference to the array of signers (and thus preventing adding any new signers to the applet's signers array), there is no guarantee that there are no other holes left that expose these applet identities in another way.

An important point to note about this example is that none of the standard Java protection mechanisms seem to help. Such things as access modifiers and type abstraction are not relevant here, since the attack does not interact with `Identity` objects, it only needs to acquire references to them and copy those references [BV99].

Bokowski and Vitek's research introduces a novel and reliable way of checking for reference exposure by introducing *confined types*. The confined types ensure that all the references to the instances of classes that are confined originate from the objects in the same domain, which is taken to be a Java package. In the example presented in this section, it allows the programmers of the `java.security` package to confirm at compile time that none of the key data structures used in code signing escape the scope of their defining package [BV99]. Confinement and ownership are closely related concepts discussed later in this chapter.

2.2.5 The Pervasiveness of Aliasing

A large number of data structures rely on pointers. One of the first realisations on the path to become a proficient programmer is the idea that dealing with data in memory is about dealing with pointers, rather than values.

After presenting the examples of aliasing in this section, it is not my intent to discourage its use. Indeed, such a basic data structure as the circular doubly linked list shown in Figure 2.1 would not be possible without aliasing. Therefore, aliasing should not be fully avoided, but rather approached in a controlled and informed fashion. If a programmer wants a private field of their class to be free from aliasing problems, then there should be support in a language to allow them to do so.

The applet identities bug in JDK v1.1.1 was discovered nearly a decade ago [Gro97] and yet nothing changed in the way modern languages like Java and C# approach aliasing. This is an unfortunate state of affairs that may have come to be due to a large burden imposed by the existing alias control systems. Generic Ownership is an attempt to provide an easy way to manage aliasing, reaping its full benefits without any of the defects described in this section.

2.3 Encapsulation

Encapsulation is about preserving object boundaries. One is often taught in software engineering courses that information that is private to the object should not be exposed. A similar concept applies to collections of objects. A private pointer to an object that constitutes the state of another object should not be shared or given to anyone.

Since the realisation in the early nineties that objects do not provide encapsulation [Hog91], a large body of research has been developed to address this problem.

Early attempts [Hog91, Alm97, GTZ98, HLW+92] to tackle alias control came up with schemes enforcing *full encapsulation* which turned out to be unusable in practice. Later attempts [NVP98, Cla02, MPH99, BV99] provided more flexible ways to enforce alias protection, paving the way to practical applications of encapsulation in programming [AKC02, Boy04]. In this section, I will provide a brief overview of the alias protection research.

2.3.1 The Geneva Convention

In *The Geneva Convention on the Treatment of Object Aliasing*, Hogg et al. [HLW+92] categorised the aliasing research as four approaches: *detection*, *advertisement*, *prevention*, and *alias control*.

Detection is defined to be "static or dynamic (run-time) diagnosis of potential or actual aliasing". Advertisement is defined to be "annotations that help modularise detection by declaring aliasing properties of methods". Prevention is defined to be "constructs that disallow aliasing in a statically checkable fashion". Control is defined to be "methods that isolate the effects of aliasing".

Hogg et al. emphasised the futility of detection since static detection of aliasing is NP-hard [Lan92]. They considered advertisement and prevention to be worthwhile endeavours, but agreed that alias control is the most promising approach of them all.

In the rest of this section, I consider full alias encapsulation schemes (alias prevention), and flexible alias encapsulation schemes (alias control). These were developed in the decade that followed the Geneva Convention.

2.3.2 Uniqueness

Uniqueness is an approach to object encapsulation where some objects can be marked as having only a single incoming reference which can be checked by the language implementation. Uniqueness is the most basic way of guaranteeing encapsulation: a unique object is encapsulated within its sole referring object [BNR01]. This encapsulation implies that the surrounding object can depend upon the unique object for its private state without the aliasing-related concerns.

Unfortunately, uniqueness is not granular enough to allow it to be the sole solution to the problems caused by aliasing. Only some aliasing problems can be resolved by having unique references to objects. Allowing parts of the aggregate to refer to one another more than once is impossible using uniqueness and thus it is not feasible to provide aliasing control for collections and other aggregate objects.

2.3.3 Full Alias Encapsulation

Islands were proposed by Hogg [Hog91] and were the first proposal to constrain aliasing. The ideas from it have been used in every other proposal created since. The idea is to divide objects into "islands" connected together by "bridges". Islands are collections of objects that can be connected together without restriction via references that are only accessible to other objects outside an island via a *bridge* object. No object that is not a member of the island holds any reference to an object within the island apart from the bridge. The islands allow partitioning of object graphs into aliased regions so that the side effects are constrained to them and make formal reasoning about the whole system easier.

Balloons were proposed by Almeida [Alm98, Alm97] and concentrate on making it possible to share objects while preserving the privacy of their content. In this proposal, any class can be marked as a *balloon* which will then ensure that any instance of this class can only have a *unique* reference to it. Balloons can then either be *free* (unattached to any other object), or non-free (attached to some other object inside a program). Other objects referred to by such a balloon object become part of its balloon and cannot be accessed by objects outside the current balloon. Balloon types greatly simplify reasoning about objects being shared throughout a program.

Sandwich Types were proposed by Genius, Trapp, and Zimmermann [GTZ98] and improve on balloons to allow grouping of objects inside a single balloon into a separate place on the heap. Grouping objects inside a single balloon greatly improves locality and performance of programs utilising alias control. This is a good example of aliasing research causing advances in other areas of object-oriented programming such as performance.

All three of these full alias encapsulation systems partition the object graph (or heap) into regions — the objects in which cannot refer to each other, except via a controlled set of accessors. Full encapsulation provides a mechanism to ensure that certain parts of a program are inaccessible by other parts and thus do not suffer from aliasing defects. Unfortunately, these schemes prevent the vast majority of useful programs by disallowing something as simple as a collection from storing items that are referred to by other parts of a program. Implementing a linked list or a queue is a challenge in either the Islands or the Balloons approach.

2.3.4 Flexible Alias Encapsulation

Noble, Vitek, and Potter [NVP98] recognised the need for having a more granular control of object access. If you put an object into a container, you are forced to release it from a balloon or island that it is contained in — thus losing the guarantee of the absence of aliases to the object offered by full alias encapsulation schemes above. *Flexible Alias Protection* offers an alternative way to classify objects in terms of the kind of access allowed to them. The objects are divided into *representation* — which are private, and *arguments*

— which can be shared. It is possible to create container objects that would have their implementation hidden from the outside world, while making the elements stored in such containers accessible from the outside, without breaking the container's encapsulation.

In Flexible Alias Protection (FLAP) a selection of *aliasing modes* is developed that classifies objects into private (using mode **rep**), arguments (mode **arg**), unaliased (mode **free**), values (mode **val**), and other objects (mode **var**). The ground breaking ideas from FLAP can be applied to any programming language. Indeed FLAP has evolved into at least three established approaches to alias management: ownership types, universes, and confined types (all of which are described in the rest of this chapter). FLAP was not fully formalised in its original form and lacked a number of features: for example, none of the aliasing modes account for the owner of `this` — the current object.

2.3.5 Confinement

Every class in Java belongs to a package (if none is specified then it belongs to a so-called default package). Confinement came out of the work around Flexible Alias Protection and allows certain classes inside a domain — for example a package in Java or namespace in C++ — to be declared private to that domain. The language implementation can then guarantee that no instances of such *confined classes* are accessed by instances of classes outside the domain.

The idea of confinement is explored by Bokowski and Vitek [BV99] and further by Grothoff, Palsberg and Vitek [GPV01]. Confinement is viewed from the point of view of security. For example, in the JDK v1.1.1 aliasing case described above, the class `Identity` is defined inside the `java.security` package. The problem arises when instances of classes from different packages are allowed to directly access the `Identity` objects. Bokowski and Vitek demonstrate a solution [BV99] to the JDK v1.1.1 aliasing problem that uses an implementation of `Identity` confined to the `java.security` package while exposing a public version of it to the rest of the program. Since inside the package a new and secure version is used for storing the information, making it confined to the package renders it impossible to expose it.

The code in Figure 2.7 shows a more secure version of the solution to the JDK v1.1.1 problem that is verified using a confinement checker [GPV01] that confirms that a class `SecureIdentity` is confined to the `java.security` package. Confined types are utilised by Grothoff et al. [GPV01] to make it possible to write software libraries with formal guarantees that appropriate classes are going to be hidden from the outside. They distribute a tool called *Kacheck/J* that checks confinement constraints for Java programs.

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

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

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

// Inside some class in java.security package:
private SecureIdentity[] signers;

...

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

Figure 2.7: `SecureIdentity` Solution to the Aliased Identity Array Problem

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.

The package need not be the unit of confinement. For example, objects can also be confined to some general domain defined by the programmer (for example by explicitly enumerating the classes in each domain).

2.3.6 Ownership

Ownership also came out of the work around Flexible Alias Protection and allows objects to be grouped into an *ownership tree* in which parents *own* their children and every object in the tree outside the parent has to go through the parent object to access any of the children. This graph theoretical property of domination allows a large number of security constraints to be enforced in object-oriented programs paving the way to provide a complete, yet

flexible control of access to objects.

Every program has an underlying object graph [PNC98]. Objects in the graph are destroyed and created as a program runs. To be able to find unused objects, the garbage collector maintains a special selection of objects called the *root set*. The garbage, or no longer used objects, are defined as follows:

> An object is *garbage* **if and only if** there is no reference chain from some object in the root set to that object.

Since encapsulation is about controlling which objects are allowed to refer to which objects, it is natural to control access by controlling references in the object graph [Cla02]. Each object in the object graph has an owner which is another object that controls reference chains coming from other objects to the owned object. The concept of ownership is aligned with the concept of *dominators* in graph theory.

Ownership uses the concept of the root set to structure object graphs. If one posits a global root r through which all the objects in the root set of an object graph can be accessed. Then, *ownership* can be defined as follows:

> An object a *owns* another object b if all the paths from the root r to the object b go through a. In this case a is called the *owner* of b.

The implication of ownership is that no object outside the owner a is allowed to have a reference to b. Ownership allows us to structure an object graph into an implicit ownership tree. To clarify this concept, consider the example in Figures 2.8, 2.9, and 2.10. The first one gives a simple example of an object graph with root R. The second one shows an ownership tree constructed from the graph just before, note how it has exactly the same number of nodes but a lot fewer edges — since this time around edges are those of the ownership tree, rather than of the object graph. The third and final picture shows just the ownership tree.

Both ownership and confinement relate to an object's encapsulation with respect to aliasing. Ownership and confinement restrict the amount of aliasing possible within the object graph. Ownership examines a real picture of the object graph and states which objects are within other object's *shadow* — part of the object graph that can be reached by following references from the object [PNC98]. No one outside those objects underneath the owner in the ownership tree is allowed to have references to those inside.

Confinement looks at the collection of classes which will produce object graphs when executed. Confinement checkers can derive from the program's code what instances of which classes will be guaranteed to stay within the shadow of their defining package in the sense that no one outside those objects whose classes are in the same package will have references to them throughout all possible lifetimes of the program.

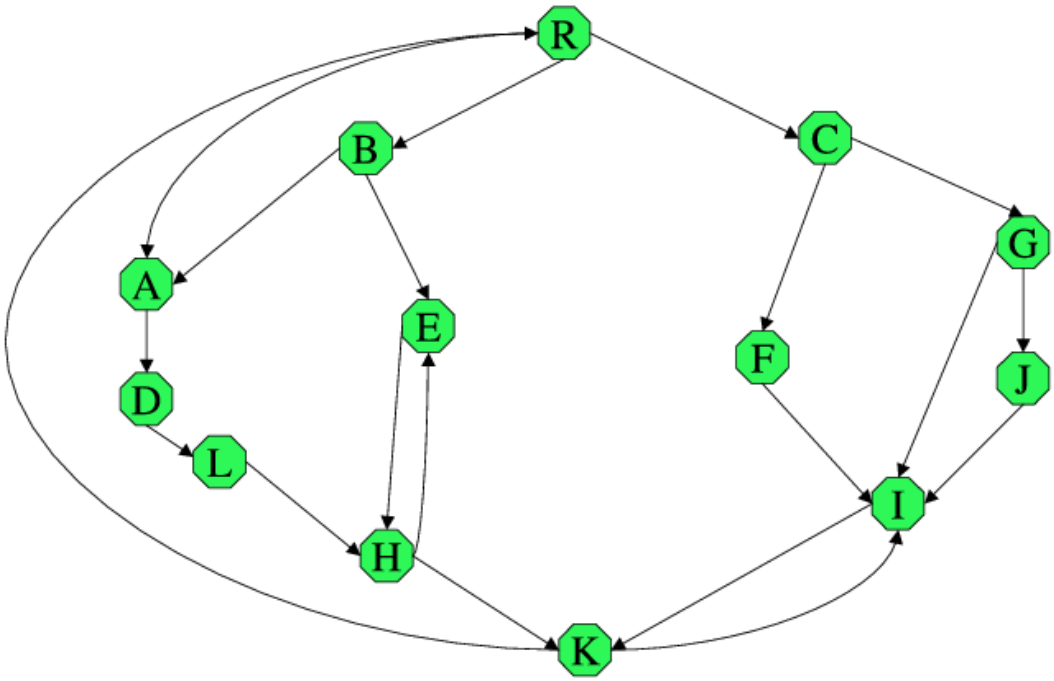

Figure 2.8: Ownership Tree Example (Part 1): An Object Graph

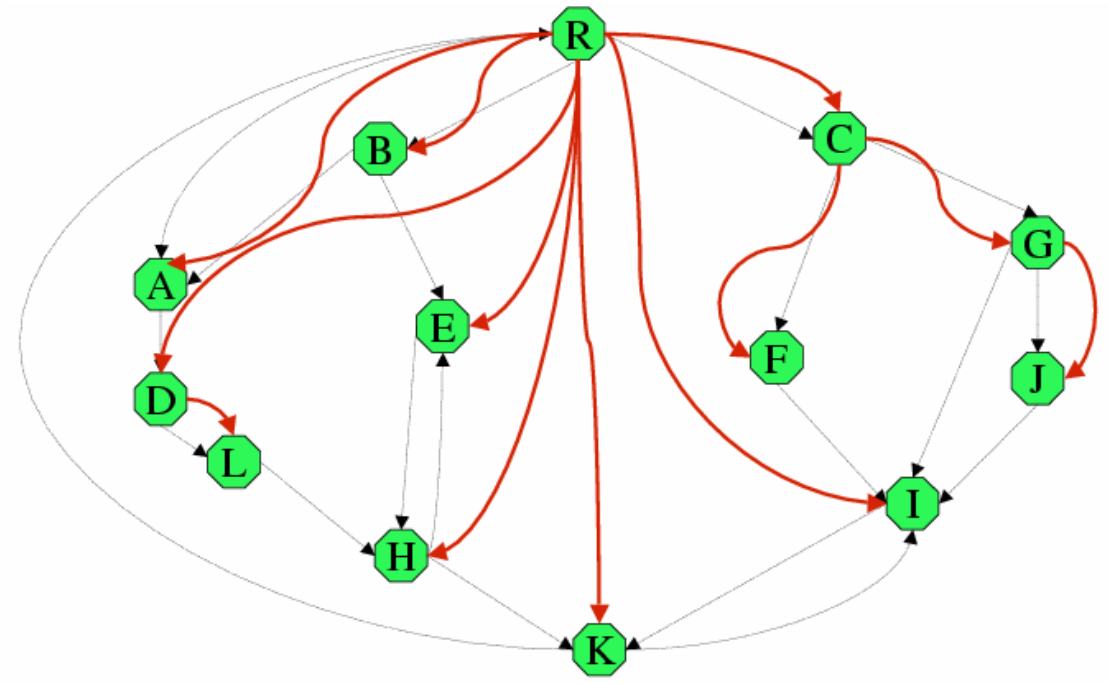

Figure 2.9: Ownership Tree Example (Part 2): An Ownership Tree and the Object Graph

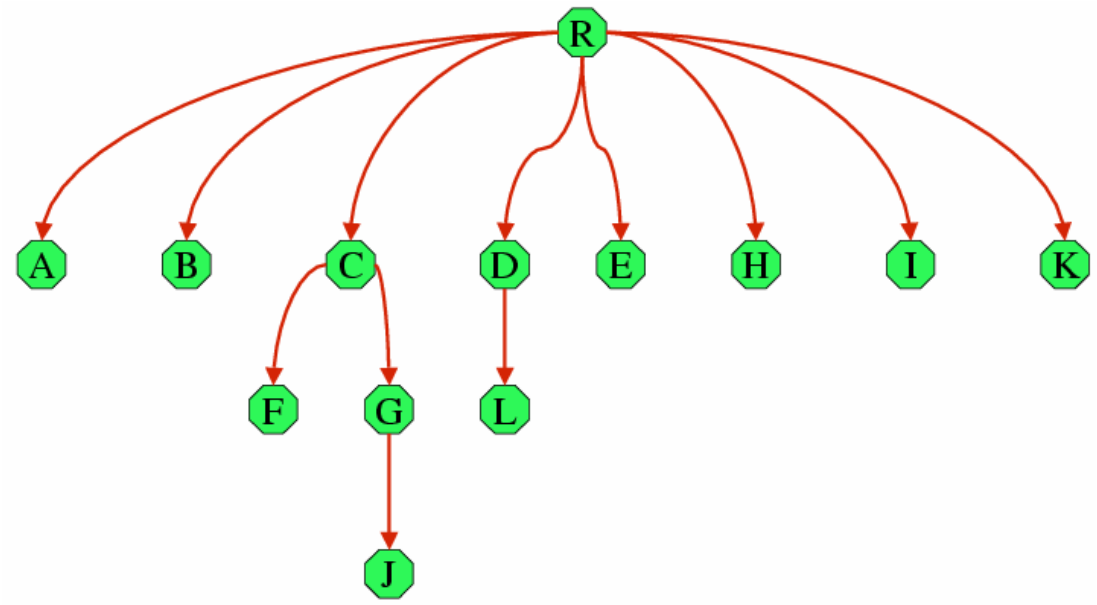

Figure 2.10: Ownership Tree Example (Part 3): An Ownership Tree

2.4 Ownership Schemes

Islands, ownership and confinement are all essentially forms of object encapsulation. All of these schemes are attempts to establish an encapsulation *boundary* that protects some objects contained *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 containment invariant which states that objects outside a particular boundary may not access objects contained 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 [NBT+03]. For example, in confined types [VB01], the unit of containment is a Java package: all the instances of public classes within that package form the encapsulation boundary; all the instances of package-scoped 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. A selection of relevant alias management schemes is discussed next.

2.4.1 Ownership Type Systems

Ownership (also referred to as *deep ownership* [CW03]) enforces the *owners as dominators* property over the program's object graph [Cla02]. If an object l *owns* another object l', then any path from the root of the object graph to l' has to go through l (l dominates l'). This property is crucial for many useful practical applications of ownership types.

In a system supporting (deep) ownership, an *inside* ($\prec$) relationship between different objects is defined [CPN98, BLS03]. The ownership invariant for such a system then states that if an object l refers to another object l', then l has to be *inside* the owner of l'. Chapter 5 proves the *ownership invariant* for Generic Ownership.

Ownership comes at a cost to program's expressiveness [CPN98], which is addressed in different ways by allowing local variables [CD02] or inner classes [BLS03] to have a much less restrictive access than inter-object references. For example, a system with classic deep ownership does not allow iterators for collections to access the internal representation of the collection object. In the Java Collections Library, this means that the implementation of the map will not be compatible with deep ownership since its iterator needs to have a direct pointer to the private table of map entries owned by map alone.

Alternatively, languages providing only *shallow ownership* (e.g., AliasJava [AKC02]) do not enforce an owners as dominators property. The ownership invariant for shallow ownership type systems guarantees only that if an object instantiates a class with an owner `This` then the new instance is accessible by the owning object and any other object that is passed the permission (owner `This`). Shallow ownership is strictly weaker than deep ownership. Allowing programmers to construct object graphs where domination is broken clearly admits more programs, but cannot provide the transitive containment guarantees offered by deep ownership. In addition to deep and shallow ownership, objects can be owned statically: for example by Java-style packages in Confined Types [VB01, ZPV06].

Rather than purely concentrating on the structure of the object graph, Lu and Potter develop an alias control system that separates the *object accessibility* concept (which object is allowed to access which object in the system) and the *reference capability* concept (object contexts used in ownership types) to provide a setting allowing more flexible type system with owner variance [LP06a]. The access context is introduced in addition to the object contexts and is used to formulate the containment invariant that structures the object graph based on the accessibility permissions, rather than ownership alone.

2.4.2 Universes: Read-Write vs Read-Only Access

Universes [MPH99] is a variant of an ownership type system that divides objects into those that are *read only* which can be shared throughout the object graph and those that are *read-write* and have to be private. Each object in the graph belongs to exactly one universe and has another universe associated with it that it *owns*. The owner of each universe is the

only object that has a read-write access to the objects inside the universe. Additionally, objects inside the same universe have read-write access to each other. Any other access between objects in different universes is always read-only. This invariant on the object graph is guaranteed by a sound type system and includes both references from fields inside objects and from local variables.

The motivation behind the development of the Universes type system was the need of alias control for modular verification of object-oriented programs. An important observation made by Peter Müller and Arnd Poetzsh-Heffter [MPH99] was that classical specification techniques based on pre- and postconditions, invariants, modifies-clauses, and history constraints cannot directly express representation encapsulation successfully captured by alias management systems [LM04]. The reason for this was that the invariants and similar specifications can depend on the state of the other objects, and modifications to the latter can affect the properties of the part of the program being reasoned about. Since such modifications cannot be controlled in the presence of unrestricted aliasing, formal reasoning about object-oriented programs becomes hard.

Universes have enjoyed considerable development over the last decade, ranging from becoming part of the Java Modelling Language (JML) [DM05] to applying them to Design Patterns [Näg06]. I do not discuss Universes further in my thesis, but a paper on Generic Universe Types [DDM07] suggests a combination of owners-as-modifiers with type genericity and provides valuable insights into the generic types and aliasing.

2.4.3 External Uniqueness

External Uniqueness is described by Clarke and Wrigstad [CW03]. Instead of making every object reference unique, an object graph is partitioned into groups of objects so that a reference to a group remains unique, while internal references between the objects in the group are unrestricted. Such groups of objects are similar to universes or ownership relationships partitioning the object graph, but permit a better balance between full restrictiveness (ownership) and making all the outside references read-only (universes).

In fact, external uniqueness is a small refinement of ownership rules that makes each externally unique edge also an edge in the dominator tree in the object graph. This allows external uniqueness to have the object graph topology guarantees required for ownership applications (for example in concurrency or persistence). It also avoids the excessive restrictiveness of ownership by allowing movement of externally unique references that effectively change the ownership of parts of the object graph.

2.4.4 Ownership in Concurrency and Persistence

Boyapati [Boy04] utilises ownership to detect data races and deadlocks in object-oriented programs with ownership annotations and allow safe persistent storage of objects [BR01, BLR02]. He developed a language called *SafeJava* [Boy04] and provided a full set of type systems formalising the operation of the language and its properties.

Since ownership, by the virtue of the owners-as-dominators property, provides guarantees about the structure of the resulting object graph, one can make assumptions when having a reference to a single object about all the objects that it owns. If one is to lock access to an object for concurrency purposes, with ownership one can be sure that all the objects it owns will not have any additional references. One can then proceed to make concurrent operations on a whole subset of the object graph.

Similarly, a whole subset of the object graph can be persisted using only the owner of the subset to detect whether any of the objects in this subset are used in the rest of the program or not. To resolve some of the expressivity issues with ownership, inner classes are allowed unhindered access to the enclosing classes. Therefore, constructs like iterators are allowed to access the inner details of the collection objects without breaking the ownership constraints.

2.4.5 Ownership in Software Architecture

Software Engineering quickly benefited from alias management with the development by Aldrich [AKC02] of ArchJava — a programming system that allows the description of the software architecture to be part of the program itself. Ownership types are integrated into the type system and the language implementation accompanying the project [1].

ArchJava provides a language called *AliasJava* that uses ownership to enforce appropriate access between objects inside the program. AliasJava allows the programmer to provide annotations to a program's code that mark appropriate objects as private. The AliasJava annotations are **owned** for objects that are only accessible by the creator, **lent** — a time-bounded ability to share the objects, **unique** — allowing uniqueness, and **shared** — for generally accessible objects.

A significant number of practical problems of getting ownership to work in practice were resolved during the development of AliasJava and a number of inference schemes for ownership annotations were proposed. AliasJava does not provide a dominator tree-like structure for the object graph, thus only providing permissions to access particular objects, rather than whole object graph structure guarantees — as utilised for concurrency and persistence described above.

Aldrich, Kostadinov, and Chambers took the concepts emerging from the ArchJava

[1] `http://www.archjava.org/`

project back to the ownership type system research by introducing a novel concept of *ownership domains* [AC04]. Ownership domains take the next step in the flexibility of ownership type systems by allowing explicit ownership permissions to be manipulated as part of the programming language.

2.4.6 Preserving Object Invariants

The ownership and universes work also extends to object invariants [LM04] providing the classic verification techniques with enough power to address aliasing problems. This approach results in more flexible proposals such as the concept of *friendship* — allowing invariants to span shared state [BN04b]. Friendship is similar to the C++ friendship between classes as it allows one class to give access privileges to another class by linking their invariants. Spec# [BLS04] is a programming language and implementation distributed by Microsoft Research that has support for some of the object reference invariants.

2.5 Overview of the Formal Foundations

In the remainder of this chapter, I overview and refer to the relevant papers about the selected object-oriented languages research that is essential to understanding the concepts developed in Generic Ownership.

When it comes to introducing the ideas described in this chapter into modern programming languages, one needs to be sure that such a move will not break the existing language. A new feature can make it possible to have programs that no longer can be predictably executed on a computer.

Every language can be formalised as a *type system* that gives the syntax of the language and a set of rules describing if a given expression that is syntactically correct is also *well typed* [DD85]. Well-typedness means that all the rule conditions for a particular expression are met. A number of *reduction rules* is also supplied showing how the valid language expressions reduce during the execution of a program.

Once the rules for a language are formulated, two traditional type system properties can be proved: *type preservation* and *progress*. Type preservation states that any expression of a given type, when reduced to another expression, will still have the same type. For example, if an object-oriented program with a local variable of type `Number` accepts the result of a method call returning `Integer` (which is a subtype of `Number`), one wants the expression to be valid. Progress states that any valid expression in the language being defined will be either a value, or it will be possible to apply one of the reduction rules. In other words, well behaved programs should never get "stuck". Together, type preservation and progress are known as *type soundness* and a language with a type system that is proven

to be sound will have a guarantee that well typed programs will be able to execute in a predictable fashion on any correct implementation of the language.

I use a syntactic approach to type soundness pioneered by Wright and Felleisen [WF94]. In this section, I provide an overview of the formal techniques utilised in my thesis. I start by describing Featherweight Generic Java — capturing the functional substrate of Java in a clean and simple type system in Subsection 2.5.1. I then describe how FGJ can be extended to support imperative features of Java following Pierce [Pie02] in Subsection 2.5.2. Finally, I briefly overview the ownership types systems, providing the theoretical background on ownership used in Generic Ownership, in Subsections 2.5.3, 2.5.4, and 2.5.5.

2.5.1 Featherweight Generic Java (FGJ)

Featherweight Generic Java (FGJ) [IPW01a] is a non-algorithmic type system describing a subset of the Java language complete with syntax, typing, and reduction rules. It has been proven to be sound and is utilised widely by researchers to define languages formally. FGJ has only five expressions: variable, new class instance creation, method call, field access, and type cast. It does not have any imperative features of Java, such as field update or heap, thus capturing the functional aspects underlying the language. This simplification allows for a very simple and compact type system, paving the way to provide a simple tool for reasoning about Java programs.

Figure 2.11 is taken from the original FGJ paper by Igarashi et al. [IPW01a] and shows

```
1  class A extends Object {
2    A() { super(); }
3  }
4
5  class B extends Object {
6    B() { super(); }
7  }
8
9  class Pair<X extends Object, Y extends Object>
10   extends Object {
11
12     X fst;
13     Y snd;
14     Pair(X fst, Y snd) {
15       super(); this.fst = fst; this.snd = snd;
16     }
17     <Z extends Object> Pair<Z, Y> setfst(Z newfst) {
18       return new Pair<Z, Y>(newfst, this.snd);
19     }
20 }
```

Figure 2.11: An Example of a FGJ Program

```
T ::= X  |  N
N ::= C<T̄>
L ::= class C<X̄ ◁ N̄> ◁N {T̄ f̄; K M̄}
K ::= C(T̄ f̄) { super(f̄); this.f̄=f̄; }
M ::= <X̄ ◁ N̄> T m(T̄ x̄) { return e; }
e ::= x | e.f | e.m<T̄>(ē) | new N(ē) | (N)e

X ranges over the type variables.
N ranges over the nonvariable types.
```

CT class table: a mapping from class names C to class declarations L.
Δ type environment: a mapping from type variables to nonvariable types.
Γ type environment: a mapping from variables to types.

Figure 2.12: FGJ Syntax and Environments

an example of an FGJ program. This program looks very similar to a typical Java 5 program except that it uses only the five allowed FGJ expressions. The typing environment and execution of the FGJ programs are modelled in great detail by a sound FGJ type system.

Figure 2.12 shows the syntax of the FGJ language and the environments used in the language definition. An FGJ program consists of class declarations, followed by a program expression. Class table CT contains these class declarations, while the environments Δ and Γ accumulate the type-related information used by the rules during the type checking of FGJ programs.

At the core of the FGJ type system is the GT-CLASS rule describing when a class declaration is considered valid in the FGJ language:

$$\frac{\overline{\mathtt{X}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{N}}, \mathtt{N}, \overline{\mathtt{T}}\ \mathtt{OK} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN}\ \mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}> \quad \mathtt{K} = \mathtt{C}(\overline{\mathtt{U}}\ \overline{\mathtt{g}}, \overline{\mathtt{T}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}}{\mathtt{class}\ \mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}> \triangleleft \mathtt{N}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}}$$

This rule is a good example of a typical type rule in most type systems. It has a statement under the bar showing a full class declaration of a class C with generic type parameters $\overline{\mathtt{X}}$ (where the bar means that it is a list) with bounds $\overline{\mathtt{N}}$. Class C extends another class N and has a number of fields $\overline{\mathtt{f}}$ of corresponding types $\overline{\mathtt{T}}$, constructor K and a list of method declarations $\overline{\mathtt{M}}$. FGJ considers such a class declaration OK if and only if the four statements above the bar are all true.

The statements above the bar check the well-formedness of types $\overline{\mathtt{N}}, \mathtt{N}, \overline{\mathtt{T}}$, look up the fields of the superclass N, check the declarations of all the methods $\overline{\mathtt{M}}$, and the declaration of the constructor K. If all of these four statements check out using the other rules in the FGJ type system, then the class declaration is considered valid.

The total number of rules describing the properties and the execution of the FGJ language is only two pages long, which makes it possible to work with such a small

language and extend it with a variety of other features that need to be investigated as to their effect on the soundness of a Java-like language. The original language formulated by Igarashi et al. [IPW01a] was Featherweight Java (FJ), which was then extended with generic types into Featherweight Generic Java (FGJ) demonstrating that type genericity can be introduced in a language like Java safely and allowing the researchers to experiment with the different ways of handling generic type information.

I present the rules and the type soundness theorem for FGJ in Appendix A at the end of this thesis for the reader's convenience.

2.5.2 Imperative FGJ

FGJ is a functional language that does not have a heap or object identities or references between objects. To be able to reason about aliasing and ownership one needs to have these features. Lack of object graph modelling information is the main driving force behind extending the FGJ type system with heap, objects and references between them, as well as field updates, local variables, and `null`. The Featherweight Generic Ownership (FGO) type system underlying Generic Ownership, presented in Chapter 5, has all of these features making it possible to reason about ownership in a suitable setting. FGJ can be extended with imperative features following standard type theory techniques [Pie02].

For example, a heap can be modelled by locations and store. Each object is created at a unique location which is then used in a store to map from each location to the ones pointed at by the location's object's fields. There is also an environment storing the map from locations to their type (the class that is instantiated by the object at a particular location). Each operation involving changes to the store, such as object creation or field update has to describe the change to the store as part of the reduction rule. Hence, each reduction rule not only contains the expressions $\mathtt{e}$ and $\mathtt{e}'$ that are being reduced from one to another but also the stores S and S' that correspond to each expression. While in FGJ, all the information about a state of the program is contained in an expression $\mathtt{e}$, in an imperative extension, the program state is expressed in the pair of expression and store: $\mathtt{e}, S$.

2.5.3 Ownership Types

Ownership was first formalised as Ownership Types in by Clarke et al. [CPN98]. The original ownership type system proposed making ownership information part of a type. Ownership types are basically types annotated with *object contexts* (or owners) — similar to the aliasing modes used in the Flexible Alias Protection. For example, owner **this** replaces **rep**, owner **world** replaces the absence of **rep**, and owner **owner** accounts for the owner of `this`. Therefore, it is possible to provide a complete set of expression typing rules and reduction rules that also check that ownership of objects is preserved. By proving

the type system sound, the authors demonstrate that if one is to mark any field or variable as owned by the enclosing instance, then no reference from objects not owned by the same enclosing instance is possible at run-time.

While mentioning the possibility of treating the ownership information that is part of types in the same way as type genericity, this facility was omitted for simplicity. Another concession to the simplicity to clarify the concepts involved in ownership includes the omission of subtyping. Both omissions were remedied in later ownership type systems by the same authors [Cla02, CD02]. On the other hand, the essential properties of ownership are present, such as the static visibility check that ensures that instances owned by the current object are only manipulated using variable `this` and none other.

The original ownership types system was further expanded and amended by Clarke in his PhD thesis [Cla02]. As part of his work he develops the concept of a *containment invariant*: a theorem that can be proved about a language to show that it provides ownership guarantees. In this thesis, I prove two separate containment invariants — one for confinement and one for ownership — to demonstrate that Generic Ownership safely supports all of these features.

2.5.4 Confined Types

Zhao et al. [ZPV06] provide a sound type system based on FGJ that shows how to support confinement and generic types. They start with an extension to Featherweight Java that supports confinement and later extend the resulting Confined Featherweight Java with support for generic types. They formulate a *confinement invariant* to demonstrate that their language supports confinement. The confinement invariant is expressed 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 class inside which a method is defined.

2.5.5 Ownership Types and Effects Systems

The ownership type system most closely related to Generic Ownership is presented by Clarke and Drossopoulou in their work to provide support for ownership types and effects [CD02]. While not taking the effects treatment into account, Generic Ownership adopts the owner nesting check that at class declaration time requires all the owner parameters in the class header to be *outside* the owner of the current class instance. Then, at every class instantiation time, the owners involved are checked yet again to be outside the main owner of the type. Together the latter allows for the proof of the containment invariant to assume that owner nesting is preserved at all times. Most importantly, the language presented in the paper (Joe) supports subtyping — as opposed to the earlier ownership

type systems. Generic Ownership adopts the Joe's treatment of subtyping that requires the owner of the class to always be preserved. The authors of Joe chose to omit the treatment of type genericity that is presented in this thesis.

Chapter 3

Generic Ownership

Contents

Genericity and Ownership are two language mechanisms that, in different ways, allow programmers to make the intentions behind their code more explicit. These capabilities can provide programmers with more support, typically by detecting errors statically, at compile time, that would otherwise only be detected (or even remain undetected) once a program is run. This chapter guides the reader through the ideas underlying Generic Ownership.

Section 3.1 and Section 3.2 portray the state of the art in the generics and ownership research fields. Section 3.3 discusses a straightforward way of combining ownership and genericity. Section 3.4 presents a much improved approach called *Generic Ownership*. Section 3.5 overviews the expressiveness of Generic Ownership. Section 3.6 discusses Generic Ownership's confinement support. Section 3.7 presents manifest ownership — a novel technique of preserving ownership information for unannotated classes. Section 3.8 describes Ownership Generic Java's class hierarchy. Section 3.9 delves a little deeper

into Ownership Generic Java's language design and Section 3.10 provides a number of motivating examples.

3.1 Generics: State of the Art

Genericity allows us to use type parameters to give a better description of the type of a variable one is dealing with. Genericity allows more sensible collection types (e.g., of `Node`s rather than *anything*), better compile-time error detection, and more readable and reusable code.

The code in Figure 2.2 exhibits a number of well-known weaknesses [BOSW98, KS01]. One prominent weakness is that the code relies upon subtyping to store objects of various types within the `Map` itself and within the `Vector` implementing the `Map`. That is, both the `Map` and the `Vector` simply store `Object`s. Considering the `Vector`, the `Map` may store `Node`s into the `Vector` object, because Java `Vector`s hold any kind of `Object` and a `Node` is a kind of `Object`. However, when getting a `Node` from the `Vector` (via an iterator in this case) the code requires a dynamic cast to ensure the object returned actually is a `Node`.

Malicious or buggy code, for example, could insert some other kind of object into that `Vector`, causing this cast to fail at runtime and triggering an exception.

This is an old problem, and the solutions are equally old, dating back to the mid-1970s [Mil78]. The class definitions must be made generic so that particular instances of the classes can be created for particular argument types. Java now supports generic types [Sun05, BOSW98]. Figure 3.1 presents a version of the `Map` class written using genericity, in Java 5 syntax.

While being widely adopted and appreciated in functional languages, and in some object-oriented languages such as Eiffel, Ada, Modula-3, and C++, genericity is only now on the edge of acceptance in other popular object-oriented languages such as Java and C#. There have been a range of proposals to add genericity to Java [AFM97, MBL97, OW97, BOSW98]; one of which, GJ [BOSW98], has been adopted as the basis for the latest release of Java (Java 5) [Sun05]. C# has developed along similar lines [KS01].

Comparing Figures 2.2 and 3.1 illustrates both the advantages and disadvantages of generic types. Regarding the advantages, the types of objects stored in the `Vector` or `Map` can now be preserved when they are returned, so there is no need for a typecast when objects are removed from the `Vector`. Method declarations can also carry more information, using generic type parameters like "`Key`" or "`Value`" instead of "`Object`". As a result, only objects of the right types can be stored into `Map`s and `Vector`s, while the attempts to store the wrong types of objects will be detected at compile time.

The main disadvantage is that Figure 3.1 is more syntactically complex than Figure 2.2.

```
1  public class Map<Key, Value> {
2    private Vector<Node<Key, Value>> nodes;
3
4    void put(Key key, Value value) {
5      nodes.add(new Node<Key, Value>(key, value));
6    }
7    Value get(Key k) {
8      Iterator<Node<Key, Value>> i = nodes.iterator();
9      while (i.hasNext()) {
10       Node<Key, Value> mn = i.next();
11       if (mn.key.equals(k))
12         return mn.value;
13     }
14     return null;
15   }
16 }
17 class Node<Key, Value> {
18   public Key key;
19   public Value value;
20
21   Node(Key key, Value value) {
22     this.key = key; this.value = value;
23   }
24 }
```

Figure 3.1: `Map` Class Example (Java 5)

The class definitions in the generic version declare formal generic type parameters, and then class instantiations must provide actual values for those parameters, which results in types like "`Vector<Nodes <Key, Value>> nodes`" rather than "`Vector nodes`".

3.2 Ownership: State of the Art

To return to the example, considering the basic `Map` implementation from Figure 2.2, the `nodes` field containing the `Vector` in the `Map` is declared as private. Java will ensure that the field can only be accessed from within the `Map` class. This is done because the `Vector` is an internal part of the implementation of the `Map` class and should not be accessed outside. Inserting or removing elements from the `Vector`, or perhaps acquiring but not releasing its internal lock could break the invariants of the `Map` class and cause runtime errors.

Unfortunately, the name based protection used in Java and most other programming languages is not strong enough to keep the `Vector` truly private to the `Map`. An erroneous programmer could insert a public method that exposed the `Vector`:

```
public Vector exposeVector() {return nodes;}
```

with no objection from the compiler. Any resulting errors will be subtle, possibly appearing at runtime long after the execution of the `exposeVector` method, and thus be difficult to identify and resolve. These kinds of errors have been identified as occurring in many Java libraries [Pug07] and have caused significant problems for language security mechanisms [VB01].

Ownership types [CPN98, CD02, MPH99, AKC02, BLS03] protect against aliasing errors by allowing programmers to restrict access to objects at runtime, rather than just the names of variables used to store them. The key idea is that *representation* objects (like the `Map`'s `Vector`) are nested and encapsulated inside the objects to which they belong (i.e. the `Vector` belongs to the `Map` and the `Map` owns the `Vector`). Clarke [Cla02] formulated an ownership invariant: there can be no incoming references that bypass owners. This means that an object cannot refer to a second object directly, unless the first object is itself inside the second object's owner. Because this nesting (and thus the protection) is transitive, it is called *deep* ownership [CW03]: if an `Array` is part of a `Vector`'s representation, the array should be owned by the vector and is thus nested inside both the vector and the map. Enforcing encapsulation via deep ownership has many practical and theoretical applications [BDF+03, LM04, Boy04, AC04].

Figure 3.2 gives an example of the `Map` class using ownership types. The syntax used is proposed by Boyapati [Boy04, Figure 2.7]. Comparing Figure 3.2 with Figure 2.2 and Figure 3.1 illustrates both the strengths and weaknesses of ownership types. The most obvious difference is the presence of a range of *ownership parameter* definitions such as "<mOwner, kOwner, vOwner>". The class declaration includes these three new [ownership] parameters: `mOwner` represents the ownership of the instance of the `Map` class, and "`kOwner`" and "`vOwner`" describe the ownership of the keys and values that will be stored in a map. These parameters are then instantiated as the types are used, for example when a `Node` is created within the `put` method of `Map`. Ownership systems adopt the approach where the owner of all the keys (or values) in the ownership map is the same: just like the class of all the keys (or values) in the generic map is the same.

Ownership parameters can be instantiated via the keyword "`this`", which ensures that the current object (the object usually denoted "`this`" in Java) *owns* the object being declared [CPN98]. The `Vector` object is marked in this way as being owned by the `Map`, for example, so with the help of additional checks any attempt to access or pass the `Vector` object outside the `Map` object will be detected and prevented at compile time. Code such as the `exposeVector()` method will be unable to cause any damage by breaching encapsulation. The difference between `this` and `mOwner` is that `mOwner` is the owner of the `Map` instance, which one does not necessarily know about, and `this` is the `Map` instance itself, since it is allowed to own objects in its own right.

The ownership parameters carry ownership information around the program, so that the ownership of the keys and values can be maintained outside the `Map`. For example, the

```
1  public class Map<mOwner, kOwner, vOwner> {
2    private Vector<this, this> nodes;
3
4    void put(Object<kOwner> key,
5             Object<vOwner> value) {
6
7      nodes.add(new Node<this, kOwner, vOwner>
8                        (key, value));
9    }
10
11   Object<vOwner> get(Object<kOwner> key) {
12     Iterator<this, this> i = nodes.iterator();
13     while (i.hasNext()) {
14       Node<this, kOwner, vOwner> mn =
15         (Node<this, kOwner, vOwner>) i.next();
16
17       if (mn.key.equals(key)) {
18         return mn.value;
19       }
20     }
21     return null;
22   }
23 }
24 class Node<mapNodeOwner, kOwner, vOwner>
25   public Object<kOwner> key;
26   public Object<vOwner> value;
27
28   Node(Object<kOwner> key,
29        Object<vOwner> value) {
30     this.key = key;
31     this.value = value;
32   }
33 }
```

Figure 3.2: `Map` Class Example (Ownership Types for Java)

ownership of the keys and values may be specified by each instance of the `Map` class, but using the `kOwner` and `vOwner` ownership parameters, the fields that will store keys and values inside the subsidiary `Node` objects will have the correct ownership for these fields.

The main disadvantage of using ownership types is similar to that of generic types which adds syntactic complexity as a result of ownership parameters. The code in Figure 3.2 also has all the type-related problems of the "straight" Java code: the problems that are addressed by genericity. This code relies on subtyping to store different types of objects, and requires type casts when objects are removed from the `Vector` (or subsequently from the `Map`). Although the code may look generic, all the declared types are simple Java types, such as `Object`, with the problems that entails. While the `Vector` stored in the `nodes` field can no longer be exposed outside of the `Map` instance that owns it, malicious

```
1  public class Map<mOwner, kOwner, vOwner>
2    [Key extends Object<kOwner>,
3     Value extends Object<vOwner>] {
4
5    private Vector<this, kOwner, vOwner>
6      [Node<this, kOwner, vOwner>[Key, Value]] nodes;
7
8    void put(Key key, Value value) {
9      nodes.add(new Node<this, kOwner, vOwner>
10       [Key, Value](key, value));
11   }
12
13   Value get(Key key) {
14     Iterator<this>[Node<this, kOwner, vOwner>
15       [Key, Value]] i = nodes.iterator();
16
17     while(i.hasNext()) {
18       Node<this, kOwner, vOwner>
19         [Key, Value] mn = i.next();
20
21       if (mn.key.equals(k)) {
22         return mn.value;
23       }
24     }
25     return null;
26   }
27 }
28 class Node<mapNodeOwner, kOwner, vOwner>
29   [Key extends Object<kOwner>,
30    Value extends Object<vOwner>] {
31
32   public Key key; public Value value;
33   Node(Key key, Value value) {
34     this.key = key; this.value = value;
35   }
36 }
```

Figure 3.3: `Map` Class Example (Combined Generic and Ownership Types)

or buggy programming within that class can once again break code by directly inserting incorrect types into the `Vector`.

3.3 Combining Ownership and Generics

There are two separate but similar techniques that manage which objects may be accessed by which types, fields, or expressions. Genericity manages the object accesses using compile-time types, and ownership manages the object accesses using compile-time object

structures. These two mechanisms appear to be orthogonal, raising the question:

How can they both be included within a single programming language?

Figure 3.3 repeats the `Map` example using a hypothetical language separately supporting both genericity with parameters marked with square braces (`[` and `]`) and ownership with parameters marked with angle braces (< and >). The language syntax is based on Boyapati's [Boy04, page 29] and resembles the original Flexible Alias Protection proposal [NVP98].

The code in Figure 3.3 compared with previous figures, now has both type and ownership parameters, each taken from their respective languages. Types can be instantiated for keys and values, as in the type generic system. Instantiating these types removes the reliance on subtyping and the associated type casts, which are an error-prone, run-time mechanism that type genericity alleviates. For the ownership system, objects can be tagged as owned by `this`, ownership can be recorded via owner parameters, and therefore any method exposing an owned object would be detected and prevented.

The syntax required to implement both ownership and genericity separately means that this code is significantly more complex than any of the other examples. So complex, arguably, that it would be unusable in practice, because both classes require *five* ownership and type parameters. Some relief can be provided by limited type inference of these parameters but the inference rules become too much of an unnecessary burden that will hurt the likelihood of getting ownership into a language like Java.

3.4 Generic Ownership

Generic Ownership is a new linguistic mechanism that combines genericity and ownership into a single simple language. As in the hypothetical example of Figure 3.3, Generic Ownership provides the benefits of both type and ownership parameterisation: catching all the errors and avoiding all the bugs that the generic and ownership languages do individually. Unlike that example, Generic Ownership treats ownership and genericity as one single aspect of language design, and so code using Generic Ownership is no more syntactically complex than code that is either type-parametric or ownership-parametric. The key technical contribution of Generic Ownership is that it treats ownership as an additional kind of generic type information. This means that existing generic type systems can be extended to carry ownership information with only minimal changes.

Figure 3.4 revisits the `Map` example for the last time. This time, it is written in the new language design, Ownership Generic Java (OGJ). The code in Figure 3.4 is type-generic, as definitions of fields in `Node` and methods everywhere use generic types such as `Key` and `Value` rather than plain class types such as `Object`. The code is also ownership-generic,

```
1  public class Map<Key extends Object<KOwner>,
2                   Value extends Object<VOwner>,
3                   Owner extends World> {
4    private Vector<Node<Key, Value, This>,
5                   This> nodes;
6
7    public void put(Key key, Value value) {
8      nodes.add(new Node<Key, Value, This>(key, value));
9    }
10
11   public Value get(Key key) {
12     Iterator<Node<Key, Value, This>, This>
13       i = nodes.iterator();
14
15     while (i.hasNext()) {
16       Node<Key, Value, This> mn = i.next();
17         if (mn.key.equals(key)) {
18           return mn.value;
19         }
20       }
21     return null;
22   }
23 }
24
25 class Node<Key extends Object<KOwner>,
26            Value extends Object<VOwner>,
27            Owner extends World> {
28   public Key key;
29   public Value value;
30
31   public void Node(Key key, Value value) {
32     this.key = key; this.value = value;
33   }
34 }
```

Figure 3.4: `Map` Class Example (Ownership Generic Java)

as every class has an extra type parameter that represents the object's owner. This parameter is placed *last*, it is typically named `Owner`, and it is called the *owner parameter*. When creating an object, one can mark it as owned by the current object `this` by instantiating the new object's owner parameter with the owner `This`. All OGJ classes descend from a new parameterised root `Object<O>` that declares an owner parameter, and all subclasses must invariantly preserve their owner parameter. Owners descend from a separate root `World`, which also acts as a second owner constant, meaning that access to an object is unrestricted.

Recall, for example, the type of the `nodes` vector private to the `Map` object considered in Section 3.2:

```
public Vector exposeVector() { return nodes; }
```

This code is not valid in OGJ, since every type has an owner parameter and casting to raw types (types with their generic parameters omitted [IPW01b]) is prohibited. The type of the field `nodes` in Figure 3.4 has an owner parameter `This`. If one tries to give a return type of `exposeVector` method an owner parameter `This`:

```
public Vector<Node<Key, Value, This>, This>
  exposeVector() { return nodes; }
```

then this code will be valid OGJ, but the method can only be called if the result can be assigned to something that is a supertype of `Vector<..., This>`. Since OGJ preserves owners over subtyping, any valid supertype of the return type will have to have an owner `This`. Hence the return type will only typecheck if the method is called from *the same instance* of `Map`. In other words, this `exposeVector` method *cannot* expose the vector. If we attempt to declare `exposeVector` with a return type having any other owner parameter, then the return type and the return value's (`nodes`) type will not be assignment compatible in OGJ, as their owner parameters will be different.

Note that the owner `This` is very similar to the variable `this` used in Java to denote the current instance of the class. Hence, using `This` in different instances will refer to different owners, just as using `this` in different instances will refer to different instances of what may be the same class. The mechanics of ensuring that `This` owner is instance specific is described in detail in Chapter 5 in the discussion of the *this* function.

3.5 Expressing Ownership with OGJ

Comparing Figure 3.4 with Figures 2.2, 3.1, 3.2, and 3.3 shows that the code in Figure 3.4 is slightly more complex than the individual type genericity or ownership examples, but simpler than the straightforward combination in Figure 3.3. The declaration of the `Map` class has only *three* parameters (the same as in Figure 3.1 and Figure 3.2). Furthermore, the type of field `nodes` in Figure 3.4 is more readable:

```
Vector<Node<Key, Value, This>, This> nodes;
```

rather than:

```
Vector<this, kOwner, vOwner>
  [Nodes<this, kOwner, vOwner>[Key, Value]] nodes;
```

presented in Figure 3.3.

Every OGJ class has a distinguished owner parameter, hence the bounds of formal generic parameters (e.g., `Key extends Object<KOwner>`) must be declared with a *placeholder* owner parameter (`KOwner` in this case). Programmers are *not* required to declare placeholder parameters: rather they are bound implicitly, with very similar semantics

to Java wildcards [TEH+04, LP06a]. Within the scope of the type parameter declaration (a generic type or method) the bound type variable (e.g., `Key`) ranges simultaneously over type and owner. Owners can also be passed explicitly (generally as arguments to formal parameters bounded by `World`) — indeed, the distinguished owner parameter is just a special case of this.

The implicit binding of owner parameters — and, more importantly, the combination of ownership into types — reduces the number of formal arguments required by Generic Ownership. Because the owners are bounded implicitly, programmers do not need to write code such as:

```
public class Map<KO extends World, VO extends World,
  Key extends Object<KO>, Value extends Object<VO>,
  Owner extends World> { ... }
```

The implicit binding is generally sufficient: we find that ownership and type parameters are hardly ever used independently. Chapter 5 describes the *placeholderowners* function that performs the binding behind the scenes by adding the missing owner type parameters as new owner parameters bounded by `World`.

If a programmer wishes to produce a specialised version of a map class (`MyMap`) that is not type generic — say it can store only names and addresses — but which needs to be ownership generic, additional type parameters (`NameO`, `AdrO`) bounded by `World` can be declared explicitly to carry the ownership for names and addresses.

```
class MyMap<NameO extends World, AdrO extends World,
  Owner extends World> {
    void    put(Name<NameO> n, Address<AdrO> a);
    Address get(Name<NameO> n);
    List<Name<NameO>, World>
      getAllPeopleOnThisStreet(String s);
}
```

Given that the `MyMap` class requires three parameters anyway, standard OGJ style is to declare two bounded generic parameters, and always instantiate those parameters with the same types as their bounds (although, for ownership polymorphism, with different owners):

```
class MyMap<MyName extends Name<NameO>,
  MyAddress extends Address<AdrO>, Owner extends World> {
    void    put(MyName n, MyAddress a);
    Address get(MyName n);
    List<MyName, World>  getAllPeopleOnThisStreet(String s);
}
```

The only disadvantage of the generic ownership version of the `MyMap` class compared to the ownership version is the same as of that of the generic version in Java 5. Inside the generic ownership version of `MyMap` a new instance of the `MyName` class cannot be created, since it is a type variable which is erased before run-time. In the ownership version of `MyMap` it is possible to create a new instance of `Name<NameO>`. In the generic ownership version or even generic version of `MyMap`:

```
class MyMap<MyName extends Name, MyAddress extends Address> {
  ...
}
```

Java 5 does not support instantiation of the type variables (e.g. `new MyName()` in the example above) or assigning a new instance of `Name` to `MyName`. If the language does not implement erasure [BOSW98] like C# or the upcoming versions of Java, then this will not be a problem since the type of `MyName` will be defined at run-time.

Finally, to live up to the full potential of generics in Java 5, Generic Ownership allows owner parameters to be mixed with generic method parameters. Owners on methods allow more granular control of what access each method has to other objects, and they can be utilised usefully for more granular alias control with little overhead. For example, the following method can create a new `Name` object even though the class to which the `addNewName` method belongs might not have access to owner `NameO`:

```
<NameO extends World>
  void addNewName(String name) {
    this.put(new Name<NameO>(name), null);
}
```

Owner method parameters come up in practice when implementing OGJ's static methods. Since static methods do not belong to any instance of a class, they cannot have generic owner parameters coming from the class declaration. Without owner parameters for methods, creation of new classes with particular owners and implementation of static methods would not be easily possible.

In summary, these examples show how OGJ can provide independent ownership and type genericity — with five separate parameters — and ownership genericity without type genericity — with "naked" owner parameters, or by using parameters instantiated at their type bounds. Type parametricity can be provided without ownership parametricity by either supplying ownership constants (e.g., `World`) at point of use, or using manifest ownership (described below) to fix an owner for all instances of a class. An important observation is that due to the Generic Ownership requirement that owners should stay unchanged during the subtyping, Generic Ownership does not support owner parameter variance. Interaction between owners and variance is addressed by Lu and Potter [LP06a].

3.6 Confinement Support

Confinement is a form of ownership that confines classes to packages, rather than run-time instances of classes. There are advantages to this approach in its simplicity and its applications [PNZV05, CRN03, ANC+06]. Most importantly, Generic Ownership for confinement (called Generic Confinement [PNCB06a]) is so easy to express formally that it can reuse Featherweight Generic Java's type soundness result by the virtue of being defined as a subset of the Java language. However, just having confinement is insufficient to demonstrate that Generic Ownership can support deep ownership. Therefore, while Chapter 4 presents Generic Confinement, Chapter 5 presents a full Featherweight Generic Ownership type system with support for both ownership and confinement together with the required type soundness proofs.

Confinement is expressed using an owner constant named after the current package. So a package called `m` has a matching owner constant `M`, which is only accessible within its package. Whereas objects owned by `this` can only be accessed from within their owner and objects owned by a package can be accessed by any object instance of a class within that package.

Figure 3.5 shows an example of a declaration of an OGJ class called `mMain` — with its owner parameter `Owner` bound to its superclass's (`Object` in this case) owner parameter, and then how a stack class can be instantiated with different ownership.

Throughout the discussion in this thesis, I adopt the convention that package names are lower case single letters prefixing the class names. For example, class `mMain` is inside package `m`. Single capital letters are then used to refer to the owners of those packages. Thus for package `m`, its owner is `M` and the classes confined to this package can use it in their declarations as in `mMain<M>`. This feature makes the syntax in the OGJ type system as clean as possible. In the actual language implementation described in Chapter 6 the syntax is fully compatible with Java and class `mMain` actually becomes `m.Main`.

Within the `mMain` class four methods return *different types of* `OwnedStack` objects. One object is public, another is confined to package `m`, the next object is owned by a particular instance of class `mMain`, and the last object is owned by the owner of `mMain` class and can be shared with others owned by the same owner. `publicStack` stores `Object<World>` instances that are accessible from anywhere, because `Object`'s owner parameter is instantiated with `World`. The second stack stores `mMain` instances that are also globally accessible, however the stack itself has owner `M`, meaning that it is only accessible within package `m`. `privateStack` stores instances of `mMain` accessible inside package `m` only, while `privateStack` is only accessible by objects owned by a particular instance of `mMain` that created it. `sharedStack` stores the same sort of instances of `mMain` as the private stack, except that it is accessible by any other instances which have the same owner as the current instance of `mMain`. In each case, the stack's

```
1  class mMain<Owner extends World> extends Object<Owner> {
2
3    OwnedStack<Object<World>, World> publicStack() {
4      return new OwnedStack<Object<World>, World>();
5    }
6
7    OwnedStack<mMain<World>, M> confinedStack() {
8      return new OwnedStack<mMain<World>, M>();
9    }
10
11   OwnedStack<mMain<M>, This> privateStack() {
12     return new OwnedStack<mMain<M>,This>();
13   }
14
15   OwnedStack<mMain<M>, Owner> sharedStack() {
16     return new OwnedStack<mMain<M>, Owner>();
17   }
18 }
```

Figure 3.5: Confinement Support in Generic Ownership

second parameter describes its owner. These stacks illustrate how OGJ provides both *type genericity* (the stacks hold different item types) and *ownership polymorphism* (the stacks belong in different protection domains).

3.7 Manifest Ownership

OGJ supports a form of *manifest ownership* [Cla02] that allows classes without explicit owner type parameters. The class's owner is fixed, so all objects of that class have the same owner. This is the same way that in Java 5 a non-generic `IntegerList` class can be defined as extending `List<Integer>`, binding and fixing the list's type parameter.

Manifest ownership allows one to fit existing Java classes into the Generic Ownership class hierarchy by simply making Java's root class `Object` into a manifest class:

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

To avoid name conflicts (`Object` vs `Object<O>`), OGJ can choose a different name for the root of the class hierarchy, such as `OObject<O>`, meaning *owned* object. With this definition `Object` and every class inheriting from it has a default owner parameter `World` (thus making them publicly accessible). We can write the following familiar declaration of a public `Stack` object, which is indistinguishable from that of Java:

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

With manifest ownership, every `Stack` instance has an owner originating from OGJ's root class. This manifest owner can be looked up by the language implementation by traversing

the `Stack`'s class hierarchy until the class declaration that fixed the owner is found.

Manifest ownership allows streamlined integration of Generic Ownership with plain Java 5. Consider the following alternative formulation of a public stack class:

```
class PublicStack extends OwnedStack<World> { }
```

In this example, the *owner* of class `PublicStack` is `World`, thus all of its instances are owned by `World`. Uses of `PublicStack` require no owner type parameter, necause the owner is bound in the class declaration.

3.8 OGJ Class Hierarchy

Figure 3.6 shows the hierarchies of classes in OGJ. *Pure* program classes have an explicit owner type parameter. *Manifest* program classes have an owner fixed when subclassing a pure class. *Owner classes* lie outside the program class hierarchy because they cannot be instantiated in programs, e.g., `U` owner class corresponding to package `u`. Pure classes use them to bind their owner type parameters, as shown by the dashed arrow on the diagram.

The hierarchy of program classes is rooted at `Object<Owner extends World>`. Pure classes invariantly maintain the owner parameter, while manifest classes have an owner that is found from their superclasses. Vanilla Java classes form a subset of manifest classes. Owner classes inherit from the class `World`. There is one owner class corresponding to each package, as well as a special owner class `This` used to denote ownership of particular instances by the objects that instantiate them.

The diagram also shows owners of type $\texttt{This}_l$, the owners of objects owned by the object at location l. These are used in the Generic Ownership formalism, and are subclasses of `This`. The owners of objects cannot be used explicitly in the program and are not considered part of the class hierarchy, rather they are shown on the diagram for clarity.

3.9 OGJ Language Design

Ownership Generic Java is designed as a minimal extension to Java 5 [BOSW98]. The key difference is that OGJ allows classes to be declared with a distinguished (last) ownership type parameter (conventionally `Owner extends World`). Classes without an ownership parameter ("plain Java" classes descending from `java.lang.Object`) are treated as if they used manifest ownership.

OGJ supplies a number of ownership type constants: `World`, `Package`, and `This`. When bound to a class's ownership type parameter, these constants mark the instances as public, as confined within their package, or as owned by the current "this" object, respectively. The owner `Package` used in the program is automatically replaced with the current package's owner class name to avoid the burden of having to manipulate package

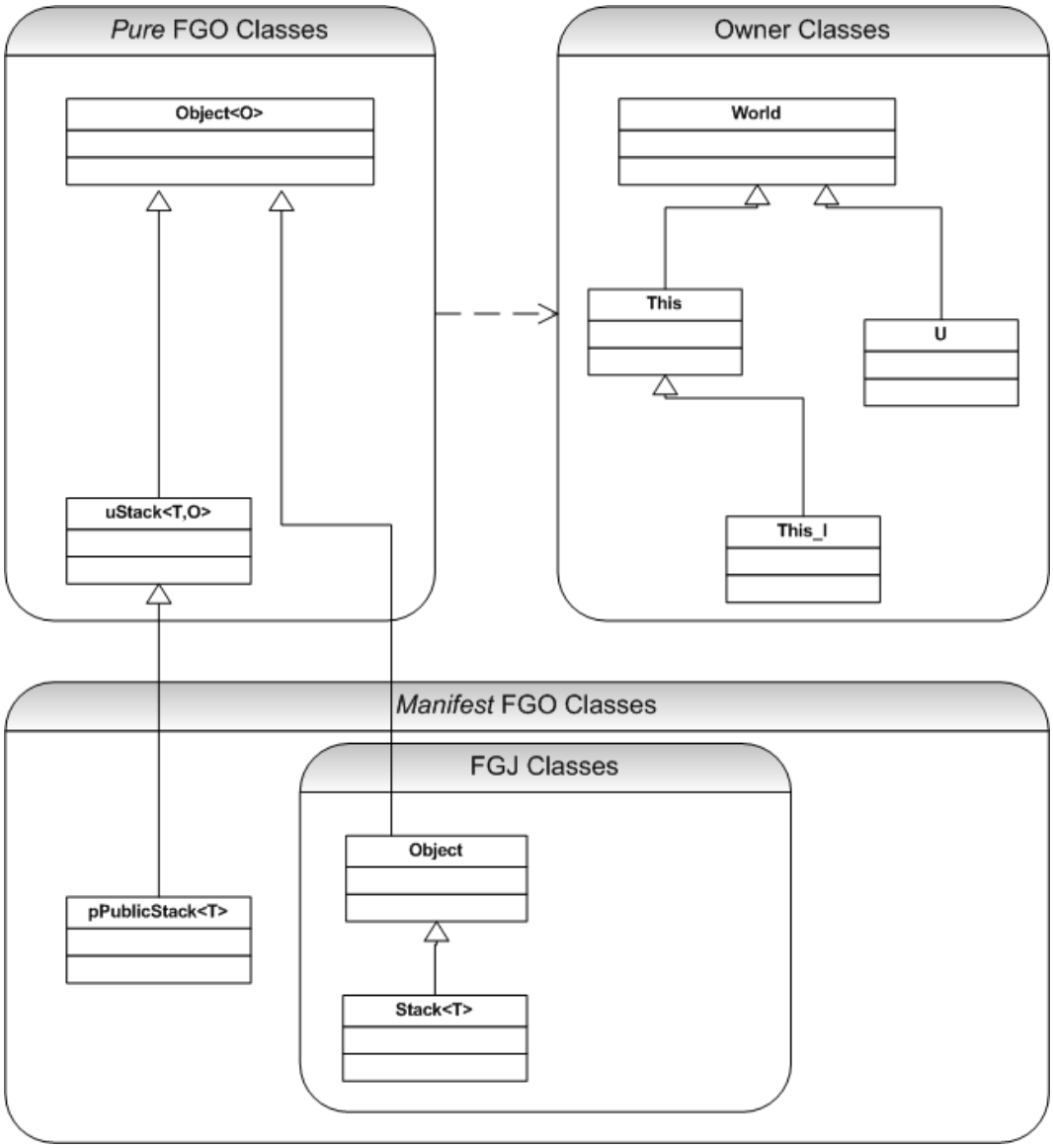

Figure 3.6: OGJ Program Classes and Owner Classes

owner classes explicitly. Since it is impossible to use a package owner class name outside its package, there is always only one valid replacement for owner `Package`. To ensure deep ownership, OGJ restricts the types which can be formed so that the distinguished (last) ownership type parameter is always *inside or equal to* (greater or equally encapsulated than) any other parameters' ownership type.

The ordering on the owners involved in a valid OGJ type is as follows. When an actual owner parameter is `World`, all the other type parameters must have an owner `World`. If the actual owner parameter is a package or an ownership type variable, then that package (or variable) and `World` are permissible. If the actual owner parameter is an enclosing class's `Owner` then `World`, `Owner`, and other (placeholder) formal ownership type parameters of the enclosing class are permissible. Finally if an ownership type parameter is bound to

`This`, then the other parameters may be bound to anything.

The motivation behind such owner nesting restrictions is that the distinguished owner parameter defines who is allowed to access the class instance. If we have a public instance having access to a private instance by the virtue of one of the type parameters (e.g., `SomeClass<This, World>`) then such an instance can be potentially accessed by anyone in the object graph. This may break the dominator relationship required by the deep ownership by providing a path from the root of the object graph bypassing the owner of this instance of the `SomeClass`. The dominance restriction imposed by deep ownership was discussed in Chapter 2. If one is to have a `Map<Key, Value, World>` class, then both `Key` and `Value` would have to be owned by `World` too, while the table of map entries (or nodes) can be a private field inside the `Map` class owned by its instance `This`.

The presence of ownership type constants and their preservation over subtyping is sufficient to provide package confinement. To enforce per-object ownership, OGJ ensures that fields and methods with types with an actual owner parameter of `This` can only be accessed via Java's `this` keyword, either explicitly or implicitly. Assignments such as `this.pvtField = other.pvtField` between two `Node` instances are illegal if `this.pvtField` is owned by `this`; similarly method or field accesses involving a `This` owner are only permitted on the current "`this`" object.

To ensure that ownership information cannot be lost, OGJ requires type narrowing and widening to preserve ownership. This follows Java's existing rules for subtyping parameterised types, except that it prevents casts to raw types [IPW01b, BOSW98] when such casts would delete an ownership parameter. OGJ must also restrict wildcards for ownership types as type variable bounds, and prevent reflection and serialisation (cloning) so that it cannot breach ownership.

These are *the only* restrictions imposed by Generic Ownership on top of the vanilla type generic language. These restrictions alone achieve ownership *and* confinement guarantees comparable to the alternative systems [ZPV06, AKC02, Boy04] with a large amount of the burden carried by the underlying sound type generic system. Hence, these restrictions highlight the concepts important to the mechanism of ownership, excluding the rest of the rules required by the alternative type systems as technical details.

Chapter 6 goes into more detail of the OGJ language implementation [Pot05] that includes a prototype that can bootstrap itself (albeit with substantial use of standard (manifest) Java classes) and compile a growing test suite.

3.10 Comparative Examples of OGJ Programs

To demonstrate the difference between OGJ and the other ownership languages, consider the examples in the Figure 3.7 and Figure 3.8. Both of these simple examples

```
1  import ogj.ownership.*;
2
3  class Point<Owner extends World> {
4    Integer x; Integer y;
5    Point(Integer x, Integer y) {
6      this.x = x; this.y = y;
7    }
8  }
9
10 class Rectangle<Owner extends World> {
11   private Point<This> upperLeft;
12   private Point<This> lowerRight;
13
14   public Rectangle(Point<Owner> ul, Point<Owner> lr) {
15     // Simple assignment: upperLeft = ul, is illegal
16     // in both Java and OGJ due to incompatible type
17     // parameters This and Owner, hence copying is forced
18     upperLeft = new Point<This>(ul.x, ul.y);
19     lowerRight = new Point<This>(lr.x, lr.y);
20   }
21
22   public void doIt() {
23     Point<This> p;
24     p = this.upperLeft;
25     p = this.exposeUpperLeft();
26     Rectangle<Owner> ro = this;
27     p = ro.upperLeft; // WRONG in OGJ (not in Java)
28     p = ro.exposeUpperLeft(); // WRONG in OGJ (not in Java)
29   }
30
31   // This method can only be called from inside the instance
32   // itself: this.exposeUpperLeft()
33   private Point<This> exposeUpperLeft() {
34     return upperLeft;
35   }
36
37   public Point<Owner> getUpperLeft() {
38     // return upperLeft; // WRONG in both Java and OGJ.
39     return new Point<Owner>(upperLeft.x, upperLeft.y);
40   }
41 }
```

Figure 3.7: `Rectangle` Class in OGJ

are compiled with the latest version of AliasJava[1] and OGJ[2]. Both examples have two private fields protected from erroneous exposure by making them *owned* by the instance of `Rectangle` that created them. The constructor is forced to make a private copy of

[1] `http://www.archjava.org/`
[2] `http://www.mcs.vuw.ac.nz/~alex/ogj/`

```
1  class Point {
2    int x; int y;
3    Point(int x, int y) {
4      this.x = x; this.y = y;
5    }
6  }
7
8  class Rectangle {
9    private owned Point upperLeft;
10   private owned Point lowerRight;
11
12   public Rectangle(Point ul, Point lr) {
13     // ArchJava forces to copy the values, since the default
14     // annotation (lent) does not match annotation owned.
15     upperLeft = new Point(ul.x, ul.y);
16     lowerRight = new Point(lr.x, lr.y);
17   }
18
19   public void doIt() {
20     owned Point p;
21     p = this.upperLeft;
22     p = this.exposeUpperLeft();
23     Rectangle r = this;
24     p = r.upperLeft; // WRONG
25     p = r.exposeUpperLeft(); // WRONG
26   }
27
28   private owned Point exposeUpperLeft() {
29     return upperLeft;
30   }
31
32   public unique Point getUpperLeft() {
33     // The copy is enforced using unique annotation.
34     // return upperLeft; // WRONG.
35     return new Point(upperLeft.x, upperLeft.y); // RIGHT
36   }
37 }
```

Figure 3.8: `Rectangle` Class in AliasJava

the supplied `Point` references. The method `doIt` fails to expose the private fields via `exposeUpperLeft()` unless the method receiver is explicitly `this`. It is interesting to observe that a lot of work to do with aliasing protection is performed by Java 5 on behalf of OGJ, as the comments in the figure point out.

AliasJava has full support for ownership with annotations such as `owned` added on top of Java syntax, while OGJ's syntax is 100% Java compatible. OGJ also fully supports confinement, in addition to ownership, while still keeping Java syntax. This means that it inherits the problems of Java 5, such as lack of proper generic array support. But on

the other hand, defaulting of the *last* type parameter to be an owner is much simpler than using custom annotations [PNCB04]. Most importantly, OGJ is the first working language implementation that supports ownership, confinement, and generic types at the same time.

Chapter 4

Featherweight Generic Confinement

Contents

This chapter continues the efforts to provide effective object encapsulation within practical programming languages. The main goal is to obtain a simpler formalism than the existing approaches, with few new concepts. The key insight behind this chapter is that confinement and ownership type systems can readily be modelled within existing parametric polymorphic type systems: in fact, this chapter demonstrates 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. In comparison, the deep ownership described in Chapter 5 requires sufficient amount of modifications to the language to require a full type soundness proof. The FGJ+c type system has been presented in Potanin et al. [PNCB06a].

In this chapter I present the approach that I call Featherweight Generic Confinement (FGC) and the type system called FGJ+c [1] (for Featherweight Generic Java plus Confinement) which provides confinement guarantees and type polymorphism (type genericity). The fundamental ideas of Generic Ownership are present throughout FGJ+c.

Confinement allows classes inside a domain (for example a Java package) to be declared private to that domain. 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.

FGJ+c presents the additional features required by confinement as a few simple restrictions, rather than being spread throughout the type system. These restrictions are (1) owner preservation through subtyping, and (2) ability to tell the owner of each type via its last type parameter. The benefit of formalising confinement only is that the resulting system is able to reuse FGJ's type soundness property.

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.

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 the programming languages.

The main difference between FGJ+c and alternative type systems providing similar guarantees (such as Confined Featherweight Generic Java (CFGJ) [ZPV06]) is that FGJ+c takes an existing type polymorphic system, Featherweight Generic Java (FGJ) [IPW01a], and adds confinement to it by imposing a small number of additional restrictions. The resulting FGJ+c type system is simpler than the alternatives and it is easier to prove the confinement guarantees comparable to those of CFGJ.

In what follows, I outline the main principles behind FGJ+c, give a formal presentation of FGJ+c's type system and a proof of its confinement invariant.

4.1 Featherweight Generic Java + Confinement

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 [Cla02] every pure FGJ+c class is required to have at least one type parameter to carry this confinement information. Following both the confinement and ownership literature, this extra parameter is called the *owner* type parameter.

[1] The plus in FGJ+c symbolises that it is not a stand-alone type system like FGO in the next chapter, but rather a collection of restrictions to FGJ that enforce confinement.

FGJ+c uses the *last* type parameter to record an object's owner to promote upwards compatibility with FGJ. All FGJ+c classes descend from a new class `CObject<O>` (for confinable object) which has just one parameter and all of its subclasses must invariantly preserve this owner. The preservation of owner over subtyping lies at the foundation of Generic Confinement.

The FGJ+c code in Figure 4.1 demonstrates a possible implementation of a functional stack inside package `u`. Note that all class names are prefixed by a package identifier, e.g., `uStack` is a `Stack` in package `u`. This is a convention to indicate the package within which each class is defined. I 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. The classes that extend class `World` are used to indicate ownership. For each package I 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 the stack. The presence of the two type parameters 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 type parameter of the stack describing the type of the items that stack contains is bounded by class `Any` — which allows any subclass of `CObject<O>` to be used in its place for *any* owner parameter. The `Any` bound was not required in the stack example in Section 3.6 because that example only showed the uses of the stack class, rather than stack class declaration. Please observe that one cannot make `uNode` to be 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 4.2 presents an example of utilising the stack. The FGJ+c code in Figure 4.2 demonstrates how confinement inside package `s` can be enforced using owner classes. Although `void` is not part of FGJ, FGJ can be trivially extended with `void`. The presence of `void` simplifies the presentation of the idea in this figure. 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 type — `uStack <sPassword<S>, S>` — includes the owner class `S` that can only be written inside package `s`. If one tries to access the contents of `sPasswordManager`'s stack in a different package (for example, by calling `getSecretPassword` in package `m`) one will not 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, the type soundness of FGJ (and thus FGJ+c) ensures that there is no way around this restriction.

FGJ is a functional language, thus the result of `addSecretPassword` cannot be assigned outside the package `s`. This does not stop one from ignoring the resulting value

```
1  // Package u (as in ``util'').
2
3  // Class uNode serves as a "null object" for the purposes of
4  // implementing a uStack.
5  class uNode<T extends Any, Owner extends World>
6    extends CObject<Owner> {
7
8    uNode() { super(); }
9  }
10
11 // Class uStackNode is a simple stack node used by uStack.
12 class uStackNode<T extends Any, Owner extends World>
13   extends uNode<T, Owner> {
14
15   T element; uNode<T, Owner> nextNode;
16
17   uStackNode(T element, uNode<T, Owner> nextNode) {
18     super(); this.element = element;
19     this.nextNode = nextNode;
20   }
21 }
22
23 // Class uStack implements a simple functional stack.
24 class uStack<T extends Any, Owner extends World>
25   extends CObject<Owner> {
26
27   uNode<T, Owner> root;
28
29   uStack(uNode<T, Owner> root) {
30     super(); this.root = root;
31   }
32
33   uStack<T, Owner> push(T element) {
34     return new uStack<T, Owner>(
35       new uStackNode<T, Owner>(element, this.root));
36   }
37
38   uStack<T, Owner> pop() {
39     return new uStack<T, Owner>(
40       ((uStackNode<T, Owner>) this.root).nextNode);
41   }
42
43   T top() {
44     return ((uStackNode<T, Owner>) this.root).element;
45   }
46 }
```

Figure 4.1: FGJ+c Stack Example

```
1  // Package s (as in ``secret'').
2  class sPassword<Owner extends World> extends CObject<Owner> {
3    int secretID;
4    sPassword(int secretID) { super(); this.secretID = secretID; }
5  }
6
7  class sPasswordManager<Owner extends World>
8    extends CObject<Owner> {
9    sPasswordManager() { super(); }
10   uStack<sPassword<S>, S> createStack() {
11     return new uStack<sPassword<S>, S>(
12       new uNode<sPassword<S>, S>())
13   }
14
15   uStack<sPassword<S>, S> addSecretPassword(int secretID) {
16     return this.createStack().push(new sPassword(secretID));
17   }
18
19   // This method can only be called inside package s.
20   // Outside of package s one cannot write the type
21   // of the return element because of owner S.
22   sPassword<S> getSecretPassword() {
23     return this.addSecretPassword(7).top();
24   }
25 }
26
27 // Package m (as in ``main'').
28 class mMain<Owner extends World> extends CObject<Owner> {
29   mMain() { super(); }
30
31   sPasswordManager<M> createPasswordManager() {
32     return new sPasswordManager<M>();
33   }
34
35   void addSecretPassword() {
36     this.createPasswordManager().addSecretPassword(42);
37   }
38
39   void getSecretPassword() {
40     // One cannot perform a call to this method or assign the
41     // result to anything, including the super class CObject<S>,
42     // since not allowed to use owner S outside package s.
43     this.createPasswordManager().getSecretPassword()
44   }
45 }
```

Figure 4.2: FGJ+c Confinement Violation Example

Kinds of Classes	In package `p`	Outside package `p`
Classes with owner `World`	Allowed	Allowed
Classes with owner `P`	Allowed	Not Allowed
Classes with one of type parameters involving owner `P`	Allowed	Depends on the class owner
Classes with other owners	Not Allowed	Depends on the class owner

Table 4.1: Allowed Use of Owner Classes

when using this code in package `m` for the sake of example. Since one cannot have local variables or update fields in the functional setting of FGJ. This example is sufficient though to demonstrate the case in hand since the classes confined to package `s` are used outside it in package `m` as they would be in an imperative language implementing Generic Confinement (as described in Chapter 6).

4.1.1 Packages and Owner Classes

The FGJ+c domain of confinement is a package. FGJ+c needs to represent the packages in the FGJ type system as owners. FGJ+c uses parameterless FGJ classes that form a separate class hierarchy extending `World` to represent these domains (packages). These types are called owner classes as defined in Chapter 3. Because the classes that represent domains cannot be used in place of valid FGJ+c classes, they should not be instantiated during the execution of an FGJ+c program. The reason for this restriction is to make sure that any valid class in an FGJ+c program has an owner type parameter. For owner classes themselves it makes no sense to have additional owner type parameter.

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

Table 4.1 clarifies which classes are allowed in which packages depending on their owner, or the owner of one of their type parameters.

4.1.2 Manifest Ownership

Manifest ownership is described in Chapter 3, Section 3.7 and it allows classes without explicit owner type parameters. When Manifest Ownership is applied to FGJ+c, it 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`, which ensures that all instances of `Link` will be confined within that package. One cannot "fix" the owner to anything other than `L`, because the class is declared inside package `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`. Less explicitly, it is also not possible to write:

```
class qC extends lLink<qSomeType> { ... }
```

Because lLink has manifest owner `l` prohibited as owner of classes used inside an unrelated package such as `q`. This is checked by the visibility rules described later in this chapter.

4.2 FGJ + Confinement Definition

FGJ+c can be considered a strict subset of FGJ, that is, every FGJ+c program is an FGJ program if one allows for a different root of the class hierarchy. The FGJ type system is discussed in Chapter 2. 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 [IPW01a] (listed for the reader's convenience in Appendix A) along with additional rules presented in this section. For reference, Figure 4.3 shows the FGJ syntax from [IPW01a] as well as the common environments Δ, Γ, and CT used in FGJ type rules. To simplify the presentation, I assume that owner classes are syntactically distinguishable. Owners have the syntax:

```
O ::= X^O | N^O
```

```
T ::= X  |  N
N ::= C<T̄>
L ::= class C<X̄◁N̄>◁N {T̄ f̄; K M̄}
K ::= C(T̄ f̄) { super(f̄); this.f̄=f̄; }
M ::= <X̄◁N̄> T m(T̄ x̄) { return e; }
e ::= x | e.f | e.m<T̄>(ē) | new N(ē) | (N)e

X ranges over the type variables.
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.
```

Figure 4.3: FGJ+c Syntax (Identical to FGJ Syntax from Igarashi et al.) and Environments

where O ranges over all owners, X^O ranges over owner variables, and N^O 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, which can be distinguished using the following syntax:

$$N ::= C\langle \overline{T},\ O \rangle$$
$$L ::= \texttt{class}\ C\langle \overline{X} \lhd \overline{N},\ X^O \lhd N^O \rangle \lhd N\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}$$

Following the FGJ type system, the syntactical term Y corresponds to type variables and P corresponds to nonvariable types. FGJ+c also adopts the syntactical idiosyncrasies of FGJ, such as $\overline{T}\,\overline{x}$ denoting a list of pairs: $T_1\ x_1, \ldots, T_n\ x_n$, rather than two lists of types and variables. The substitution notation used in FGJ+c follows FGJ so that $[\overline{T}/\overline{X}]N$ means that every occurrence of X_i in N is replaced by T_i.

All of the additional FGJ+c rules presented in this chapter guarantee that FGJ+c programs do not break confinement, which allows us to prove a confinement invariant.

4.2.1 FGJ+c Programs

Any FGJ+c program is an FGJ program that meets the following requirement: all classes must satisfy either rule FGJ+C-CLASS (for generic classes) or rule 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 4.4 shows the type judgements used in FGJ+c rules, and Figures 4.5, 4.6, and 4.7 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

$owner_\Delta(T)$	Determines owner of type T.
$owners_\Delta(T)$	Helper function finding all owners occurring in type T.
$\Delta \vdash T\ \texttt{OK+c}$	Type T is OK.
$visible_\Delta(O, D)$	Owner O is visible in class D.
$visible_\Delta(T, D)$	Type T is visible in class D.
$\Delta;\Gamma \vdash visible(e, D)$	Expression e is visible in class D.
$\langle \overline{Y} \lhd \overline{P} \rangle\ T\ m(\overline{T}\,\overline{x})\{\ \texttt{return}\ e_0;\ \}\ \texttt{OK+c IN}\ C\langle \overline{X} \lhd \overline{N} \rangle$	Method m definition is OK.
$\texttt{class}\ C\langle \overline{X} \lhd \overline{N} \rangle \lhd N\ \{\overline{T}\,\overline{f};\ K\,\overline{M}\}\ \texttt{OK+c}$	Class C definition is OK.

Figure 4.4: FGJ+c Judgements

Bound of type (from FGJ):

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

FGJ+c Owner Lookup Function (FGJ+C-OWNER) **:**

$$
\begin{array}{lcl}
owner_\Delta(\texttt{Any}) & = & \texttt{World} \\
owner_\Delta(\texttt{X}) & = & owner_\Delta(\Delta(\texttt{X})) \\
owner_\Delta(\texttt{C<}\overline{\texttt{T}}\texttt{, O>}) & = & \texttt{O} \\
owner_\Delta(\texttt{C<}\overline{\texttt{T}}\texttt{>}) & = & owner_\Delta([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}) \\
 & & \text{where } CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N}\ \{\overline{\texttt{T}'}\ \overline{\texttt{f}}\texttt{; K}\ \overline{\texttt{M}}\}
\end{array}
$$

FGJ+c Types (FGJ+C-TYPE) **:**

$$\frac{\Delta \vdash \texttt{T <: CObject<O>} \quad \Delta \vdash \texttt{O <: World}}{\Delta \vdash \texttt{T OK+c}} \qquad \frac{\Delta \vdash \texttt{X <: Any}}{\Delta \vdash \texttt{X OK+c}}$$

Figure 4.5: FGJ Bound of type, FGJ+c Owner Lookup Function, and FGJ+c Types

the visibility of a type, Figure 4.5); secondly, they determine which types are visible in a given class (Figure 4.6); and thirdly, they propagate and check the desired constraints for fields and methods (Figure 4.7).

FGJ+c Owner Lookup Function. The *owner* function in Figure 4.5 is used by the majority of the rules to look up the owner class corresponding to a given type. The owner class 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. The first two clauses of the function definition deal with a special class `Any` described in the next section and type variables. For the former, the owner is assumed to be `World` and for the latter the owner function has to look up the owner of the bound given by the mapping in the environment Δ. The third clause deals with standard (pure) classes that contain the owner class explicitly, and the fourth and final clause recursively looks up the owner of a manifest class by traversing the super class hierarchy.

FGJ+c Types. The first part of the rule FGJ+C-TYPE in Figure 4.5 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 its superclass hierarchy). Note that type variables will be classified as valid FGJ+c types because 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, the root type `CObject<O>` is also allowed. The second rule admits the type variables bounded by `Any`.

4.2.2 Any Type Bound

FGJ requires every type variable to be bounded. To increase the expressiveness of the type system presented in this chapter, a very limited form of anonymous type parameters is introduced. 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 formal parameter is bounded by `Any`. For example, in Figure 4.1, `uStack`'s first type parameter is bounded by `Any`.

`Any` solves the problem of not knowing the bound of the owner parameter for every type parameter involved in a class declaration. To instantiate `uStack` with a type parameter that has an owner (which can be different from the owner of the stack), one would need to have specified this owner explicitly for the FGJ rules to find a bound:

```
class uStack<T extends uStackable<TOwner>,
  TOwner extends World,  Owner extends World>
    extends CObject<Owner> { ... }
```

Simply declaring additional owner parameters will make FGJ+c syntax too cumbersome to have any advantage over a naive combination of owners and type parameters. To resolve this, FGJ+c adds class `Any` just above the root class `CObject<O>`, which can be thought of as the following declaration:

```
class CObject<Owner extends World> extends Any { ... }
```

FGJ+c allows `Any` in place of the type variable bound, but not in any other part of a valid FGJ+c program (for example, one is prohibited from ever instantiating `Any`).

Class `Any` can be thought of in the same way as the owner classes - they are valid FGJ classes but do not form part of FGJ+c class hierarchy, since the root of FGJ+c class hierarchy is `CObject<O>`. The presence of `Any` allows the use of a nonvariable bound for type parameters. The importance of not allowing other uses of `Any` stems from the fact that a class with any owner can be a subtype of `Any`. `Any` is assumed to be publicly accessible (owned by `World`) and thus one cannot allow `Any` as an FGJ+c nonvariable type if one wants preservation of owners through subtyping — an essential property of FGJ+c.

The approach of using `Any` as a bound is really just encoding unbounded parametric polymorphism in the syntax of FGJ. Another approach is simply dropping the `Any` bound, which is actually unbounded polymorphism. The usual approach to encoding unbounded polymorphism in bounded polymorphism is to have bound "`<: Object`", but since this breaks owner preservation over subtyping in FGJ+c the present approach is used. The reason simply having an `Object` bound breaks owner preservation is because the owner for a type parameter bounded by `Object` either has to be fixed using a manifest class or explicitly declared as in `Object<O>`. `Any` bound explicitly assumes any owner, thus not forcing the class designer to declare all the owner bounds explicitly.

`Any` type bound does not allow any constraints on owners or types of actual parameters. More flexible schemes such as wildcards or variance [IV02] remove these restrictions and allow a single type parameter to be independently bounded for type and ownership. Lacking variance or multiple inheritance, it is not possible to express such flexible bounds in a system built on top of pure FGJ. However, this technique is applicable in other systems.

The `Any` approach can be easily generalised to provide a similar superclass for any pure FGJ+c class, allowing nonvariable bounds to be other than `Any` as in the following example:

```
class uStack<T extends uStackable<?>, Owner extends World>
  extends CObject<Owner> { ... }
```

`Any` is treated specially by the type system, not making it part of the FGJ+c class hierarchy that is formed under `CObject<O>`. Without encoding unbounded polymorphism using `Any`, FGJ+c will not be able to support stacks without providing a final value to the owner type parameter for *both* the stack itself and the items stored in it. This issue is resolved in FGO which is described in Chapter 5 using $placeholderowners$ function, albeit requiring more modifications to the type system.

The extension to `Any` is kept for the sake of simplicity. An important observation is that one still needs to ensure owner preservation over subtyping in FGJ+c. If, for example, FGJ+c is to support variance [IV02] it would be much harder to ensure that subtypes like: `uStack<S> <: uStack<World>` (`<:` stands for subtyping relation) are disallowed. A full exploration of the interaction of variance with Generic Confinement remains an open question. The presence of the `Any` type bound as presented in this subsection is enough to demonstrate the matter at hand.

4.2.3 Visibility

Visibility is the key to Generic Confinement. Visibility 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. The owner variables of a class and of its type parameters must be always visible because Generic Confinement views passing type parameters as granting permission to access the actual argument types (see Subsection 4.4.1).

The visibility rules shown in Figure 4.6 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. The function $\pi_{\mathtt{D}}$ returns the owner class corresponding to the package to which `D` belongs.

The *owner* visibility predicate ($visible_{\Delta}(\mathtt{O}, \mathtt{D})$) simply states 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`),

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

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

Type Visibility:

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

Term Visibility:

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

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

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

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

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

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

Figure 4.6: FGJ+c Visibility Rules

is a public owner (denoted World), or is the owner of one of the type parameters, which roughly follows the visibility rule in CFGJ [ZPV06].

The *type* visibility predicate allows the use of types when their owner parameter is visible. *Term* visibility rules are defined inductively on the structure of FGJ terms. For each subexpression, the rules determine whether its type is visible according to the type visibility rules.

Consider the following type, where capitals letters (e.g., P, Q, R, S) refer to owner classes marking instances confined to packages written in small letters (e.g., p, q, r, s):

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

This type describes a list declared in package p, storing items declared in package q that are confined to package r, while the list itself is confined to package s. This type of list is allowed access to objects confined to p since the code performing the access is

already inside package p, to s since the list instance is confined in that package during the execution, and to the instances confined to package r because one of the list's type parameters is confined in that package (*i.e.*, is owned by R). The latter is required, for example, to allow the list to return its element type. Any classes confined to q or any other package cannot be accessed inside the list.

4.2.4 Classes and Methods

Figure 4.7 shows how visibility constraints are propagated through classes to their fields and methods and eventually to expressions. An 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, so that the FGJ type system's type soundness property guarantees the validity and predictable execution of all the FGJ+c programs.

The rules in Figure 4.7 say that *every* type appearing in the program, even as the type of a subexpression, *must* be visible in the current class. The typing environment Δ is built up from the method and class declaration rules using the subtyping information that is supplied via the type parameters. For classes, (FGJ+C-CLASS) and (FGJ+M-CLASS), the FGJ type system together with the FGJ+c restrictions checks 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 is in the presence of an explicit owner parameter in the class being declared. For pure FGJ+c classes, an owner parameter is present and FGJ+c requires that its immediate superclass has the same owner parameter. For manifest FGJ+c classes, the owner parameter is no longer explicit, so one needs to check that the superclass is visible with respect to the current class. Any is allowed as a 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 a class's type parameters gives permission to access classes confined to that owner. This issue is further addressed in the discussion in Chapter 7. In addition, FGJ+c ensures that the all types instantiated in programs are a subtype of CObject<O> for some owner class O and thus prevents owner classes from being instantiated. The FGJ+C-CLASS rule also enforces the restriction that only owner classes can be used as owners.

For methods (FGJ+C-METHOD), FGJ+c checks that the argument and return types are visible, that the method body expression satisfies the visibility constraints, and that the type variables of the method have bounds which are visible.

FGJ+c Methods:

$$\frac{\Delta = \{\overline{\mathtt{X}} <: \overline{\mathtt{N}},\ \overline{\mathtt{Y}} <: \overline{\mathtt{P}}\} \quad \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}, \overline{\mathtt{P}}\ \mathtt{OK{+}c} \quad \mathit{visible}_\Delta(\overline{\mathtt{T}},\ \mathtt{C}) \quad \mathit{visible}_\Delta(\mathtt{T},\ \mathtt{C}) \quad \mathit{visible}_\Delta(\overline{\mathtt{P}},\ \mathtt{C}) \quad \Delta;\ \overline{\mathtt{x}} : \overline{\mathtt{T}},\ \mathtt{this} : \mathtt{C} < \overline{\mathtt{X}} > \ \vdash \ \mathit{visible}(\mathtt{e}_0,\ \mathtt{C})}{< \overline{\mathtt{Y}} \lhd \overline{\mathtt{P}} > \mathtt{T}\,\mathtt{m}(\overline{\mathtt{T}}\,\overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \mathtt{OK{+}c\ IN}\ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} >} \quad \text{(FGJ+C-METHOD)}$$

FGJ+c Classes:

$$\frac{\mathtt{N} = \mathtt{C}' < \overline{\mathtt{T}'}, \mathtt{X}^0 > \quad \Delta = \{\overline{\mathtt{X}} <: \overline{\mathtt{N}},\ \mathtt{X}^0 <: \mathtt{N}^0\} \quad \Delta \vdash \mathtt{N}, \overline{\mathtt{T}}\ \mathtt{OK{+}c} \quad \forall \mathtt{N}' \in \overline{\mathtt{N}} : \mathtt{N}' = \mathtt{Any} \ \vee \ \Delta \vdash \mathtt{N}'\ \mathtt{OK{+}c} \quad \overline{\mathtt{M}}\ \mathtt{OK{+}c\ IN}\ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}},\ \mathtt{X}^0 \lhd \mathtt{N}^0 > \quad \mathit{visible}_\Delta(\mathtt{N}^0,\ \mathtt{C}) \quad \mathit{visible}_\Delta(\overline{\mathtt{T}},\ \mathtt{C}) \quad \mathit{visible}_\Delta(\overline{\mathtt{N}},\ \mathtt{C}) \quad \mathtt{class}\ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}},\ \mathtt{X}^0 \lhd \mathtt{N}^0 > \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}}{\mathtt{class}\ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}},\ \mathtt{X}^0 \lhd \mathtt{N}^0 > \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK{+}c}} \quad \text{(FGJ+C-CLASS)}$$

$$\frac{\Delta = \{\overline{\mathtt{X}} <: \overline{\mathtt{N}}\} \quad \mathit{visible}_\Delta(\mathtt{N},\ \mathtt{C}) \quad \Delta \vdash \mathtt{N}, \overline{\mathtt{T}}\ \mathtt{OK{+}c} \quad \overline{\mathtt{M}}\ \mathtt{OK{+}c\ IN}\ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \quad \forall \mathtt{N}' \in \overline{\mathtt{N}} : \mathtt{N}' = \mathtt{Any} \ \vee \ \Delta \vdash \mathtt{N}'\ \mathtt{OK{+}c} \quad \mathit{visible}_\Delta(\overline{\mathtt{T}},\ \mathtt{C}) \quad \mathit{visible}_\Delta(\overline{\mathtt{N}},\ \mathtt{C}) \quad \mathtt{class}\ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}}{\mathtt{class}\ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK{+}c}} \quad \text{(FGJ+M-CLASS)}$$

Figure 4.7: FGJ+c Method and Class Rules

An alternative design would be to use visibility checks throughout the existing FGJ rules (as in CFJ [ZPV06]), rather than building in the appropriate checks into the (FGJ+C-CLASS) and (FGJ+M-CLASS) rules. The advantage of the separate approach is that one can reuse all of the FGJ rules (and proofs) and that this demonstrates that FGJ+c does not need a type system stronger than FGJ.

4.3 Confinement Guarantees

The confinement invariant states that types that are not visible within the current package are not accessible. I assume that FGJ+c only deals with programs for which all classes are well typed FGJ+c classes as given by the rules in Figure 4.7. FGJ+c also assumes that all the FGJ requirements, including FGJ class well-formedness are satisfied in addition to the FGJ+c class well-formedness. The confinement invariant then states that during the execution, any expression being evaluated cannot result in an instance of a class that is not *visible* within the current protection domain — in this case a Java package. This result relies on the fact that the owner parameter is preserved in the class hierarchy. FGJ+c uses the FGJ type system's reduction rules.

Lemma 1. ***(Ownership Invariance)*** *If* $\Delta \vdash \mathtt{S} <: \mathtt{T}$ *and* $\Delta \vdash \mathtt{T} <: \mathtt{CObject} < \mathtt{O} >$*, then*

$owner_\Delta(\mathtt{S}) = owner_\Delta(\mathtt{T}) = \mathtt{O}$.

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 1. ***(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. *If* $\mathtt{e} \xrightarrow{*} \mathtt{new\ D} < \overline{\mathtt{T_D}} > (\overline{\mathtt{e}})$*, then* $visible_\Delta\ (\mathtt{D} < \overline{\mathtt{T_D}} >,\ \mathtt{C})$.

Proof. Because the class is a well formed FGJ+c class, its methods are well formed FGJ+c methods. This means that all the expressions (since they can only occur in method bodies) are well formed too. The latter and FGJ's subformula property [2] imply that, for appropriate Δ and Γ, both $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$ and $\Delta;\Gamma \vdash visible(\mathtt{e},\ \mathtt{C})$ hold. From this one can derive using term visibility rules that $visible_\Delta(\mathtt{T},\ \mathtt{C})$, and hence by V-TYPE $visible_\Delta(owner_\Delta(\mathtt{T}),\ \mathtt{C})$.

To establish the visibility one needs to find an owner of $\mathtt{D} < \overline{\mathtt{T_D}} > (\overline{\mathtt{e}})$ that e reduces to. By the FGJ's type preservation property, there is a $\mathtt{T}'$ such that $\Delta;\Gamma \vdash \mathtt{new\ D} < \overline{\mathtt{T_D}} > (\overline{\mathtt{e}}) : \mathtt{T}'$, where $\Delta \vdash \mathtt{T}' <: \mathtt{T}$. Furthermore, one has that $\Delta;\Gamma \vdash \mathtt{new\ D} < \overline{\mathtt{T_D}} > (\overline{\mathtt{e}}) : \mathtt{D} < \overline{\mathtt{T_D}} >$, and hence clearly $\Delta \vdash \mathtt{D} < \overline{\mathtt{T_D}} > <: \mathtt{T}'$, and $\Delta \vdash \mathtt{D} < \overline{\mathtt{T_D}} > <: \mathtt{T}$.

By the Ownership Invariance lemma, $owner_\Delta(\mathtt{D} < \overline{\mathtt{T_D}} >) = owner_\Delta(\mathtt{T})$ from which one deduces $visible_\Delta(owner_\Delta(\mathtt{D} < \overline{\mathtt{T_D}} >),\ \mathtt{C})$, and hence $visible_\Delta(\mathtt{D} < \overline{\mathtt{T_D}} >,\ \mathtt{C})$.

□

4.4 Discussion

Every FGJ+c program is a valid FGJ program — it type checks and executes following FGJ's evaluation rules, with no illegal accesses to the confined instances of other classes. Every FGJ class can be mapped to a manifest FGJ+c class by an FGJ's `Object` extending `CObject<World>`. Therefore all FGJ programs are FGJ+c programs with manifest owner `World`. The special treatment that the owner classes receive when interpreted by the additional FGJ+c rules allows us to guarantee a confinement invariant. This section delves into a few of the more interesting aspects of FGJ+c.

4.4.1 Generic Confinement

Generic Confinement raises two issues that are also discussed by Zhao et al. [ZPV06].

First, as with ownership type systems [Cla02], 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 the case of FGJ+c, the actual owner

[2] The subformula property basically means that all the subexpressions of an expression have to be well formed since the expression typing rules recurse into every possible subexpression and check its well-formedness.

parameter gives access to the classes confined within the package corresponding to the actual owner parameter. This access granting mechanism is made explicit in the owner visibility rule, which ensures an owner is visible if it is the owner of any of the actual type parameters. Not addressing this issue in the generic setting can lead, for example, to collections not being able to do anything (including store or return) to the items they store.

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

```
class p1Foo {
  ...
  Bar<P2> m() {
    return (new uMap<Integer<World>,
      Bar<P2>, P1>).get(new Integer(42));
  }
}
```

One can do the reduction and see that although this code could be located *anywhere* (in this case, inside method `m` of an unrelated class `Foo` in some package `P1`), *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 `P1`) and can appear as intermediate results as the expression is evaluated. This breaks neither the FGJ+c confinement invariant nor that of Zhao et al. [ZPV06], 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 FGJ+c prevents such accesses.

For example, if `uMap` exposed one of its `uEntry` objects that are confined to package `u` and typed as `uEntry<U>`, then the following `p1Foo`'s method `m` variation will not be valid in FGJ+c. In particular the type of the subexpression involving the return type of `getFirstEntry()` used to call `getValue` will not type check using the FGJ+c rules.

```
class p1Foo {
  ...
  Bar<P2> m() {
    return (new uMap<Integer<World>,
      Bar<P2>, P1>).getFirstEntry().getValue();
  }
}
```

4.4.2 Towards Ownership

FGJ is a functional subset of Java that omits imperative aspects such as assignments, field updates, local variables, etc. The next chapter presents Featherweight Generic Ownership where FGJ is extended with imperative features following Pierce [Pie02]. I formulate an ownership invariant comparable to that of ownership types with few substantial additions to the visibility tests and preservation of owners over subtyping property 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. While it would be possible to formulate an ownership type system over FGJ, one would not be able to prove any of the ownership invariants without at least being able to distinguish between difference instances of objects. 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 dealing with all of the issues that are avoided when trying to concentrate on Generic Confinement alone.

4.5 Summary

Generic Confinement unifies two notions, genericity and confinement. In particular, in this chapter I 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 [BV99]. 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. This chapter proves this is possible for confinement while the next chapter extends this work to more discriminating systems such as ownership types.

Chapter 5

Featherweight Generic Ownership

Contents

In this chapter, I present Featherweight Generic Ownership (FGO) which provides object ownership support in an imperative programming language derived from Featherweight Generic Java (FGJ) [IPW01a]. FGO shows that a modern imperative language like

Java or C# can provide both ownership and generics in a unified and type sound manner. By formulating the FGO type system and proving it sound, I demonstrate that introducing Generic Ownership into Java will not break the language. The FGO type system has been presented in Potanin et al. [PNCB06b].

FGO builds on the ideas presented in Chapter 4 on Featherweight Generic Confinement (FGC). FGC demonstrates that Generic Confinement can be completely subsumed by a standard type generic language like FGJ. On the other hand, FGJ (and hence FGC) cannot reason about the relationships between different instances of classes required for object ownership because they define purely functional languages. FGO specifies an imperative language with heap and references, rather than a functional language as in the case of FGC and FGJ. The presence of object references allows FGO to provide full object ownership support, rather than confinement only support as in the case of FGC.

The FGO type system extends FGJ [IPW01a] with imperative features such as assignment, locations (modelling object references), and field updates. The Generic Ownership features of FGC, such as owner parameters and their preservation over the subtyping, are then introduced in a similar manner to Chapter 4. Additionally, support for object ownership in the form of the $this$ function is introduced.

The full FGO type system has only a few more rules than FGJ. In this chapter, I describe the FGO type system in detail, provide complete type soundness proofs, and prove ownership invariant theorems. FGO is the first type system to fully support confinement, object ownership, and type genericity (type polymorphism). FGO is backed up by a prototype implementation described in Chapter 6. Furthermore, the OGJ code examples in Chapters 3 and 4 are also valid in FGO.

5.1 Syntax

Figure 5.1 shows FGO's syntax. The syntax is derived from FGJ by adding expressions for locations, `let`, field update, and `null`. The figure contains definitions for syntactical terms corresponding to types (`T`), type variables (`X`), nonvariable types (`N`), class declarations (`L`), method declarations (`M`), and expressions (`e`).

The environment Δ contains mappings from variables to their types, mappings from type variables to nonvariable types, and mappings from locations to their types. I use the **<:** symbol to denote the subtyping relationship. There is no explicit constructor declaration: fields are initialised to `null`. This is important because ownership implies that objects owned by `This` cannot be created before the object to which they will belong. The latter would be possible if ownership transfer was supported, but this is left for future work.

FGO uses *CT* (class table) to denote a mapping from class names `C` to class declarations `L`. An FGO program is an expression with an appropriately initialised class table *CT*. FGO

$\mathtt{T} ::= \mathtt{X} \mid \mathtt{N}$	Type.
$\mathtt{N} ::= \mathtt{C} < \overline{\mathtt{T}} >$	Nonvariable type.
$\mathtt{L} ::= \mathtt{class}\ \mathtt{C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \lhd \mathtt{N}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}$	Class declaration.
$\mathtt{M} ::= < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\mathtt{return}\ \mathtt{e};\}$	Method declaration.
$\mathtt{e} ::= \mathtt{e_s} \mid l \mid l\ >\ \mathtt{e} \mid \mathtt{error}$	Expressions.
$\mathtt{e_s} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m} < \overline{\mathtt{T}} > (\overline{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{N}() \mid (\mathtt{N})\ \mathtt{e}$ $\mid \mathtt{e.f} = \mathtt{e} \mid \mathtt{let}\ \mathtt{x} = \mathtt{e}\ \mathtt{in}\ \mathtt{e} \mid \mathtt{null}$	Source expressions.
$v\ ::=\ l \mid \mathtt{null}$	Values.
$l \in \mathit{locations}$	Locations.
$\Delta = \{\mathtt{x}\ :\ \mathtt{T}\} \cup \{\mathtt{X} <: \mathtt{N}\} \cup \{l\ :\ \mathtt{N}\}$	Environment that maps (1) variables to their types, (2) type variables to nonvariable types, (3) locations to their types.
$P ::= \mathtt{C} \mid l$	Permission.
$S ::= \{l \mapsto \mathtt{N}(\overline{v})\}$	Store.

Figure 5.1: FGO Syntax

does not have a sequence expression that is normally denoted by a semicolon (;). A sequence $\mathtt{e}'; \mathtt{e}$ is a syntactic sugar that can be modelled with a `let` expression of the form:

$$\mathtt{let}\ \mathtt{x} = \mathtt{e}'\ \mathtt{in}\ \mathtt{e}$$

Alternatively, nested methods calling each other in sequence can emulate sequences. To simplify the presentation, I avoid such variants of expressions. I use semicolons to separate field and method declarations, as well as to denote the end of the method's expression.

FGO (just like FGJ+c earlier) adopts the syntactical idiosyncrasies of FGJ, such as $\overline{\mathtt{T}}\ \overline{\mathtt{x}}$ denoting a list of pairs: $\mathtt{T_1}\ \mathtt{x_1}, \ldots, \mathtt{T_n}\ \mathtt{x_n}$, rather than two lists of types and variables. The substitution notation used in FGJ+c follows FGJ so that $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}$ means that every occurrence of $\mathtt{X_i}$ in N is replaced by $\mathtt{T_i}$. Finally, an update notation for the store S is used as follows: $S' = S[l \mapsto \mathtt{N}(\overline{\mathtt{null}})]$ stands for an updated store S' where the mapping for location l is replaced to be a mapping from l to $\mathtt{N}(\overline{\mathtt{null}})$.

In more detail, T is a syntactical term for an FGO type that can be either a type variable (X) or a nonvariable type (N). A nonvariable type term (N) consists of a class name (C) and a list of type parameters ($\overline{\mathtt{T}}$). A class declaration (L) specifies a class name (C), bounds for every type variable used in the type parameter list ($\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}$), a nonvariable super type (N), field names with their types ($\overline{\mathtt{T}}\ \overline{\mathtt{f}}$), and a list of method declarations ($\overline{\mathtt{M}}$).

Each method declaration has a list of method type parameters supplying a bound for each type variable, followed by a return type, a method name, a list of method arguments and their types, and finally a return statement with the expression used when evaluating the method. The expression term e can be any of the expressions that can appear in the source of the program ($\mathtt{e_s}$), as well as a location (l), an expression $l\ >\ \mathtt{e}$ that captures the location

in the object store of the instance of the class that contains the method declaration of the method whose expression is being executed, or `error` arising from a bad cast or `null` dereference. The last three cannot appear in the source of the FGO program but appear during the evaluation of the reduction rules.

An expression l > `e` is created every time a method with receiver object l and method expression `e` is invoked. This expression preserves the information about the receiver location (l) so that it can be used in place of the permission P during the type checking of the subexpressions of `e`. Permissions are used by the visibility rules and the $this$ function described later in Section 5.2.2.

The source expressions ($\mathtt{e_s}$) include five FGJ expressions: variable, field access, method invocation, object creation, and cast; as well as field update, local variable creation, and `null`. Finally, being an imperative language, FGO includes values (v) that can be either a location (l) or `null`. Note that no primitive types are supported by FGO, since to reason about aliasing only references are required as values.

Permission P (a class or a location) is used to verify if a particular type with a certain owner can be present in the current expression as described in the visibility rules in Section 5.5. Store (S) is used to represent information about the heap, recording the object stored in each location and values of the object's fields.

FGO owner classes are just types, but I assume that *the* owner class of a particular class is syntactically distinguishable from the other owners present among the types:

$$\mathtt{O} ::= \mathtt{X^O} \mid \mathtt{N^O}$$

where `O` ranges over all owners, $\mathtt{X^O}$ ranges over owner variables, and $\mathtt{N^O}$ ranges over nonvariable owners such as `World` and `This`, as well as the owner classes corresponding to the packages. FGO uses capitals (`U`) for the owner class corresponding to a lower case package name (`u`). `This` with subscript l ($\mathtt{This}_l$) is used for the owner class corresponding to the owner of an object at location l. Pure FGO types and classes are written to include an owner as their last type parameter or argument. Pure types and classes can be distinguished using the following syntax:

$$\mathtt{N}_{pure} \ ::= \ \mathtt{C<}\overline{\mathtt{T}},\ \mathtt{O>}$$
$$\mathtt{L}_{pure} \ ::= \ \mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}},\ \mathtt{X^O} \lhd \mathtt{N^O > } \lhd \mathtt{N\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\mathtt{\}}$$

This syntactical distinction in no way means that owner classes are treated differently from any other types. It is only a convenient mechanism used to distinguish *the* owner of a particular class from the other owners present in its declaration and use.

$\Delta \vdash \texttt{T}\ \texttt{OK}$	Type T is OK.
$\Delta \vdash \texttt{T} <: \texttt{U}$	Type T is a subtype of type U.
$\Delta; P \vdash \texttt{e} : \texttt{T}$	Expression e is well typed w.r.t. permission *P*.
$\Delta; P \vdash \mathit{visible}(\texttt{e})$	Expression e is visible w.r.t. permission *P*.
$\Delta \vdash S$	Store (heap) is well formed.
$\Delta\texttt{OK}$	All locations in the environment are well-typed.
$\Delta \vdash <\overline{\texttt{Y}} \lhd \overline{\texttt{P}}>\ \texttt{T}\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\{\texttt{return}\ \texttt{e}_0;\}\ \texttt{FGO IN C},\ \texttt{C}^{\texttt{O}}$	Method m definition is OK.
$\texttt{class C} < \overline{\texttt{X}} \lhd \overline{\texttt{N}} > \lhd \texttt{N}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\}\ \texttt{FGO}$	Class C definition is OK.

Figure 5.2: FGO Judgements

π_C	the package owner class corresponding to class C
$\mathit{this}_P(\texttt{e})$	validates the use of "*this.*" calls in expression e
$\mathit{owner}_\Delta(\texttt{T})$	the owner of type T
$\mathit{visible}_\Delta(\texttt{O}, \texttt{C})$	owner O is visible in class C
$\mathit{visible}_\Delta(\texttt{T}, \texttt{C})$	type T is visible in class C

Figure 5.3: FGO Functions

5.2 Type Judgements and Functions

The FGO type system uses the type judgements shown in Figure 5.2. These are the well formed type judgement, the well formed subtype judgement, and the well typed expression judgement that come from the FGJ type system. The *visible* judgement for expressions is the same as in FGC in Chapter 4 and is very similar to the *visible* judgement used by Zhao et al. in CFGJ [ZPV06]. The store well-formedness, method and class definition judgements are standard for FGJ-style type systems [IPW01a]. Following the FGJ type system, the syntactical term Y corresponds to type variables and P corresponds to nonvariable types. The $\texttt{C}^{\texttt{O}}$ in a method definition judgement refers to the owner of class C where the method is declared.

FGO makes use of a number of functions to simplify the presentation, as shown in Figure 5.3. π_C is assumed to be an implicit lookup function; *this*, *owner*, and *visible* are described in detail in the rest of this chapter. While the *this* function is specific to FGO, *owner* and *visible* functions were already utilised in FGC as discussed in Chapter 4.

5.2.1 Lookup and Auxiliary Functions

Figure 5.4 contains the owner lookup function that is FGO specific as well as the dictionary functions from FGJ (for fields and methods). The *owner* function gives the owner of a type. The owner of a manifest class is found by traversing the class hierarchy. Owner lookup also allows *"naked" owners* — the owner classes on their own described in Section 3.5 — to be classified as owners of themselves.

Owner Lookup (FGO-OWNER):

$$
\begin{array}{lcl}
owner_{\Delta}(\mathtt{O}) & = & \mathtt{O} \\
owner_{\Delta}(\mathtt{X}) & = & owner_{\Delta}(\Delta(\mathtt{X})) \\
owner_{\Delta}(\mathtt{C}\!<\!\overline{\mathtt{T}},\ \mathtt{O}\!>) & = & \mathtt{O} \\
owner_{\Delta}(\mathtt{C}\!<\!\overline{\mathtt{T}}\!>) & = & owner_{\Delta}([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}), \text{where} \\
 & & CT(\mathtt{C}) = \mathtt{class\ C}\!<\!\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\!> \lhd \mathtt{N}\{\overline{\mathtt{T}'}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}
\end{array}
$$

Field Lookup:

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

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

Method Type Lookup:

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

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

Method Body Lookup:

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

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

Figure 5.4: FGO Lookup Functions

Field lookup uses class declarations and assumes that the root of the class hierarchy has no field declarations (see F-OBJECT) — this convention is following the FGJ type system. The result of the *fields* function is a list of fields, following FGJ it is assumed that fields are not overridden using the same name so that the formulation of the FGO type system is simpler. If the field name is undefined, then the field lookup function will return an empty list.

Method type and body lookups are similar to the field lookup except that they assume that the method name they are after is present at some level of the class hierarchy. If the method name is undefined, then method lookup functions will be undefined.

Figure 5.5 shows the FGJ function $bound_{\Delta}$ used to lookup the bound of a type — using the appropriate environment for the type variable and becoming an identity function for a nonvariable type. This figure also contains the subclassing rules from FGJ that enforce that subclass is a reflexive and transitive relation that is defined by the class declarations.

Bound of Type:

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

Valid Method Overriding:

$$\frac{mtype(\mathtt{m},\ \mathtt{N}) = <\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{Q}} > \overline{\mathtt{U}} \rightarrow \mathtt{U_0} \quad \Rightarrow\ \overline{\mathtt{P}}, \overline{\mathtt{T}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}](\overline{\mathtt{Q}}, \overline{\mathtt{U}}) \text{ and } \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \vdash \mathtt{T_0} <: [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}]\mathtt{U_0}}{override(\mathtt{m},\ \mathtt{N},\ <\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}} > \overline{\mathtt{T}} \rightarrow \mathtt{T_0})}$$

Figure 5.5: FGO Auxiliary Functions

The last part of the figure contains the *override* rule (identical to FGJ) that validates method overriding among the subclasses. This is used in the method rule in Figure 5.12. The reason we present the overriding rule from FGJ is for completeness of the FGO type system's presentation. In Chapter 4, the override rule and the lookup functions for fields and methods are also implicitly present in the FGJ+c type system, since the FGJ type system presented in Appendix A has them. Following FGJ, method overriding allows covariance of the return types, but not contravariance of the parameter types.

5.2.2 The This Function

The *this* function — see Figure 5.6 — is used extensively during the typing of the FGO expressions. This function helps enforce ownership, as it ensures that types involving `This` can only be used within the current object, that is, as part of message sends or field accesses upon `this`. Every occurrence of `This` in the type of a method call or field access is substituted with the result of calling the *this* function. If the type involves `This`, the expression will typecheck only if the target of the call or field access is `this`.

$$this_{\mathtt{C}}(\mathtt{this}) = \mathtt{This} \qquad this_{l}(l) = \mathtt{This}_{l} \qquad this_{P}(\ldots) = \bot$$

Figure 5.6: FGO This Function

In detail, there are two distinct places where the *this* function is used. They are distinguished by the permission P that is present on the left hand side of the expression type rules. The permission can be thought of as the *current context*, which denotes either the current class or package during the type checking of the program's source, or the current instance that is also known as `this` during run-time.

The first place is during the validation of FGO class declarations (in the class and method typing rules in Figure 5.12, which rely on expression typing in Figure 5.9). Here, the permission P contains the class `C` currently being validated. When typing a field access or method call inside `C`, the *this* function is called upon the expression $\mathtt{e_0}$ that is the target

of the field access ($\texttt{e}_0\texttt{.f}$) or method call ($\texttt{e}_0\texttt{.m}()$). Then, all occurrences of `This` in the types of the method or field are substituted by the result of the *this* function. If the target is `this` (e.g., `this.f`), then *this* function returns `This`, the substitution will replace `This` with itself, and so the expression typechecks, even if it involves `This` types. If the target is other than `this`, the *this* function returns an undefined ($\perp$) result, so `This` is substituted by $\perp$ (leaving other types unaffected), and any expressions with `This` types will fail to typecheck. Thus, FGO ensures that `This` types can only be used upon `this`.

The second place the *this* function appears is during the reduction of FGO expressions (e.g., R-METHOD in Figure 5.15). In this case, expression types include locations (l). Every occurrence of a `This` owner is replaced by a location specific $\texttt{This}_l$. The expression typing rules further ensure that every occurrence of `This` is made location specific. To achieve this, the T-CONTEXT rule (discussed later in this chapter) in Figure 5.9 ensures that the permission P represents the current location l. As the expression typing rules recurse into the structure of the expression `e`, every occurrence of `This` is replaced appropriately by this_l to be $\texttt{This}_l$.

In either case, the *this* function causes any expression containing an invalid use of the `This` owner class to have an undefined type. When any FGO program — a list of class declarations followed by an expression — is type checked, a proof tree is constructed that during the validation of the expression type expands the *this* function. If the expression contains an invalid use of `This`, then the resulting expression type is undefined and thus the whole FGO program fails to type check.

FGO type soundness guarantees that any well formed FGO class will not allow access to a field or method with owner $\texttt{This}_l$ during the reduction unless the current execution context is location l.

Since the presence of the *this* function implies that operations on the objects owned by `This` are restricted to the current instance, it becomes impossible to pass the owned object to another instance. While this is a good thing since it makes it easy to prove that no other object gets the reference to the object owned by `This`, this approach can be considered too restrictive. For example, the type system cannot detect a situation where the current object is stored in a different variable:

```
class Foo {
  private Secret<This> s;
  void bar() {
    Foo me = this;
    this.s; // OK
    me.s; // Not OK
  }
}
```

Such restriction is to be expected since the FGO type system does not perform any data

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

(WF-TYPE):

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \ \lhd\ \mathtt{N}\,\{\ldots\} \quad \Delta \vdash \mathtt{N} <: \mathtt{Object} < \mathtt{O} > \quad \Delta \vdash \mathtt{O} <: \mathtt{World} \quad \Delta \vdash \overline{\mathtt{T}}\ \mathtt{OK} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}} \quad \forall \mathtt{T} \in \overline{\mathtt{T}} : owner_{\Delta}(\mathtt{C} < \overline{\mathtt{T}} >) <: owner_{\Delta}(\mathtt{T})}{\Delta \vdash \mathtt{C} < \overline{\mathtt{T}} > \mathtt{OK}}$$

Figure 5.7: FGO Type Well-Formedness Rules

flow analysis required to detect such uses of the `this` variable.

5.3 Well Formed Types and Subtyping

FGO's type well-formedness rules are shown in Figure 5.7. These are the same as those of FGJ, except that the root of the class hierarchy — `Object` — is parameterised (just like the root of FGJ+c class hierarchy). The grey clause in the type formation rule ensures that FGO supports deep ownership: WF-TYPE enforces the nesting of owner parameters essential to ensure *owners as dominators* (see Section 2.3) object encapsulation. The owner nesting follows the rule that the main owner parameter is *outside or equal to* the rest of the owner parameters as discussed in Section 3.9. The nesting of owner parameters is used to prove the ownership invariant theorem at the end of this chapter.

The nesting of owners is checked both at the class declaration time (using the class declaration rule described later in Section 5.6) and at the type instantiation time (using WF-TYPE rule in Figure 5.7). The owner of the class has to be *inside* all of the other owners coming from its various type parameters to prohibit occurrences similar to:

```
class BreaksDeepOwnership<SecretOwner extends World,
  Owner extends World> extends Object<Owner> {
    Object<World> doIt() {
      return new BreaksDeepOwnership<This, World>();
    }
}
```

This example shows how an object owned by someone can be passed around the entire object graph since its owner is `World`, while having permission to access inner workings of a particular private instance. This can lead to more than one path from the root of the object graph to the owned instance, thus breaking the owners-as-dominators property.

Please observe that although Δ contains three possible syntactic categories in its domain ($\mathtt{X}, \mathtt{x}, l$), only type variables ($\mathtt{X}$) are covered by the WF-VAR rule. Also, most of

Subtyping:

$$\frac{}{\Delta \vdash \mathtt{T} <: \mathtt{T}} \text{ (S-REFL)} \qquad \frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \quad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}} \text{ (S-TRANS)}$$

$$\frac{}{\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})} \text{ (S-VAR)} \qquad \frac{CT(\mathtt{C}) = \mathtt{class\ C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \ \lhd \ \mathtt{N}\,\{\ldots\}}{\Delta \vdash \mathtt{C} < \overline{\mathtt{T}} > <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}} \text{ (S-CLASS)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \ \lhd \ \mathtt{N}\,\{\ldots\}}{\Delta \vdash \pi_{\mathtt{C}} <: \mathtt{World}} \qquad \frac{l \in dom(\Delta)}{\Delta \vdash \mathtt{This}_l <: owner_{\Delta}(\Delta(l))} \text{ (S-OWNER)}$$

Figure 5.8: FGO Subtyping Rules

these rules are also present implicitly in the FGJ+c type system in Chapter 4 since they are standard FGJ rules (except for the parts dealing with instance owners and with deep ownership).

Figure 5.8 shows FGO's subtyping rules. Apart from S-OWNER they are taken verbatim from FGJ. The first two rules enforce that the subtyping relationship is reflexive and transitive. The second two rules enforce that type variables are subtypes of their bounds and that the subtyping information for classes comes from their declarations.

The first of the two S-OWNER rules ensures that `World` forms the top of the owner class hierarchy which any package owner class extends directly. The second rule states that the location-specific owner class $\mathtt{This}_l$ extends the owner of the class whose instance is stored at location l. These two subtyping rules together are required for the ownership invariant's definition of owner classes being *inside* one another presented in Section 5.10. In particular, it ensures that `This` is *inside* `Owner` is *inside* `World` at all times. The owner nesting (*inside*) relationship is thus made to follow the owner subtyping relationship for the convenience of the ownership invariant's proof.

The rest of the owner class hierarchy (for location-specific owners and for owner variables, including `This`) is built up during type checking as discussed in the class rule description in Section 5.6. Finally, although Δ contains three possible syntactic categories in its domain ($\mathtt{X}$, $\mathtt{x}$,l), only locations (l) are covered by the second S-OWNER rule.

5.4 Expressions

Figure 5.9 shows the expression typing rules. These are the standard FGJ rules with added support for locations, assignment, `null`, and `let` expressions [Pie02]. These rules define the types and possible well-formedness constraints for all possible expressions in the source of the FGO program or during its execution. They utilise both the environment Δ and the permission P denoting the current class or instance depending on the use of the expression typing rule. The reduction rules discussed later in this chapter contain the

(T-FIELD):

$$\frac{\Delta; P \vdash \mathtt{e}_0 \ : \ \mathtt{T}_0 \quad \Delta \vdash \mathtt{T}\ \mathtt{OK} \quad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \quad \mathtt{T} = [\mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\mathtt{T}_i}{\Delta; P \vdash \mathtt{e}_0.\mathtt{f}_i \ : \ \mathtt{T}}$$

(T-FIELD-SET):

$$\frac{\Delta; P \vdash \mathtt{e}_0 \ : \ \mathtt{T}_0 \quad \Delta; P \vdash \mathtt{e} \ : \ \mathtt{T} \quad \Delta \vdash \mathtt{T}\ \mathtt{OK} \quad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \quad \mathtt{T} = [\mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\mathtt{T}_i}{\Delta; P \vdash \mathtt{e}_0.\mathtt{f}_i = \mathtt{e} \ : \ \mathtt{T}}$$

(T-METHOD):

$$\frac{\begin{array}{c}\forall \mathtt{V}' \in \overline{\mathtt{V}} \ : \ (\Delta \vdash \mathtt{V}'\ \mathtt{OK} \ \vee \ \mathtt{V}' <: \mathtt{World}) \ \wedge \ (\ \mathit{owner}_\Delta(\mathtt{T}_0) <: \mathit{owner}_\Delta(\mathtt{V}')\ ) \\ \mathit{mtype}(\mathtt{m},\ \mathit{bound}_\Delta(\mathtt{T}_0)) = <\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}} > \overline{\mathtt{U}} \to \mathtt{U} \\ \Delta; P \vdash \overline{\mathtt{e}} \ : \ \overline{\mathtt{S}} \quad \Delta; P \vdash \mathtt{e}_0 \ : \ \mathtt{T}_0 \quad \Delta \vdash \mathtt{T}\ \mathtt{OK} \\ \mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ \mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\mathtt{U} \quad \Delta \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ \mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\overline{\mathtt{P}} \\ \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ \mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\overline{\mathtt{U}}\end{array}}{\Delta; P \vdash \mathtt{e}_0.\mathtt{m} < \overline{\mathtt{V}} > (\overline{\mathtt{e}}) \ : \ \mathtt{T}}$$

(T-CAST):

$$\frac{\Delta \vdash \mathtt{N}\ \mathtt{OK} \quad \Delta; P \vdash \mathtt{e}_0 \ : \ \mathtt{T}_0}{\Delta; P \vdash (\mathtt{N})\mathtt{e}_0 \ : \ \mathtt{N}}$$

(T-LET):

$$\frac{\Delta; P \vdash \mathtt{e}_0 \ : \ \mathtt{T}_0 \quad \Delta,\ \mathtt{x} : \mathtt{T}_0; P \vdash \mathtt{e} \ : \ \mathtt{T}}{\Delta; P \vdash \mathtt{let}\ \mathtt{x} = \mathtt{e}_0\ \mathtt{in}\ \mathtt{e} \ : \ \mathtt{T}}$$

$$\frac{\Delta; l \vdash \mathtt{e} \ : \ \mathtt{T}}{\Delta; P \vdash l \ > \ \mathtt{e} \ : \ \mathtt{T}}\ \text{(T-CONTEXT)} \qquad \frac{\Delta \vdash \mathtt{N}\ \mathtt{OK}}{\Delta; P \vdash \mathtt{new}\ \mathtt{N}() \ : \ \mathtt{N}}\ \text{(T-NEW)}$$

$$\frac{\Delta \vdash \mathtt{T}\ \mathtt{OK}}{\Delta; P \vdash \mathtt{error} \ : \ \mathtt{T}}\ \text{(T-ERROR)} \qquad \frac{\Delta \vdash \mathtt{T}\ \mathtt{OK}}{\Delta; P \vdash \mathtt{null} \ : \ \mathtt{T}}\ \text{(T-NULL)}$$

$$\frac{}{\Delta; P \vdash \mathtt{x} \ : \Delta(\mathtt{x})}\ \text{(T-VAR)} \qquad \frac{}{\Delta; P \vdash l \ : \ \Delta(l)}\ \text{(T-LOC)}$$

Figure 5.9: FGO Expression Typing

behaviour of object locations l present in some of the expressions. Expressions containing locations cannot occur in FGO program source, and only appear during the reductions.

The T-FIELD rule constrains a field access. The index i is used to refer to the position of the field in the fields list returned by the *fields* function. This rule checks that the expression that is the receiver of the field access is a well typed FGO expression and that the field type is a well formed FGO type. The T-FIELD rule also ensures that the field exists in the corresponding class declaration and applies the *this* function check to make sure that if a field is private (has owner `This`) then it can only be accessed using a `this` call.

The T-FIELD-SET rule describes an expression that sets the value of a field. This rule checks that the receiver expression is well typed in FGO and that the expression being assigned is well typed. This rule also checks that the resulting type is a well formed FGO type and that the field requested exists. Finally, it applies the $this$ function to ensure that fields owned by `This` are not used by other instances, similar to the T-FIELD rule above.

The T-METHOD rule — the most complex expression rule — checks that the method type parameters are OK in FGO or are valid owner variables (hence both a well-formedness check *and* allowing `W'` to be a subtype of `World`). These checks allow "naked" owners discussed in Chapter 3. This rule then checks that owners of method type parameters are properly nested inside the owner of the class that the method is part of (supplied via permission P) — this nesting is required by the owners-as-dominators property in the same manner as the grey check in WF-TYPE rule earlier.

The rest of the checks are similar to FGJ: the T-METHOD rule looks up the method's type, checks that the method arguments are well typed FGO expressions and that the resulting type is OK in FGO. Finally, the rule applies the $this$ function to the method's return type, type parameters, and arguments to make sure that the methods using private (with owner `This`) instances are only called on `this`.

The T-CAST rule checks if the expression being cast is well typed. The reduction rules will check the actual type of the expression and if it matches the requested type, failing with run-time error if necessary. It is possible to avoid some run-time casting failures using a static check whether the upcast or downcast is valid and failing to type check if neither succeed, which is the technique employed by FGJ [IPW01a]. But since in the presence of locations and `null`, I already model the `error` arising in the case of `null` dereferences, one may as well model casting `error` in the same way for simplicity.

The T-LET rule checks that the expression that gets the local variable (`e`) and the expression that is the local variable ($\texttt{e}_0$) are both well typed in FGO. An appropriate type for the local variable `x` is stored in the typing environment Δ before type checking the expression `e`.

The T-CONTEXT rule is only used during the type preservation proof — it ensures that the expression is well typed with respect to the receiver location l. This makes it possible for the other rules to replace the owner class `This` with a correct location-specific owner $\texttt{This}_l$. This is one of the expressions that cannot appear in the source of a FGO program.

The T-NEW rule checks that the type of the instance being created is a well formed FGO type. The T-ERROR and T-NULL rules ensure that `error` and `null` are well formed FGO types. Finally, the T-VAR and T-LOC rules lookup an appropriate type for variables and locations using the typing environment Δ.

The important observation about the expression rules is that these are not sufficient to ensure ownership. An additional set of visibility rules is required to prevent incorrect accesses for different owners. These are presented in the next section.

$$visible_{\Delta}(\mathtt{T},\ \mathtt{C}) \quad = \quad visible_{\Delta}(owner_{\Delta}(\mathtt{T}),\ \mathtt{C}) \qquad \text{(V-TYPE)}$$

$$visible_{\Delta}(\mathtt{O},\ \mathtt{C}) \quad = \quad \mathtt{O} \in owners(\mathtt{C}) \cup \{\mathtt{This}, \pi_{\mathtt{C}}, \mathtt{World}\} \qquad \text{(V-OWNER)}$$

$$\text{where } owners(\mathtt{C}) \quad = \quad \begin{cases} \{owner_{\Delta}(\mathtt{N}') \mid \mathtt{N}' \in \overline{\mathtt{N}}, \mathtt{N}\}, \\ \qquad \text{if } CT(\mathtt{C}) = \mathtt{class\ C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \lhd \mathtt{N}\{\ldots\} \\ \{\mathtt{X^O}\} \cup \{owner_{\Delta}(\mathtt{N}') \mid \mathtt{N}' \in \overline{\mathtt{N}}\}, \\ \qquad \text{if } CT(\mathtt{C}) = \mathtt{class\ C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}},\ \mathtt{X^O} \lhd \mathtt{N^O} > \lhd \mathtt{N}\{\ldots\} \end{cases}$$

Figure 5.10: FGO Type and Owner Visibility Rules

5.5 Visibility

Visibility plays an essential role in FGO and FGC. The term visibility is similar to FGC and ConfinedFGJ [ZPV06] rules except accounting for the additional expressions present in FGO. Owner and type visibility is FGC and FGO specific and makes use of the owner class `This` for the FGO version of the rules.

Figure 5.10 shows the owner visibility rule that checks if an owner `O` is visible inside class `C`. This is the case if the owner is `World`, belongs to the same package as `C`, or is an owner of one of the type parameters used when instantiating `C`. Supplying an actual owner parameter to a class gives that class permission to access everything owned by that parameter. This, for example, can allow a type polymorphic class to have private access to more than one package.

FGO does not restrict visibility of the `This` owner class, and relies on the *this* function described earlier to stop illegal uses of `This`. Type visibility (V-TYPE) checks the owner of a given type for visibility.

Term visibility (Figure 5.11) recursively checks all the types involved in the possible expressions of FGO to make sure that they are visible in a given class `C`. Since these checks are performed on class declarations and are not required during reduction, locations are not present in these expressions. For example, it means that T-CONTEXT expression is not covered by the visibility rules).

5.6 Classes and Methods

Figure 5.12 presents FGO class definition rules for standard FGO classes (FGO-CLASS-PURE) and manifest classes described in Chapter 4 (FGO-CLASS-MANIFEST). The figure also presents the FGO method definition rule.

The first rule, FGO-CLASS-PURE, defines a standard FGO class. The first line

$$\frac{\Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{e_0}) \quad \Delta;\mathtt{C} \vdash \mathtt{e.f_i} : \mathtt{T_0} \quad \mathit{visible}_\Delta(\mathtt{T_0},\ \mathtt{C})}{\Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{e_0.f_i})} \text{ (V-FIELD)}$$

(V-FIELD-SET):

$$\frac{\Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{e_0}) \ \ \Delta;\mathtt{C} \vdash \mathtt{e_0.f_i} : \mathtt{T_0} \ \ \mathit{visible}_\Delta(\mathtt{T_0},\ \mathtt{C}) \ \ \Delta;\mathtt{C} \vdash \mathtt{e} : \mathtt{T} \ \ \mathit{visible}_\Delta(\mathtt{T},\ \mathtt{C})}{\Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{e_0.f_i} = \mathtt{e})}$$

(V-METHOD):

$$\frac{\Delta;\mathtt{C} \vdash \mathtt{e_0.m}(\overline{\mathtt{e_0}}) : \mathtt{T_0} \quad \mathit{visible}_\Delta(\mathtt{T_0},\ \mathtt{C}) \quad \Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{e_0}) \quad \Delta;\mathtt{C} \vdash \mathit{visible}(\overline{\mathtt{e}})}{\Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{e_0.m}(\overline{\mathtt{e}}))}$$

$$\frac{\Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{e_0}) \quad \mathit{visible}_\Delta(\mathtt{N},\ \mathtt{C})}{\Delta;\mathtt{C} \vdash \mathit{visible}((\mathtt{N})\ \mathtt{e_0})} \text{ (V-CAST)}$$

$$\frac{\begin{array}{c}\Delta;\mathtt{C} \vdash \mathtt{e_0} : \mathtt{T_0} \quad \mathit{visible}_\Delta(\mathtt{T_0},\ \mathtt{C}) \quad \Delta;\mathtt{C} \vdash \mathtt{x} : \mathtt{T_x} \\ \mathit{visible}_\Delta(\mathtt{T_x},\ \mathtt{C}) \quad \Delta;\mathtt{C} \vdash \mathtt{e} : \mathtt{T} \quad \mathit{visible}_\Delta(\mathtt{T},\ \mathtt{C})\end{array}}{\Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{let\ x} = \mathtt{e_0\ in\ e})} \text{ (V-LET)}$$

$$\frac{\Delta;\mathtt{C} \vdash \mathtt{x} : \mathtt{T} \quad \mathit{visible}_\Delta(\mathtt{T},\ \mathtt{C})}{\Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{x})} \text{ (V-VAR)} \qquad \frac{\mathit{visible}_\Delta(\mathtt{N},\ \mathtt{C})}{\Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{new\ N}())} \text{ (V-NEW)}$$

$$\frac{}{\Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{error})} \text{ (V-ERROR)} \qquad \frac{}{\Delta;\mathtt{C} \vdash \mathit{visible}(\mathtt{null})} \text{ (V-NULL)}$$

Figure 5.11: FGO Term Visibility Rules

initialises the FGO type environment (Δ) with information about subtype relationships read from the class declaration, and it also initialises the missing owner variables as described in the placeholder owners function in the end of this section. The second line ensures that the current class's owner ($\mathtt{X^0}$) is nested inside the other owners. This grey clause allows us to check deep ownership when proving an ownership invariant. The second line also checks the well-formedness of the super class and the types of the fields. The third line of the rule allows "naked" owners as type parameters and ensures that the method declarations are valid for the current class and its owner. Finally, the fourth line ensures that the super class (`N`) has the same owner ($\mathtt{X^0}$) and checks that the current classes' principal owner bound ($\mathtt{N^0}$), field types, and type parameters are all visible inside the current class. FGO checks the visibility of the owner bound $\mathtt{N^0}$ rather than owner $\mathtt{X^0}$ to disallow declarations that might try for example to confine a class declared in package `p` to package `u` making it unusable. The latter will be prevented by the visibility check since owner `U` is invisible inside package `p`.

The second rule (FGO-CLASS-MANIFEST) deals with the manifest FGO classes — classes that have a fixed owner taken from one of their superclasses. These are usually used to represent standard FGJ (or Java) classes with a fixed owner `World`. The FGO-CLASS-MANIFEST rule is very similar to the previous "pure" rule, except that the owner is

FGO Class Definition (FGO-CLASS-PURE):

$$\frac{\begin{array}{c}\Delta = \{\overline{\mathtt{X}} <: \overline{\mathtt{N}}, \mathtt{X}^{\mathtt{0}} <: \mathtt{N}^{\mathtt{0}}, \mathtt{This} <: \mathtt{X}^{\mathtt{0}}\} \cup \mathit{placeholderowners}_{\Delta}(\overline{\mathtt{N}}) \\ \cup \left(\bigcup_{\mathtt{X}' \in \overline{\mathtt{X}}} \{\mathtt{X}^{\mathtt{0}} <: \mathit{owner}_{\Delta}(\mathtt{X}')\}\right) \qquad \Delta \vdash \mathtt{N}, \overline{\mathtt{T}}\ \mathtt{OK} \\ \forall \mathtt{N}' \in \overline{\mathtt{N}} : \Delta \vdash \mathtt{N}'\ \mathtt{OK} \vee \Delta \vdash \mathtt{N}' <: \mathtt{World} \qquad \Delta \vdash \overline{\mathtt{M}}\ \mathtt{FGO\ IN\ C},\ \mathtt{X}^{\mathtt{0}} \\ \mathtt{N} = \mathtt{D} < \overline{\mathtt{T}'}, \mathtt{X}^{\mathtt{0}} > \qquad \mathit{visible}_{\Delta}(\mathtt{N}^{\mathtt{0}},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta}(\overline{\mathtt{T}},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta}(\overline{\mathtt{N}},\ \mathtt{C})\end{array}}{\mathtt{class\ C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}},\ \mathtt{X}^{\mathtt{0}} \lhd \mathtt{N}^{\mathtt{0}} > \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \mathtt{FGO}}$$

FGO (Manifest) Class Definition (FGO-CLASS-MANIFEST):

$$\frac{\begin{array}{c}\Delta = \{\overline{\mathtt{X}} <: \overline{\mathtt{N}}, \mathtt{This} <: \mathit{owner}_{\Delta}(\mathtt{N})\} \cup \mathit{placeholderowners}_{\Delta}(\overline{\mathtt{N}}) \\ \cup \left(\bigcup_{\mathtt{X}' \in \overline{\mathtt{X}}} \{\mathit{owner}_{\Delta}(\mathtt{N}) <: \mathit{owner}_{\Delta}(\mathtt{X}')\}\right) \qquad \Delta \vdash \mathtt{N}, \overline{\mathtt{T}}, \overline{\mathtt{N}}\ \mathtt{OK} \\ \Delta \vdash \overline{\mathtt{M}}\ \mathtt{FGO\ IN\ C},\ \mathit{owner}_{\Delta}(\mathtt{N}) \\ \mathit{visible}_{\Delta}(\mathtt{N},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta}(\overline{\mathtt{T}},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta}(\overline{\mathtt{N}},\ \mathtt{C})\end{array}}{\mathtt{class\ C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \mathtt{FGO}}$$

FGO Method Definition (FGO-METHOD):

$$\frac{\begin{array}{c}\Delta' = \Delta \cup \{\overline{\mathtt{Y}} <: \overline{\mathtt{P}}\} \cup \mathit{placeholderowners}_{\Delta}(\overline{\mathtt{N}}) \\ \cup \left(\bigcup_{\mathtt{Y}' \in \overline{\mathtt{Y}}} \{\mathtt{C}^{\mathtt{0}} <: \mathit{owner}_{\Delta}(\mathtt{Y}')\}\right) \qquad \Delta' \vdash \overline{\mathtt{T}}, \mathtt{T}\ \mathtt{OK} \\ \Delta' \vdash \forall \mathtt{P}' \in \overline{\mathtt{P}} : (\mathtt{P}'\ \mathtt{OK}) \vee (\mathtt{P}' <: \mathtt{World}) \qquad CT(\mathtt{C}) = \mathtt{class\ C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \ \lhd \mathtt{N}\ \{\ldots\} \\ \mathit{visible}_{\Delta'}(\overline{\mathtt{T}},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta'}(\mathtt{T},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta'}(\overline{\mathtt{P}},\ \mathtt{C}) \\ \Delta',\ \overline{\mathtt{x}} : \overline{\mathtt{T}},\ \mathtt{this} : \mathtt{C} < \overline{\mathtt{X}} > ; \mathtt{C} \vdash \mathit{visible}(\mathtt{e}_{\mathtt{0}}) \\ \Delta',\ \overline{\mathtt{x}} : \overline{\mathtt{T}},\ \mathtt{this} : \mathtt{C} < \overline{\mathtt{X}} > ; \mathtt{C} \vdash \mathtt{e}_{\mathtt{0}} : \mathtt{S} \\ \Delta' \vdash \mathtt{S} <: \mathtt{T} \qquad \mathit{override}(\mathtt{m},\ \mathtt{N},\ < \overline{\mathtt{Y}} \lhd \overline{\mathtt{P}} > \overline{\mathtt{T}} \rightarrow \mathtt{T})\end{array}}{\Delta \vdash < \overline{\mathtt{Y}} \lhd \overline{\mathtt{P}} > \mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}_{\mathtt{0}};\ \}\ \mathtt{FGO\ IN\ C},\ \mathtt{C}^{\mathtt{0}}}$$

Figure 5.12: FGO Class and Method Rules

looked up from the superclass and is not present explicitly in the current class declaration. Although FGO uses syntax to distinguish *the* classes' owner, the other type parameters of both classes and methods *can be* either types or owners without any special treatment from the type system — a benefit of Generic Ownership merging type parameters and ownership types.

The first line of the FGO-CLASS-MANIFEST rule defines the FGO type environment (Δ) with the subtype information and initialising placeholder owners as described at the end of this section. The second line enforces the owner nesting required of any ownership type system providing deep ownership and checks that the types involved are well formed FGO types. The third line checks the method declarations and the final, fourth line checks that all the types involved are visible in the current class.

To summarise, both of the class rules check that (1) all the types involved (types of fields and type parameters) are *visible* within the domain of the owner of the class being declared or its superclass; (2) all the types are *well formed* FGO types; and (3) all the method declarations are valid. The grey clauses ensure that the owner nesting (the

distinguished owner is *inside* the owners of the other type parameters) is preserved for the purposes of the deep ownership invariant theorem proven below in Section 5.10. The difference between the two class rules is that the owner of the class has to be looked up for a manifest version and is provided explicitly for the pure version.

The method typing rule in Figure 5.12 adds additional subtyping relationships and additional placeholder owners to the environment Δ supplied by the class declaration rule. The second line checks the owner nesting required for the deep ownership with respect to the owner of the class inside which the method being checked is declared. The second line also checks the well-formedness of the types of method arguments and a method's return type. The third line allows the method type parameters to be owner classes on their own and specifies the class declaration used for the class inside which the current method is declared so that the method rule can refer to the type parameters used in the class declaration. The fourth line checks the visibility of all the types involved in method declaration. The fifth and sixth lines check the visibility and well-formedness of the type of the method expression. To perform these checks on the expression, the assumptions on the left about the environment include the types for `this` and the method parameters. The checks also specify the permission to be the current class `C` so that this context can be used to detect any illegal accesses to package owner classes. Finally, the sixth line, in exactly the same manner as FGJ, checks the type of the expression with respect to the method return type and verifies the validity of method overriding (if applicable) — this preserves Java's requirement that a method with the same name and arguments has to return a result that is a proper subtype of the super class's method result type it overrides. The contravariance of method arguments is omitted for simplicity.

Additionally, FGO allows class declarations to use implicit *placeholder owner parameters* in formal type parameter bounds. Consider:

```
class List<E extends Foo<FO>, Owner extends World> { ... }
```

In this example, `List`'s formal parameter `E` can only be bounded by the actual type parameters that are subclasses of `Foo`; the owner of that type parameter (`FO`) can be different from the owner of the list (`Owner`) — although `FO` has to be "outside" `Owner`. Since FGO (like FGJ) requires every type variable to be bounded, one needs to make sure that the FGO type environment contains an appropriate mapping for implicit placeholder parameters like `FO`. The function $placeholderowners$ in Figure 5.13 does exactly that, by making sure that any (placeholder) owner used in the type bounds is recorded in the FGO environment Δ as being a subtype of `World` (unless the owner already has a bound declared for it explicitly, in which case it is used instead of `World`). This ensures that every owner present in the class declaration is bound, whether implicitly or explicitly. The FGO well formed types that are not owners are not affected by the presence of owners, since all of the owners are subtypes of `World` and not subtypes of `Object<O>`. These two

Placeholder Owners Function:

$$
\begin{array}{lcl}
placeholderowners_{\Delta}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) & = & \{owner_{\Delta}(\texttt{C<}\overline{\texttt{T}}\texttt{>})\texttt{<: World}\} \cup \\
 & & \cup\ placeholderowners_{\Delta}(\overline{\texttt{T}}) \\
 & & \text{if } owner_{\Delta}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) \notin dom(\Delta) \\
placeholderowners_{\Delta}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) & = & placeholderowners_{\Delta}(\overline{\texttt{T}}) \\
 & & \text{otherwise} \\
placeholderowners_{\Delta}(\texttt{X}) & = & \{\}
\end{array}
$$

Figure 5.13: FGO Placeholder Owners Function

hierarchies are shown in Figure 3.6 from Section 3.8. `Any` type bound mechanism in FGJ+c described in the previous chapter achieves a similar goal to that of the $placeholderowners$ function by allowing any owner to be used in a particular type parameter in an FGJ+c class.

Figure 5.13 shows the $placeholderowners$ function definition, which accepts a type. For every nonvariable type $placeholderowners$ adds its owner (the last type variable in a pure FGO class) to the type environment Δ unless it is already present. Then, for every type parameter, $placeholderowners$ recursively calls itself to add any additional owners than might not be recorded in Δ. The $placeholderowners$ function recurses into the subexpressions for complex type expressions until everything has been expanded.

5.7 Representing the Heap

This section addresses the representation of the heap in the FGO type system. The store typing rules shown in Figure 5.14 are mostly standard [CD02, AC04]. The mapping Δ contains the types for each location and the FGO-STORE-WF rule in Figure 5.14 ensures that every one of the types is well formed. The mapping S maps the locations to the types of classes that are instantiated at these locations together with further location values for each field in these instances. The main FGO-STORE rule ensures that not only the types are well formed, but also that each field location's type is a well formed FGO type and is a correct subtype of the declared field type. The FGO type system does not have any explicit ownership constraints in the store rule — the benefit of ownership information being part of the type is that subtyping ensures that none of the ownership constraints are broken. Finally, in this rule FGO only considers the part of the environment Δ that maps locations l to their types, which is used to validate the domain of S. The difference between $S[l]$ and $\Delta(l)$ is that $S[l]$ returns the type of the location together with location values stored in each field (e.g., $\texttt{N}(\overline{v})$) and $\Delta(l)$ returns the type of the location (e.g., `N`).

Store Well-Formedness (FGO-STORE-WF):

$$\frac{\forall l \in dom(\Delta) : \Delta \vdash\ \Delta(l)\ \texttt{OK}}{\Delta\ \texttt{OK}}$$

Store Typing (FGO-STORE):

$$\frac{\begin{array}{c}\Delta\ \texttt{OK} \qquad dom_l(\Delta)^\dagger = dom(S) \\ S[l] = \texttt{N}(\overline{v}) \Longrightarrow \Delta(l) = \texttt{N} \qquad \Delta(l) = \texttt{N} \Longrightarrow \exists \overline{v} : S[l] = \texttt{N}(\overline{v}) \\ (S[l,i] = l') \wedge (\mathit{fields}(\Delta(l)) = \overline{\texttt{T}}\ \overline{\texttt{f}}) \Longrightarrow \Delta \vdash \Delta(l') <: [\texttt{This}_l/\texttt{This}]\texttt{T}_i \\ (S[l,\ i] = l') \Longrightarrow \Delta \vdash\ \Delta(l')\ \texttt{OK}\end{array}}{\Delta \vdash S}$$

† $dom_l(\Delta)$ refers to the domain of Δ that is restricted to *locations* only.

Figure 5.14: FGO Store

5.8 Reduction Rules

Figure 5.15 shows the small step semantics reduction rules. Again, these are standard given the expressions that FGO supports. The notation shows how an expression e and store S together reduce to a new expression e′ with possibly an updated store S': $\texttt{e}, S \rightarrow \texttt{e}', S'$. Symbol l denotes locations and v denotes values of particular fields (which can be `null`).

R-NEW reduces a newly created instance to the newly created location l, storing only `null` for each of the class's fields. R-FIELD uses the *fields* lookup function and store S to reduce the expression to the value stored in a particular field. R-FIELD-SET modifies the store S by replacing the value stored in a particular field $\texttt{f}_i$ and, similarly to the behaviour of the R-FIELD rule, reduces to the new value that was just stored. R-METHOD looks up the appropriate method body using the *mbody* function and reduces to an expression of the form $l >$ e where the actual arguments are substituted in place of formal parameters, the current location l is substituted in place of `this`, and the current instance's owner $\texttt{This}_l$ is substituted in place of owner `This`.

Neither field nor method accesses are allowed on `null`, which is dealt with by R-*-NULL reduction rules reducing such expressions to `error`. FGO does not have to explicitly prohibit l from being `null` in the field and method access rules due to the FGO syntax only allowing values (v) to be `null` and not locations (l). The FGO's `error` captures non-type errors that cannot be captured easily by a decidable type system, on the other hand `error` is well-typed to allow type preservation proof for the FGO type system to deal only with well typed expressions.

R-CAST and R-BAD-CAST deal with type casts by allowing them if the type of the location is a subtype of the cast target type, and reducing to `error` otherwise. R-CONTEXT removes the no longer necessary information about the receiver of the method call, once the reduction has evaluated the expression's right hand side to a value v. Finally R-LET substitutes every occurrence of variable x in the expression $\texttt{e}_0$ with its value v.

Figure 5.16 presents FGO's context reduction rule that defines the evaluation order

(R-NEW):

$$\frac{l \notin dom(S) \qquad S' = S[l \mapsto \mathtt{N}(\overline{\mathtt{null}})] \qquad |\overline{\mathtt{null}}| = |fields(\mathtt{N})|}{\mathtt{new}\ \mathtt{N}(), S \rightarrow l, S'}$$

(R-FIELD):

$$\frac{S[l] = \mathtt{N}(\overline{v}) \qquad fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{l.\mathtt{f}_i, S \rightarrow v_i, S}$$

(R-FIELD-SET):

$$\frac{S[l] = \mathtt{N}(\overline{v}) \qquad fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad S' = S[l \mapsto \mathtt{N}(v_0, ..., v_{i-1}, v, v_{i+1}, ..., v_{|\overline{\mathtt{f}}|})]}{l.\mathtt{f}_i = v, S \rightarrow v, S'}$$

(R-METHOD):

$$\frac{S[l] = \mathtt{N}(\overline{v_l}) \qquad mbody(\mathtt{m} < \overline{\mathtt{V}} >, \mathtt{N}) = \overline{\mathtt{x}}.\mathtt{e}_0}{l.\mathtt{m} < \overline{\mathtt{V}} >(\overline{v}), S \rightarrow l\ >\ [\overline{v}/\overline{\mathtt{x}}, l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]\mathtt{e}_0, S}$$

(R-*-NULL):

$$\frac{}{\mathtt{null}.\mathtt{m} < \overline{\mathtt{V}} >(\overline{v}), S \rightarrow \mathtt{error}, S} \qquad \frac{}{\mathtt{null}.\mathtt{f}_i, S \rightarrow \mathtt{error}, S} \qquad \frac{}{\mathtt{null}.\mathtt{f}_i = v, S \rightarrow \mathtt{error}, S}$$

$$\frac{S[l] = \mathtt{N}(\overline{v}) \qquad \mathtt{N} <: \mathtt{P}}{(\mathtt{P})l, S \rightarrow l, S}\ \text{(R-CAST)} \qquad \frac{S[l] = \mathtt{N}(\overline{v}) \qquad \mathtt{N} \not<: \mathtt{P}}{(\mathtt{P})l, S \rightarrow \mathtt{error}, S}\ \text{(R-BAD-CAST)}$$

$$\frac{}{l\ >\ v, S \rightarrow v, S}\ \text{(R-CONTEXT)} \qquad \frac{}{\mathtt{let}\ \mathtt{x}\ =\ v\ \mathtt{in}\ \mathtt{e}_0, S \rightarrow [v/\mathtt{x}]\mathtt{e}_0, S}\ \text{(R-LET)}$$

Figure 5.15: FGO Reduction Rules

Reduction Context Expression:

$E ::=$

- $[\,]$
- $E.\mathtt{f}$
- $E.\mathtt{f} = \mathtt{e}$
- $l.\mathtt{f} = E$
- $E.\mathtt{m} < \overline{\mathtt{T}} >(\overline{\mathtt{e}})$
- $l.\mathtt{m} < \overline{\mathtt{T}} >(\overline{l}, E, \overline{\mathtt{e}'})$
- $(\mathtt{N})E$
- $l > E$
- $\mathtt{let}\ \mathtt{x} = E\ \mathtt{in}\ \mathtt{e}$

Context Reduction Rule:

$$\frac{\mathtt{e}, S \rightarrow \mathtt{e}', S'}{E[\mathtt{e}], S \rightarrow E[\mathtt{e}'], S'}$$

Figure 5.16: FGO Context Reduction Rule

for FGO programs. An evaluation context E is an expression with a hole ($[\,]$) so that the expression $E[\mathtt{e}]$ results from placing expression $\mathtt{e}$ into the hole of E. The rule uses the evaluation context E to define which subexpression of an FGO expression $\mathtt{e}$ should be

reduced first. Evaluation context E takes expression e as its argument and replaces it with another expression with its argument expression e placed in the appropriate position that needs to be reduced first. For example, $E[\texttt{e}_0]$ can be replaced with $\texttt{e}_0\texttt{.f} = \texttt{e}$ which means that the expression on the left hand side of the field assignment has to be reduced before the expression being assigned to the field.

In the method context reduction the arguments of the method call are evaluated left to right as shown by the location replacing the ones on the left of the current argument (denoted by E) being evaluated. The context reduction follows small step semantics since the notation in Figure 5.16 is no more than a shorthand for a large number of context reduction rules.

5.9 Type Soundness

In this section, I present the proofs of the *Type Preservation* Theorem and the *Progress* Theorem. Together they prove type soundness (as defined by Wright and Felleisen [WF94]) of the FGO type system. In contrast, FGJ+c as described in Chapter 4 does not need a full type soundness proof because FGJ+c is a subset of FGJ. FGJ+c relies on the proven type soundness of FGJ for a guarantee of a safe and reliable execution of any FGJ+c (and thus FGJ) program. The type soundness result proves the absence of ordinary type errors — it does not prove the ownership guarantees. The next section presents the proofs of the ownership guarantees provided by FGO.

The type preservation theorem (also known as the subject reduction theorem) proves that for every reduction possible for any FGO expression, the resulting expression is always going to be a subtype of the original expression and that the store well-formedness is preserved. The progress theorem shows that any FGO expression will always reduce to a value or produce an error. In other words, FGO programs will never get "stuck" not being able to apply any of the reduction rules.

Type soundness shows that the FGO language is a safe and predictable tool for writing programs. One cannot and need not conclude more than this from the type soundness. On the other hand, one can prove a variety of other properties and invariants that FGO preserves for all of its programs. I show such ownership invariants in the next section.

5.9.1 Type Preservation Theorem

The type preservation theorem says that if any FGO expression reduces to another FGO expression then the latter is always a subtype of the former. Before stating the theorem, let's define a shorthand for a well typed expression in a well typed store.

Definition 1. $\Delta; P \vdash \texttt{e}, S \ : \ \texttt{T} \equiv (\Delta; P \vdash \texttt{e} \ : \ \texttt{T}) \ \wedge \ (\Delta \vdash S)$

Theorem 2. ***(Type Preservation)*** *If* $\Delta; P \vdash \mathtt{e}, S \ : \ \mathtt{T}$ *and* $\mathtt{e}, S \rightarrow \mathtt{e}', S'$*, then* $\exists \Delta' \supseteq \Delta$ *and* $\exists \mathtt{T}' <: \mathtt{T}$ *such that* $\Delta'; P \vdash \mathtt{e}', S' \ : \ \mathtt{T}'$.

Proof. Using structural induction on the reduction rules in Figure 5.15, as follows.

R-NEW

$$\frac{l \notin dom(S) \qquad S' = S[l \mapsto \mathtt{N}(\overline{\mathtt{null}})] \qquad |\overline{\mathtt{null}}| = |fields(\mathtt{N})|}{\mathtt{new}\ \mathtt{N}(), S \rightarrow l, S'} \ \text{(R-NEW)}$$

Store: Define $\Delta' = \Delta \cup \{l \rightarrow \mathtt{N}\}$. I show that $\Delta' \vdash S'$ using the FGO-STORE rule.

To satisfy the first two lines of the FGO-STORE rule. By definition of Δ' and S' ($S' = S[l \mapsto \mathtt{N}(\overline{\mathtt{null}})]$), $dom(S') = dom(S) \cup \{l\} = dom(\Delta) \cup \{l\} = dom(\Delta')$ — where Δ is restricted to the mapping of locations to their types only. $S'[l] = \mathtt{N}(\overline{v})$ — where all values have been set to `null`. $\Delta'[l] = \mathtt{N}$ by definition of Δ'. By T-NEW $\Delta \vdash \mathtt{N}\ \mathtt{OK}$ and by definition $\Delta'(l) = \mathtt{N}$, and hence $\Delta \vdash \Delta'(l)\ \mathtt{OK}$ and by FGO-STORE-WF $\Delta \vdash \Delta'\ \mathtt{OK}$.

To satisfy the third line of the FGO-STORE rule. Consider any field i in $fields(\Delta'(l))$ $= fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$. All the newly added fields are set to `null` and, by T-NULL, `null` is a subtype of any type, including $\mathtt{T}_i$.

Finally, to satisfy the fourth line of the FGO-STORE rule. By FGO-STORE-WF, for any location l' in the new store S', $\Delta' \vdash \Delta'(l')\ \mathtt{OK}$ since if $l' \neq l$, then $l' \in \Delta$ and $\Delta \vdash S$ makes $\Delta(l')\ \mathtt{OK}$ and $\Delta'(l') = \Delta(l')$; and for $l' = l$, it is already established that $\Delta'(l)\ \mathtt{OK}$. Therefore $\Delta' \vdash S'$ as required.

Expression: By T-NEW one has $\mathtt{e} = \mathtt{new}\ \mathtt{N}() : \mathtt{T}$ where $\mathtt{T} = \mathtt{N}$. By T-LOC $\mathtt{e}' = l : \mathtt{T}'$ where $\mathtt{T}' = \Delta'(l)$. By definition of Δ' above, $\Delta'(l) = \mathtt{N}$. Therefore $\mathtt{T}' = \mathtt{N}$ and $\mathtt{T} = \mathtt{N} = \mathtt{T}'$. Hence by S-REFL $\Delta' \vdash \mathtt{T}' <: \mathtt{T}$ as required.

R-FIELD

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

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

Expression: By T-FIELD one has $\mathtt{e} = l.\mathtt{f}_i : \mathtt{T}$ where $\mathtt{T} = [this_P(l)/\mathtt{This}]\mathtt{T}_i$. By T-LOC, $\mathtt{e}' = v_i : \mathtt{T}'$ where $\mathtt{T}' = \Delta'(v_i)$ if $v_i \neq \mathtt{null}$ (since T-LOC only applies to *locations* and not to `null`). By FGO-STORE and since $\Delta' = \Delta$, one has $\Delta' \vdash \Delta'(v_i) <: [\mathtt{This}_l/\mathtt{This}]\mathtt{T}_i$ or if $v_i = \mathtt{null}$ then by T-NULL, $\Delta' \vdash \mathtt{null} <: \mathtt{T}_i$.

Furthermore, before v_i is stored in the store with R-FIELD-SET, T-FIELD-SET ensures that `This` is substituted by $this_P(l)$, hence $\Delta' \vdash \mathtt{T}' <: [this_P(l)/\mathtt{This}]\mathtt{T}_i$ and not just $\Delta' \vdash \mathtt{T}' <: \mathtt{T}_i$. Therefore $\Delta' \vdash \mathtt{T}' <: \mathtt{T}$ as required. Note that $this_P(l)$ will return $\mathtt{This}_l$ since at run-time the only value of permission P allowing the types to be well formed is l.

R-FIELD-SET

$$\frac{S[l] = \mathtt{N}(\overline{v}) \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad S' = S[l \mapsto [v/v_i]\mathtt{N}(\overline{v})]}{l.\mathtt{f}_i = v, S \rightarrow v, S'} \quad \text{(R-FIELD-SET)}$$

Store: Define $\Delta' = \Delta$. The only change to S is an update that ith field of object at location l is now pointing at v instead of v_i. Futhermore, v'type is guaranteed by T-FIELD-SET to be such that $\Delta \vdash \Delta(v) <: \Delta(v_i)$. Thus, the store well-formedness is preserved.

Expression: By T-FIELD-SET, have both $\mathtt{e} : \mathtt{T}$ and $\Delta(v) = \mathtt{T}$. Since $\mathtt{e}' = v$, $\Delta' \vdash \mathtt{T}' <: \mathtt{T}$ as required.

R-METHOD

$$\frac{S[l] = \mathtt{N}(\overline{v_l}) \qquad \mathit{mbody}(\mathtt{m} < \overline{\mathtt{V}} >, \mathtt{N}) = \overline{\mathtt{x}}.\mathtt{e}_0}{l.\mathtt{m} < \overline{\mathtt{V}} >(\overline{v}), S \rightarrow l\ >\ [\overline{v}/\overline{\mathtt{x}}, l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]\mathtt{e}_0, S} \quad \text{(R-METHOD)}$$

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

Expression [1]**:** By T-METHOD one has $\mathtt{e} = l.\mathtt{m} < \overline{\mathtt{V}} >(\overline{v}) : \mathtt{T}$ where $\mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}, \mathit{this}_P(l)/\mathtt{This}]\mathtt{U}$ and $\mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\Delta(l)) = < \overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}} > \overline{\mathtt{U}} \rightarrow \mathtt{U}$ and for some class $\mathtt{C}$ such that $\Delta \vdash \mathtt{N} <: \mathtt{C}$ method $\mathtt{m}$ is declared in it (by MT-CLASS). By method typing rule (T-METHOD), $\mathtt{e}_0 : \mathtt{U}$ and hence by T-CONTEXT one has $\Delta; l \vdash [\overline{v}/\overline{\mathtt{x}}, l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]\mathtt{e}_0 : [l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]\mathtt{U} = \mathtt{T}'$. Finally, since $P = l$, the FGO this function expands $\mathit{this}_P(l)$ into $[l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]$ substitution and $\Delta' \vdash \mathtt{T}' <: \mathtt{T}$ as required.

R-METHOD/FIELD/FIELD-SET-NULL

(R-METHOD/FIELD/FIELD-SET-NULL):

$$\frac{}{\mathtt{null}.\mathtt{m} < \overline{\mathtt{V}} >(\overline{v}), S \rightarrow \mathtt{error}, S} \qquad \frac{}{\mathtt{null}.\mathtt{f}_i, S \rightarrow \mathtt{error}, S} \qquad \frac{}{\mathtt{null}.\mathtt{f}_i = v, S \rightarrow \mathtt{error}, S}$$

NB! The same proof applies to all three of these rules.

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

Expression: By T-ERROR, `error` can be any of well formed FGO types and thus will be a well formed FGO subtype of $\mathtt{T}$ for any $\mathtt{T}$.

[1] The notation for $\overline{\mathtt{x}}.\mathtt{e}_0$ is taken from FGJ and lists all the arguments of the method followed by its body.

R-CAST

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

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

Expression: By T-CAST $\mathtt{e} = (\mathtt{P})l : \mathtt{P}$ and, hence, $\mathtt{P} = \mathtt{T}$. By T-LOC $\mathtt{e}' = l : \Delta(l)$ and by R-CAST $\Delta(l) = \mathtt{N} = \mathtt{T}'$. But by R-CAST one has $\Delta' \vdash \mathtt{T}' = \mathtt{N} <: \mathtt{P} = \mathtt{T}$ as required. Basically the reduction rule enforces the preservation of the type during casting, otherwise the R-BAD-CAST applies and raises an `error`.

R-BAD-CAST

$$\frac{S[l] = \mathtt{N}(\overline{v}) \qquad \mathtt{N} \nleq: \mathtt{P}}{(\mathtt{P})l, S \rightarrow \mathtt{error}, S} \text{ (R-BAD-CAST)}$$

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

Expression: By T-ERROR, $\Delta' \vdash \mathtt{T}' = \mathtt{error} <: \mathtt{P} = \mathtt{T}$ as required. Since `error` is made a subtype of all the well formed types for the type preservation proof to be simple.

R-CONTEXT

$$\frac{}{l > v, S \rightarrow v, S} \text{ (R-CONTEXT)}$$

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

Expression: Given that $\mathtt{e} = l > v \ : \ \mathtt{T}$ if and only if $\Delta; l \vdash v : \mathtt{T}$ by T-CONTEXT, one has $\mathtt{e}' = v \ : \ \mathtt{T}$ as required.

R-LET

$$\frac{}{\mathtt{let\ x} \ = \ v \ \mathtt{in}\ \mathtt{e}_0, S \rightarrow [v/\mathtt{x}]\mathtt{e}_0, S} \text{ (R-LET)}$$

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

Expression: By T-LET one has $\mathtt{e} = \mathtt{let\ x} \ = \ v \ \mathtt{in}\ \mathtt{e}_0 \ : \ \mathtt{T}$ where $\mathtt{x} \ : \ \Delta(l), \Delta; P \vdash \mathtt{e}_0 \ : \ \mathtt{T}$. Since in $\mathtt{e}' = [v/\mathtt{x}]\mathtt{e}_0$, one has $\mathtt{x} \ : \ \Delta(v)$ by T-LET, one also has $\mathtt{e}' \ : \ \mathtt{T}$ as required.

□

5.9.2 Progress Theorem

The progress theorem shows that FGO programs do not get "stuck" and that any well typed FGO expression that does not contain free variables (closed) can be reduced to some value or FGO's `error` (the latter includes failed downcasts due to R-BAD-CAST reducing them to `error`).

Theorem 3. ***(Progress)*** *Suppose* `e` *is a closed well-typed FGO expression. Then either* `e` *is a value (or* `error`*) or there is an applicable reduction rule that contains* `e` *on the left hand side.*

Proof. I will go through every possible type that the expression `e` can be and show that progress is satisfied. According to the rules in Figure 5.9, `e` can be a field, field set, method call, cast, let, context, new, error, null, location, a variable, or a reducable expression (redex).

If `e` is an `error`, `null` or location l, then there is nothing to prove. Since `e` is a closed expression it cannot be a variable. If `e` is a redex, then it can be reduced using the context reduction rule.

We are now only left with the following cases: field, field set, method call, cast, let, context, and new. Now, I will show that for these cases one of the reduction rules applies.

There are no additional requirements for the R-LET, and R-CONTEXT rules and thus they are applicable if `e` matches their left hand side.

In the case of an expression `e` being a cast, one of R-CAST or R-BAD-CAST has to apply based on whether the two types involved are subtypes or not. In either case, the reduction result is a value.

Hence, we are now left with field, field set, method call, and new. We need to make sure that the corresponding reduction rule having `e` on its left hand side has all of the additional conditions met.

In the case of R-FIELD and R-FIELD-SET well-typedness of `N` ensures that $\mathit{fields}(\texttt{N})$ is well defined and $\texttt{f}_i$ appears in it.

In the case of R-METHOD, the fact that 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.

In the case of $l = \texttt{null}$, one of R-FIELD-NULL, R-METHOD-NULL, and R-FIELD-SET-NULL will ensure that the expression reduces to `error`.

Finally, in the case of a `new N()` expression, it always reduces to a value by R-NEW.

□

5.10 Ownership Guarantees

I start by presenting a lemma stating that in FGO, all the types preserve their ownership information. I use this lemma to prove the confinement invariant that shows that FGO guarantees confinement in a way equivalent to other systems like CFGJ [ZPV06]. I state two definitions of what it means for an object to *refer* to another object and what it means for an object to be *inside* another object. I utilise these definitions to state and prove both confinement and ownership invariants. The ownership invariant shows that FGO provides ownership in a way comparable to SafeJava [Boy04] and Ownership Types [Cla02].

Finally, in Section 5.10.4, I present a different version of an ownership invariant — called a *shallow* ownership invariant [CW03] which was discussed in Section 2.4 — that is not as strong as deep ownership invariant but provides for more flexibility [AKC02].

The major difference between these Generic Ownership proofs and earlier non-type-generic ownership invariant proofs lies in a much simpler formulation. The key benefit comes from integrating ownership into a parametric polymorphic type system, rather than building an ownership-parametric type system on top of a non-generic typed language.

Lemma 2. ***(Ownership Invariance)*** *If* $\Delta \vdash \texttt{S} <: \texttt{T}$ *and* $\Delta \vdash \texttt{T} <: \texttt{Object}\texttt{<}\texttt{O}\texttt{>}$*, 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. □

5.10.1 Refers To and Inside Definitions

First, let's define what it means for one object to refer to another object in FGO:

Definition 2. ***(Refers To)*** *An object at location* l *refers to an object at location* l' *if and only if (1, fields) for some* Δ *one has* $\Delta(l) = \texttt{N}(\bar{l})$ *and* $l' \in \bar{l}$*; or (2, locals) for some* $\Delta,\ P$ *one has* $\Delta; P \vdash l > \texttt{e} : \texttt{T}$ *and* l' *occurs as one of the subexpressions of* e.

Second, let's define an *inside* ($\prec$) relationship on owner classes for objects (e.g., $\texttt{This}_l$) in the same manner as the previous ownership type systems [CPN98, BLS03]. During the execution of any FGO program with deep ownership, if an object l refers to object l', then $\texttt{This}_l \prec owner(\Delta(l'))$. Or more formally:

Definition 3. ***(Inside)*** *Owner class* `T` *is inside (*$\prec$*) owner class* `T'` *denoted* $\texttt{T} \prec \texttt{T}'$ *if and only if* $\Delta \vdash \texttt{T} <: \texttt{T}' <: \texttt{World}$.

At class declaration validation time, the $\prec$ relationship for owner classes is as follows: $\texttt{This} \prec \texttt{Owner} \prec \texttt{World}$ (note that the FGO owner classes' subtyping relationship is along the same lines: $\texttt{This}_l <: \texttt{This} <: \texttt{World}$). During reduction, both `Owner` and

`This` will have appropriate location-specific owners (e.g., $\texttt{This}_l$) substituted for them. This allows us to prove a deep ownership invariant similar to that of Clarke [Cla02] (and as used by Boyapati [Boy04]).

5.10.2 Confinement Invariant

This section presents a confinement invariant that is equivalent to the confinement invariant supported by FGC as described in Chapter 4. The difference is that now one is in an imperative language setting, rather than a simpler language like FGJ.

Theorem 4. ***(Confinement Invariant)*** *If l refers to l' and $owner_\Delta\ (l') = \pi_{\texttt{C}}$, then $visible_\Delta\ (\pi_{\texttt{C}}, \Delta(l))$.*

Proof. According to the FGO reduction rules in Figure 5.15, the only ways that a new reference is created in the store is via one of R-NEW or R-FIELD-SET expressions. In the context of confinement, when these occur during the execution of methods inside a class `C`, they should only refer to $visible$ classes in the current package ($\pi_{\texttt{C}}$).

Hence, I will consider every possible subexpression appearing in the body of a method of a well formed FGO class `C` during program execution. I show two things:

1. If $\texttt{e} \rightarrow^* \texttt{new D}\texttt{<}\overline{\texttt{T}_\texttt{D}}\texttt{>}()$, then $visible_\Delta(\texttt{D}\texttt{<}\overline{\texttt{T}_\texttt{D}}\texttt{>},\ \texttt{C})$.

2. If $\texttt{e} \rightarrow^* \texttt{e}_\texttt{o}.\texttt{f}_\texttt{i} = \texttt{e}_1)$ and for some $P\ \Delta; P \vdash \texttt{e}_1 : \texttt{T}_1$, then $visible_\Delta(\texttt{T}_1,\ \texttt{C})$.

Because the class is a well formed FGO class, its methods are well formed FGO methods. This, plus the standard subformula property [2], implies that, for appropriate $\Delta; P$: both $\Delta; P \vdash \texttt{e} : \texttt{T}$ and $\Delta; \texttt{C} \vdash visible(\texttt{e})$ hold. From this one can derive $visible_\Delta(\texttt{T},\ \texttt{C})$, and hence $visible_\Delta(owner_\Delta(\texttt{T}),\ \texttt{C})$.

The first part is then proved as follows:

By FGO's type preservation property, there is a $\texttt{T}'$ such that $\Delta; P \vdash \texttt{new D}\texttt{<}\overline{\texttt{T}_\texttt{D}}\texttt{>}() : \texttt{T}'$, where $\Delta \vdash \texttt{T}' \texttt{<:}\ \texttt{T}$. Furthermore, one has that $\Delta; P \vdash \texttt{new D}\texttt{<}\overline{\texttt{T}_\texttt{D}}\texttt{>}() : \texttt{D}\texttt{<}\overline{\texttt{T}_\texttt{D}}\texttt{>}$, and hence clearly $\Delta \vdash \texttt{D}\texttt{<}\overline{\texttt{T}_\texttt{D}}\texttt{>} \texttt{<:}\ \texttt{T}'$, and $\Delta \vdash \texttt{D}\texttt{<}\overline{\texttt{T}_\texttt{D}}\texttt{>} \texttt{<:}\ \texttt{T}$.

By Lemma 2, $owner_\Delta(\texttt{D}\texttt{<}\overline{\texttt{T}_\texttt{D}}\texttt{>}) = owner_\Delta(\texttt{T})$, from which one deduces $visible_\Delta(owner_\Delta(\texttt{D}\texttt{<}\overline{\texttt{T}_\texttt{D}}\texttt{>}),\ \texttt{C})$, and therefore $visible_\Delta(\texttt{D}\texttt{<}\overline{\texttt{T}_\texttt{D}}\texttt{>},\ \texttt{C})$.

The second part is then proved as follows:

By FGO's type preservation property, there is a $\texttt{T}'$ such that $\Delta; P \vdash \texttt{e}_\texttt{o}.\texttt{f}_\texttt{i} = \texttt{e}_1 : \texttt{T}'$, where $\Delta \vdash \texttt{T}' \texttt{<:}\ \texttt{T}$. Furthermore, one has that $\Delta; P \vdash \texttt{e}_\texttt{o}.\texttt{f}_\texttt{i} = \texttt{e}_1 : \texttt{T}_1$, and hence clearly $\Delta \vdash \texttt{T}_1 \texttt{<:}\ \texttt{T}'$, and $\Delta \vdash \texttt{T}_1 \texttt{<:}\ \texttt{T}$.

By Lemma 2, $owner_\Delta(\texttt{T}_1) = owner_\Delta(\texttt{T})$, from which one deduces $visible_\Delta(owner_\Delta(\texttt{T}_1),\ \texttt{C})$, and therefore $visible_\Delta(\texttt{T}_1,\ \texttt{C})$. □

[2] The subformula property basically means that all the subexpressions of an expression have to be well formed since the expression typing rules recurse into every possible subexpression and check its well-formedness.

5.10.3 Ownership Invariant

This section presents a deep ownership invariant equivalent to the other established ownership type systems [Cla02, Boy04].

Theorem 5. ***(Ownership Invariant)*** *l refers to l' only if* $\texttt{This}_l \prec owner_\Delta(l')$ *or* $owner_\Delta(l') = \pi_\texttt{C}$ *and* $visible_\Delta(\pi_\texttt{C}, \Delta(l))$.

Proof. For fields by FGO-STORE, $\Delta \vdash \Delta(l') <: [\texttt{This}/\texttt{This}_l]\ \texttt{T}_\texttt{i}$. If owner is `World` or `This`, then the theorem holds by the definition of $\prec$. If owner is anything else then since well-formedness preserves owner class nesting and `This` <: `Owner` <: `O` (where `O` is the set of owners of type parameters) holds, one has $\Delta \vdash \texttt{This}_l <: \texttt{This}_{l'}$. The second part of the proof holds due to the confinement invariant. □

5.10.4 Shallow Ownership Invariant

This section assumes that the highlighted bits of WF-TYPE, T-METHOD, FGO-CLASS-MANIFEST and FGO-CLASS-PURE rules are omitted from the type system. Refer to Section 2.4 for the discussion of deep vs shallow ownership.

Theorem 6. ***(Shallow Ownership Invariant)*** *l refers to l' only if l'* $\in$ *owners(*$\Delta(l)$*) or* $visible_\Delta\ (\Delta(l'), \Delta(l))$.

Proof. By FGO type preservation, $\Delta' \vdash \Delta'(v) <: \Delta(l)$ and $\Delta \subseteq \Delta'$. By ownership invariance, $owner_\Delta(\Delta(l)) = owner_\Delta(\Delta'(l)) = owner_\Delta(\Delta'(v))$. By V-OWNER, $visible_\Delta(\Delta'(v), \Delta(l))$. □

5.11 Summary

In this chapter I presented the FGO type system together with type soundness and ownership invariance proofs. FGO is the first type system to provide support for ownership, confinement, and type polymorphism at the same time. The FGO type system is reasonably compact and easy to formulate. The only restrictions imposed by the FGO type system to provide generic ownership are owner preservation over subtyping, owner nesting, a function to handle `This` owners, and placeholder owner initialisation.

Chapter 6

Ownership Generic Java

Contents

Chapters 4 and 5 provide sound type systems that support Generic Ownership. This chapter is designed to finish making the case for Ownership Generic Java (OGJ) as a feasible extension to Java 5 by presenting a prototype of the language.

OGJ is the first publicly available language that supports confinement, (deep or shallow) ownership, and type genericity at the same time. OGJ is implemented as an extension to the Sun Microsystem's Java 5 compiler (`javac`). The implementation of the OGJ language requires minimal changes to Java 5, which advances my thesis that Generic Ownership is a practical way of getting ownership support into a modern object-oriented programming language like Java.

Chapter 3 presented the OGJ language. In Section 6.1, I review the challenges presented by a fully featured language implementation like Java 5 as opposed to a simpler formalism like Featherweight Generic Java or Featherweight Generic Ownership. Section 6.2 surveys a case study of re-writing Java Collections using OGJ and the possibilities for utilising

ownership features in the implementation of the standard collection library. I describe the details of the OGJ implementation in Section 6.3 and summarise in Section 6.4.

6.1 From FGO to OGJ

FGO (just like FGJ) is an idealised formal model of the Java programming language that is easy to reason about. In this section, I highlight how OGJ deals with generic arrays, inner classes, static fields, static methods, exceptions, interfaces and other issues — none of which are supported by the formalisation in the previous chapters. I also discuss the significance of having `java.lang.Object` as the root of the class hierarchy in both OGJ and FGO. I assume that the reflection facilities of Java have been disabled (as in the Java Enterprise Edition) for the purposes of OGJ. Leaving reflection enabled allows programmers to circumvent the compiler checks by arbitrarily changing the owners at run-time. For the remainder of this section I consider the example in Figure 6.1 of a valid Java 5 program that contains each of the issues highlighted here.

```
1  import ogj.*;
2  import java.util.*;
3
4  public class OutsideClass<Owner extends World>
5      implements SomeInterface1, SomeInterface2 {
6    public static LinkedList<String> listOfNames;
7
8    class InsideClass<OwnerInner extends World> {
9      public static void <OwnerMethod extends World>
10             staticMethod(OutsideClass<OwnerMethod> oc) {
11       // Create a new instance of OutsideClass.
12       OutsideClass<OwnerMethod> oc =
13         new OutsideClass<OwnerMethod>();
14     }
15   }
16
17   public InsideClass<Owner>[] genericArray =
18     new InsideClass[42];
19
20   private Data<This> secretData = new Data<This>();
21
22   public void exceptionalMethod() throws Exception() {
23     throw new DataException(``A simple blank exception.'',
24                             this.secretData);
25   }
26 }
```

Figure 6.1: OGJ Language Issues Example

6.1.1 Generic Arrays

Java 5 support for generic arrays [BOSW98] is slightly different from the treatment of generic types. In particular, array stores and accesses are checked by Java at run-time, at which time the generic information has been erased from the types involved. This means that making an array of `List<String>` will produce a run-time array of `List` with the knowledge about the type of the elements lost after compile time. The array is still required to only store instances of `List<String>`, hence the burden on the language implementors is to detect any illegal stores and accesses to the `List<String>` array at compile time to avoid unnecessary run time errors.

Since OGJ uses generic type information to store ownership, arrays have to be treated carefully to avoid storing references to objects not owned by the array. I created a special Java class called `OGJArray<Element, EOwner, Owner>` for encapsulating a typical Java array and behaving as a normal Java class with an owner. Note that the second parameter (`EOwner`) can be trivially eliminated using *placeholderowners* functionality, but it makes the owners more explicit for the sake of examples. I then replaced by hand all the uses of the arrays in Java programs with the uses of `OGJArray` allowing OGJ to catch any illegal accesses and retrievals using the getter and setter methods with appropriate owner parameters. The OGJ programmer is required to use the `OGJArray` class instead of standard Java arrays to gain the benefits of ownership.

In the example in Figure 6.1, field `genericArray` has to be created with `new InsideClass[42]` rather than `new InsideClass<Owner>[42]` even though the latter contains more information about the elements stored in the array. The latter will cause a Java "creation of generic array" error since variance is required to handle array stores and accesses. The "creation of generic array" Java error [Mic04] exists because the required type information for the type variables is not available at runtime. More comprehensive generic array support in Java is required for more granular treatment of generic arrays with owners. As for now, careful refactoring of the Java code to stop using plain Java arrays or a possible compile-time or bytecode rewriting that automates the process of converting arrays to `OGJArray` is a sufficient solution.

If one were to imagine class `String` as having an owner — a similar owner wrapper would have to be created. This suggests a general ownership wrapper design pattern that can be used to hide unconventional classes behind a conventional ownership-safe facade. Fortunately, immutable objects like strings are safe from the aliasing defects and do not require owners for safe programming.

6.1.2 Inner Classes

Inner classes are classes defined inside another class that get access to the private fields and the instance of the enclosing (outer) class. OGJ supports innner classes but restricts the

inner classes to have no special access rights compared to the outer classes. The meaning of `This` owner is then specific to the inner class and not the outer class. Accessing the private fields of the outer class parameterised by `This` is appropriately prohibited by OGJ. Each instance of an inner class is treated as though it was yet another Java class with no special access relationship with its outer class.

In the example in Figure 6.1 inner class `InsideClass` has a separate owner `Owner-Inner` (as opposed to the owner of the outer class: `Owner`) and any usage of owner `This` within `InsideClass` will be incompatible with the use of owner `This` in the outer class.

When designing OGJ, I had a choice of whether to allow the access to `Owner` inside the `InsideClass` or not. The simplest approach is to prohibit such access, making inner classes no different from any other class, but OGJ allows such access since it does not break the encapsulation of the outer class (as long as the owner nesting is preserved) — while accessing `This` of the outer class would potentially break such encapsulation.

How inner classes are treated is important when implementing `Iterators` in the Java Collections Library since every data structure provides its own implementation of the `Iterator` in a private inner class that is returned when calling an `iterator()` method. Such private inner classes often access the internal implementation of data structures which makes it impossible to hide such private implementation details with owner `This` while allowing iterators direct access to them.

An alternative approach is proposed by Clarke [Cla02] and widely utilised by Boyapati [BLS03] where inner classes get less restricted access to the internals of the outer class, allowing constructs such as iterators in collections to be easily implemented without breaking encapsulation. Each iterator is declared as an inner class of the collection class it iterates over. Since an iterator is an inner class, it is allowed to access the internal details of the collection implementation. An instance of the iterator class can be returned to the outside with the same owner as the collection, thus making it as encapsulated as the collection itself.

OGJ can be easily extended to accommodate for such variation, but in the meantime I consider the solution of equal treatment of outer and inner classes to be adequate and easier to understand.

6.1.3 Static Fields and Methods

Static fields and static methods in Java 5 are the same across all the instances of a particular class. When writing these there is no concept of a current instance (`this`) or instances of class type parameters. OGJ supports static fields and static methods, but one cannot refer to the enclosing class's type parameters or use an instance specific owner `This`. Instead, method type parameters can be used to supply additional owner parameters required by

the method's implementation.

Consider method `staticMethod` in Figure 6.1. Since it accepts an argument `OutsideClass<OwnerMethod>` that requires an owner, a generic method type parameter `OwnerMethod` is used to supply the owner. The static method has access to neither the owner `Owner` of the class it is declared in, nor the private owner `This`, nor any other owner parameters of the class. Since `staticMethod` now has an owner parameter, it is possible to create objects owned by `OwnerMethod` inside this method. If OGJ did not support method owner parameters, the only objects that could be created inside static methods would have nonvariable owners `World`, `Package`, or `Class`. Having owners for static methods makes it possible to create objects inside these methods with any owner, so long as they are supplied via the method's owner parameters.

When it comes to static fields, no type variables, including owner variables, can be used. This means that certain operations cannot be fully converted from Java 5 to FGO without using manifest ownership. For example, the Singleton design pattern [GHJV94] can require an assignment to the static field as follows:

```
class Singleton {
  public static Singleton instance;
  public Singleton() {
    Singleton.instance = this;
  }
}
```

If one is to annotate `Singleton` with owner parameter `Owner extends World`, then the type of `this` will be `Singleton<Owner>`. Unfortunately, the latter uses a type variable `Owner` from the class `Singleton` — which it is not possible to refer to in a static field, since it is instance-specific. Finding an acceptable solution for interaction of static fields (e.g., storing a reference to `this`) with class instances annotated with owners is a subject for future development of OGJ.

6.1.4 Exceptions

Every time an exception is thrown, it can potentially contain a pointer to the private details of the object that caused it. As it propagates outwards, it may expose objects owned by the object that caused the exception. One may ask if all the exceptions should be made public, and thus world readable, or change ownership as they propagate outwards through a system's ownership domains. Ownership in the presence of exceptions has been addressed by Dietl and Müller [DM04].

Figure 6.1 shows a method `exceptionalMethod` that throws an `Exception`. An exception is created using the `MyException` class that needs to have a pointer to a private field such as `this.data`. Throwing (or alternatively, catching) such an exception

is impossible since it contains a pointer to a field owned by `This` and cannot be accessed outside the current instance, making it problematic to implement some of the common exception handling mechanisms. One cannot allow the ownership invariant to be broken when the exception occurs since it will defeat the purpose of safe exception handling by the program itself.

OGJ does not currently support exception handling correctly — any exception that occurs during the execution of an OGJ program can potentially break encapsulation boundaries. The reason for the lack of support is Java's restriction on any class extending `Throwable` that prohibits generic type parameters. One of the many subjects for future work is to figure out the most appropriate way of dealing with Java exceptions in a generic setting.

6.1.5 The Root of the Class Hierarchy

When formalising OGJ in FGO, the root of the class hierarchy is taken to be a parameterised class `Object<Owner extends World>`, and Java's root `Object` is defined as a manifest class `Object extends Object<World>` with owner `World`. To avoid name conflict, these two classes named `Object` can be placed in different packages or the parameterised `Object` can be named `OObject` (for *owned* object) instead. When it comes to OGJ, the Java language and libraries make a lot of assumptions about the root being `Object` and none other. To work around this issue, OGJ leaves `Object` as a public class (which has manifest owner `World`) that is a root of the class hierarchy and creates class `OObject<Owner extends World> extends Object`, which is the only owner parameterised class allowed to change the ownership of its superclass `Object`. All the pure classes in OGJ then extend from class `OObject` as they do in FGO. It is part of my future work to resolve this issue.

6.1.6 Interfaces and Other Issues

FGO does not support interfaces. Figure 6.1 shows `OutsideClass` implementing two interfaces. What happens if interfaces have owners and one has to take many owners into account when declaring a new class? What if such owners are in conflict with each other due to different owner bounds? OGJ avoids resolving this issue by allowing owner parameters for interfaces and requiring the same preservation of owners for all the super interfaces as it does for the super class.

Other issues include implementing `equals` comparison for two objects that are of the same class but have different owners. Equality in the presence of ownership is discussed by Gordon [Gor07], but is restricted in OGJ because it requires compatible owners.

The implementation of *placeholderowners* functionality is left out from the OGJ

prototype, requiring manual addition of missing owners. Rather than implementing the *placeholderowners* function as described in FGO type system in Chapter 5, one may chose to utilise Java 5 wildcards to provide optional owner placeholders as in:

```
class Map<Key<?>, Value<?>, Owner> { ... }
```

In the future, I plan to investigate the wildcards approach futher.

Finally, an issue of downcasting has been addressed by ownership researchers [BLR03, WC07] and is not resolved in OGJ, providing a fertile ground for future developments with the eventual goal of bringing ownership into Java.

6.2 Case Study: Java Collections

As a case study of OGJ programming, I ported most of `collections.jar` from Java 5 to OGJ by adding appropriate owner parameters and making use of them whenever possible. The porting includes the `Vector` and `HashMap` collections classes which I discuss in the remainder of this section. Figure 6.2 shows an extract from the `HashMap` implementation in OGJ.

On one hand, the `Vector` class required encapsulating the `elementData` array inside it in `OGJArray` with an owner `This` making it private to each particular instance of `Vector`. On the other hand, the `HashMap` implementation of the `Map` interface requires the `table` of map entries to have the same ownership as the instance of the map itself. Due to iterators it was impossible to make map entries in `table` field private to each particular instance of `HashMap` as was possible in the case of the entries in the `Vector` class.

Figure 6.2 shows the code from the `HashMap` class (with the assumption of a placeholder owners function implementation that allows omission of some implicit owners for `K`, `V` etc.). Field `table` is owned by `Owner` instead of `This`. In line 24, the iterator stores a direct reference to the private field of `HashMap`. The reason for the failure to fully encapsulate the `table` field in the `HashMap` but still managing to do so in `Vector` is because the iterator for `Vector` uses the public interface methods such as `get` and `put` to access the entries inside the `Vector` without breaking the encapsulation by having a direct pointer to the field storing the entries inside a `Vector`. The implementation of the `HashMap` iterator requires the iterator to have a direct pointer bypassing the public access mechanism and thus breaking the encapsulation of the `Map`. This problem can be addressed by allowing inner classes (like `Iterator`) a greater access to the private fields of each instance or by allowing local variables temporary access that might break ownership of particular fields. It is partially possible to resolve this issue by rewriting the code to use appropriate accessors like is done in the `Vector` class.

To perform the porting, I had to add `Owner extends ogj.World` to all the

```
1  public class HashMap<K, V, Owner extends World>
2    extends AbstractMap<K, V, Owner>
3    implements Map<K, V, Owner>, Cloneable, Serializable
4  {
5  ...
6    // Note that owner of table is not This (which is desirable)
7    transient OGJArray<Entry<K, V, Owner>, Owner, Owner> table;
8  ...
9    private abstract class HashIterator<E, HIOwner extends World>
10           implements Iterator<E, HIOwner> {
11   Entry<K, V, Owner> next;      // next entry
12   Entry<K, V, Owner> current;   // current
13 ...
14   Entry<K, V, Owner> nextEntry() {
15       if (modCount != expectedModCount)
16           throw new ConcurrentModificationException();
17       Entry<K, V, Owner> e = next;
18       if (e == null)
19           throw new NoSuchElementException();
20       Entry<K, V, Owner> n = e.next;
21
22       // The following line will fail if the table is owned by
23       // This instead of Owner.
24       OGJArray<Entry<K, V, Owner>, Owner, Owner> t = table;
25
26       int i = index;
27       while (n == null && i > 0)
28           n = t.get(--i);
29       index = i;
30       next = n;
31       return current = e;
32   }
33 ...
34 }
```

Figure 6.2: `HashMap` in OGJ

classes involved and to add any additional owners that the *placeholderowners* function in FGO would add automatically. At the time of writing the OGJ prototype implementation does not fully support *placeholderowners* functionality.

Every use of the classes in the static methods required additional owner parameters to these methods, similarly, an owner instance (e.g., `ogj.World`) had to be added to the type occurring in the static fields. Every use of an array had to be replaced with `OGJArray` and `System.arraycopy` had to be modified to account for owner parameters. The collections implementation also utilises the `foreach` statement which again assumes the super type of `java.lang.Object`, thus this statement had to be replaced with a standard `for` loop.

6.3 OGJ Language Implementation

The OGJ language is implemented [Pot05] by extending the source of the Java 5 language implementation [Sun05]. The Java language implementation consists of parsing the class source files and constructing an Abstract Syntax Tree (AST) for each of them, attributing the AST, and generating class files. The implementation then provides a visitor for AST that has several visitors already implemented that perform attribution (type checking) on the AST. OGJ adds three additional visitors to perform additional checks of expressions before, during, and after the attribution phase. The only direct change to the actual Java language implementation not contained in the external visitors is the OGJ adjustments to the subtyping check that require preservation of ownership and prohibit raw types.

The owner classes used in the language to denote ownership are: `World`, `Package`, `Class`, and `This`. To make OGJ fully compatible with Java, these classes are provided as blank interfaces in the `ogj` package. This means that when an OGJ program is compiled using Java 5, all of these interfaces are treated as blank interfaces and have no effect on the program. However, the OGJ implementation treats these four interfaces in the package `ogj` specially enabling ownership support. Therefore the OGJ syntax is fully indentical to the Java 5 syntax.

The Java 5 compiler implements *erasure* [IPW01a] — a mechanism that removes the use of all the generic type parameters at compile time after performing type checking. This means that compiled classes will not use owner parameters in the bytecode after the compile time checks are performed. Publicly accessible classes and their instances marked with owner class `World` required no extra work on OGJ's behalf.

To ensure that classes utilising an owner class `Package` are not accessible outside the package that the owner class is used in, each occurrence of the `ogj.Package` is replaced with a uniquely generated class name of a package private class. Such package private classes are generated by OGJ before type checking begins and removed by OGJ at the end

of the type checking process so that the resulting compiled classes do not include them. These classes are called pseudoclasses and their only purpose is to stop two classes owned by `ogj.Package` but in different packages from having compatible types. Furthermore, since the pseudoclass replacement is private to its package, it is impossible to use the name of the pseudoclass outside its defining package without Java signalling an error, the same as any package private class in Java.

The only way to circumvent the mechanism of private package class replacement is by casting away the owner class. This is prevented in OGJ by adding extra checks in the source of the Java language implementation that prevents any class that has an owner parameter in the last place from being cast to either `java.lang.Object`, a raw type, or a class with a different owner parameter.

OGJ also provides an ability to confine an instance to a particular class similar to Class Universes [MPH99]. This is done with owner `Class` and in a manner very similar to owner `Package`, only the unique pseudoclass generated corresponds to the class inside which the owner is used. Preventing casting away of owners and using unique pseudoclasses is sufficient to implement confinement to static domains.

For example, a class defined as follows:

```
package my.util;

import ogj.*;

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

guarantees to have all of its instances confined within the `my.util` package as long as the code is compiled using OGJ.

The support for object ownership is implemented using owner class `This`. One can re-use the mechanism utilised for the `Class` owner to ensure that anything owned by `This` is confined to a particular class it is used in. To ensure that anything owned by `This` is also confined to a particular instance OGJ implements the *this* rule presented in Chapter 5. The *this* rule requires that any access to instances owned by `This` is only performed using a variable `this` that refers to the current instance and no other access is permitted. Together with owner nesting checks done in the way described in Chapter 5 this makes OGJ provide deep ownership support.

Rather than extending the Java 5 language implementation, it is also possible to express OGJ in a constraint system such as JavaCOP [ANMM06]. JavaCOP provides a rule language that can describe the extensions to a language required to be supported by the implementation. Appendix B provides a description of an alternative implementation of the OGJ language that does not require any direct modifications of the Java language implementation. The fact that OGJ was implemented in two separate ways supports my

thesis that OGJ is a practical and easy to implement extension to a modern language.

OGJ is one of many language extensions implemented using the JavaCOP rule language. The JavaCOP implementation of OGJ concentrates on the deep ownership. Unfortunately, the confinement aspect of OGJ requires substantial changes to the program, so only the ownership aspect of OGJ has been emulated in JavaCOP, leaving confinement for future work. I refer interested readers to Appendix B for the JavaCOP rule set implementing OGJ's deep ownership support.

6.4 Summary

In this chapter, I overviewed the OGJ language implementation, which is available for public download and use. I also present a case study of using OGJ to implement Java collections. Together with the formal background, this chapter concludes this thesis's presentation of the argument that Generic Ownership is a practical way of introducing ownership types into a modern programming language like Java or C#.

Chapter 7

Conclusion

Contents

In this chapter, I conclude my thesis. I outline my contributions, discuss the Generic Ownership approach with respect to the related work, and survey my plans for future work.

7.1 Contributions

In this thesis I have described and formalised Generic Ownership — a single system that encompasses both generic types, deep, per-object ownership, and confinement. The aim was to provide a practical way of integrating ownership into modern object-oriented languages, such as Java or C#, that already employ genericity. The aim was not to provide a novel mechanism combining ownership and genericity. My goal was also different to other recent ownership types work [PB05, Boy04, LP06b, AC04, KA05, LM04, Mit06] which investigates new kinds of topologies, restrictions or applications for ownership types. Generic Ownership reuses generic language constructs as much as possible to provide ownership. The language design and formalism show that ownership and generic type information can be expressed within a single system, and carried around the program as bindings to the same parameters. As a result, programs using Generic Ownership are only

slightly more complex than those using just generic types, yet enjoy the full protection provided by ownership types.

I have demonstrated the practical side of Generic Ownership by designing the Ownership Generic Java language, a seamless, syntactically compatible extension to generic-capable Java 5.

The contributions of my thesis are as follows:

- **Generic Ownership** — a new way of combining ownership and genericity by using a single parameter space to carry both type and ownership information.
- **Featherweight Generic Confinement (FGC)** — a formal model showing how Generic Ownership provides confinement guarantees with only a few simple restrictions to an established Java formalism.
- **Featherweight Generic Ownership (FGO)** — a formal model demonstrating how ownership, confinement, and generic types can be combined together in one type system using the Generic Ownership approach.
- **Ownership Generic Java (OGJ)** — a language design that supports Generic Ownership — it is the first language design that supports ownership, confinement, and generics at the same time.

7.2 Discussion

The work described in Chapter 2 showed great potential for the applications of ownership types. The possible applications include concurrency [BR01], persistence [Boy04], security [BV99], and software architecture [AKC02]. The need for ownership in modern programming languages motivated the development of the Generic Ownership ideas into an Ownership Generic Java (OGJ) language. The OGJ language syntax is fully compatible with Java and allows a programmer to denote particular objects as `World`: accessible by anyone, `Package`: accessible by instances of classes declared in the package where an object is created, `Class`: accessible by instances of the class inside which the object is created, or `This`: only accessible by the object that created the current object. Programs written in OGJ allow the programmers to reap a wide range of benefits of ownership types.

Figure 7.1 shows how OGJ fits among the schemes described in Chapter 2 (FLAP stands for Flexible Alias Protection and OT stands for the Ownership Type). The developments in the early 90's of full alias encapsulation, such as Islands and Balloons, together with the Geneva Convention on Aliasing, led to Flexible Alias Protection formalised as Ownership Types and Universes in the late 90's. The aliasing research outcomes were applied to memory optimisation using Sandwich Types and software security using Confined Types.

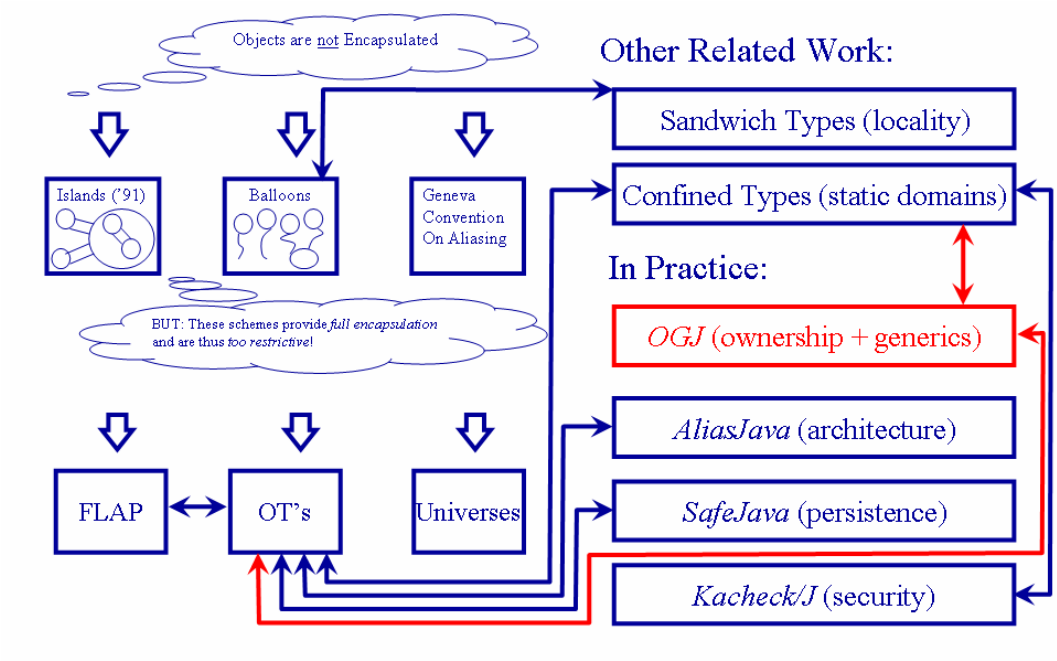

Figure 7.1: OGJ and Other Alias Management Schemes

Practical aspects of ownership were demonstrated by the development of Kacheck/J, AliasJava, and SafeJava. The OGJ language fits nicely among the practical ownership applications and brings together the confinement and ownership work.

7.2.1 Object Ownership

Object ownership can address aliasing, security, concurrency, and memory management problems, while smoothly aligning with typical object-oriented program designs. Systems using object ownership range from expressive but weighty *explicit* systems based on ownership types [CPN98] to lightweight but limited *implicit* systems based on confined types [VB01].

Ownership types make ownership information an integral part of the type. The ownership part cannot be separated or changed through subtyping or casts. Generic Java (GJ) [BOSW98] also makes the generic part of types an integral part of the type, making it an obvious choice for carrying ownership type information in Java. An alternative merger of confinement with genericity [ZPV06] that treats confinement and genericity orthogonally results in more complex proofs and type rules duplicating the job that can be shared between genericity and ownership. Simpler proofs in the Generic Ownership approach are additional evidence that merging ownership and genericity is more than a syntactical change to the language.

Explicit systems such as AliasJava, Universes, and the systems of Boyapati et al. [AC04, AKC02, BLS03, MPH99] differ primarily in one characteristic that Clarke and Wrigstad

define as *shallow* vs. *deep* ownership [CW03]. Deep ownership protects transitively nested objects, while shallow ownership does not. Generic Ownership is also an explicit deep ownership system, however, the key contribution is that Generic Ownership combines type genericity and object ownership into a single system.

Clarke and Drossopoulou [CD02] and Boyapati et al. [BLS03] describe how to exploit the strong protection provided by deep ownership. The strong, transitive protection that deep ownership provides is also a liability, because the deep ownership protection prevents programmers from accessing objects' internal structures, and can require inefficient coding idioms to move data across the objects' interfaces. Aldrich and Chambers' AliasJava [AC04] and Boyapati et al.'s SafeJava [BLS03] show how ownership types can support more flexible object graph topologies.

AliasJava adopted Generic Ownership style parameters in ownership domains [AC04] to provide type and ownership genericity. Their work was inspired by OGJ [AC04] — this thesis establishes that their adoption is sound. Krishnaswami and Aldrich [KA05] have recently formalised an extension to ownership domains using System F and permission-based ownership, illustrating a range of flexible ownership topologies, and supporting strong encapsulation via ML-style abstract types and genericity.

7.2.2 Ownership Applications

Other recent research has extended the applicability of ownership-style schemes. Lu and Potter [LP05] have shown how to ensure object references are acyclic, or to control field updates even when references are unconstrained [LP06b]: they also employ ownership wildcards. Boyland and Retert [Boy03, BR05] have described how various forms of uniqueness and ownership can be encoded using fractional permissions. None of these proposals are type-generic: however, one expects that, like AliasJava, languages designed with these systems could take advantage of generic ownership to provide both type and owner polymorphism.

Unlike more recent ownership type schemes [BLR03, WC07] FGO does not currently support runtime downcasts, thus the rule forbidding casts that would lose ownership information. This is primarily for compatibility with existing Java and GJ programs, because safe ownership downcasts require runtime information which existing language implementations do not supply nor existing libraries expect. Also, it is possible that other Ownership Type systems require many of these downcasts because they are not type-generic: as with GJ, FGO's genericity should remove the need for many of these downcasts.

All these explicit systems require additional annotations to use them. For this reason, Aldrich et al. [AKC02] and Boyapati et al. [BR01, BLR02, BLR03] have described a range of type inference schemes to provide these annotations automatically. Generic Ownership

enables a simpler approach, as many of the owner parameters that have to be inferred by other schemes are already present in type-generic code.

7.2.3 Ownership-Related Systems

Implicit confined type systems have achieved their more limited goals while keeping the number of annotations low. Vitek and Bokowski's original system [VB01] required classes to be annotated as confined, while Clarke, Richmond and Noble [CRN03] apply these ideas in the context of Enterprise Java Beans. More recent work by Zhao, Palsberg and Vitek [ZPV06] has formalised confined types based on Featherweight Java. Confined Featherweight Java also includes a notion of generic confined types. For example, generic confined types allow a collection to be confined or not depending upon the specifications of the contained elements. The Generic Ownership 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, I start from a language with generic types (Generic Java, and its formal core FGJ), and then ensure ownership and confinement directly. The Generic Ownership approach leads to a simpler formal system requiring fewer new concepts and distinctly simpler and shorter proofs.

Banerjee and Naumann [BN04a, BN02] prove a per-object representation independence result for Java. 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 removed from a type. Ultimately, their results say that confinement and ownership matter for deep reasoning about objects.

7.2.4 Ownership and Generics

Clarke's thesis [Cla02] was the first account of a system with both parametric polymorphism and ownership. This system was based on Abadi and Cardelli's object calculus [AC96], rather than a class-based language. Clarke, however, gives an encoding of a class-based language into his formalism. He further discusses how ownership can be combined with a class-based language (with inner classes), but does not provide a generic type system or language design.

A bit further afield, one finds that the implementation of the State Monad in Haskell adopts mechanisms similar to FGO [LP95]. In the State Monad, a type variable is assigned to the encapsulated state, and an appropriate hiding of the type (via rank-2 polymorphism) ensures that the state does not 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 [Cla02].

Even more generally, types are a recognised tool for managing and reasoning about aliasing. Types are both useful for characterising when aliasing may be present, when it is definitely present, when a particular piece of code does not introduce aliases, and so forth. Type-based alias analysis uses the class hierarchy to discover non-alias conditions [DMM98], though the space of types considered is restricted to ordinary program types.

7.2.5 Other Work

Ownership types are similar to region types and region polymorphism [TJ92, TT97], but serve quite different purposes: ownership is used for encapsulation while regions are used to manage memory allocation. Regions are restricted to stack-based memory allocation, while ownership supports long-lived objects and much less restrictive topologies. Just as region polymorphism permits functions to be applied to arguments in different regions, so generic ownership polymorphism allows a class to be instantiated to handle arguments with different ownership. The actual ownership type arguments act as permissions allowing the generic class access the arguments with those types.

John Potter has suggested that owners could be modelled orthogonally to class types [1]: in such a scheme, types would be a pair comprising an owner context and a class type (e.g., `T ::= C@O`, where `@` binds a type and an owner context). Cyclone, X10, and F_{own} [GMJ+02, CDE+05, KA05] types have similar structures. This approach has the advantage of keeping owners and types conceptually separate, and, if type variables can range over such pairs, providing much of the polymorphism of Generic Ownership. Compared with the Generic Ownership model, these pairs need a little more language support — a separate kind of owners, and then constructors and accessors to retrieve owners from type pairs. This may make it harder to type `this` and may be harder to incorporate into a programming language than Generic Ownership.

Regions and other systems use Hindley-Milner style type inference to make fine distinctions on potential aliasing [Bak90, OJ97, LS85]. More generally, functional programmers use a related technique called "phantom types" [FP02] to include a wide range of information within types. Generic Ownership, albeit without type inference, is similar to these approaches in that it, too, uses additional type parameters to carry ownership information, while paying a minimal syntactic cost. Work on phantom types [Hin03, FP02, LM99], where "phantom" type parameter's only purpose is to enforce well-formedness constraints, directly aligns with the Generic Ownership approach.

Recent work in Enhancing Encapsulation in OOP [The99] and in JAC [KT01] explored ways to make ownership useful in practice. C++ is working towards the idea of concepts [Str06, GJS+06] that might allow Generic Ownership style parameters in the future

[1] Personal communication, 2005.

versions of C++. Boyland [Boy05] argues that ownership is preferable to readonly in Java.

Finally, FGO's deep ownership type formation constraints are very similar to GADT type constraints [KR05], although FGO requires subtyping rather than type equality. Based on the model of Generic Ownership, one expects that a language with GADT's and subtype constraints may be strong enough to express ownership directly in its type system.

7.3 Future Work

The primary goal of my future work is to be able to design an ownership inference system. A formal inference model together with an implementation as an Eclipse plugin would allow porting of Java 5 programs to OGJ programs in the most efficient and correct way. Programmer-assisted ownership inference is the next big goal in making ownership part of day to day programming.

In Chapter 5, I presented manual proofs of the FGO type soundness. It is possible to create a machine-assisted proof of type soundness. The machine-assisted proofs catch any omissions that can happen when hand writing a tedious proof that goes through all the possible cases. It is part of the future work on Generic Ownership to convert these proofs to a machine-checkable format.

I plan to utilise the formal foundations provided by FGO to work out proof principles of ownership guarantees and to formulate a combined ownership and confinement invariant.

In Chapter 6, I explored a number of issues that arise when one combines genericity and ownership which provide a fertile ground for future investigation. Addressing these issues will enable one to explore the deep semantic relationship between the two concepts. The issues included the treatment of arrays, inner classes, static fields and methods, exceptions, the root of the class hierarchy, and interfaces.

Exceptions and ownership are addressed by Dietl and Müller [DM04]. When it comes to OGJ, generic classes in Java 5 cannot extend `Throwable` and thus simply adding owner parameters to exception classes is not going to work. Investigating exceptions in the presence of Generic Ownership forms an interesting subject of future work.

When implementing OGJ programs one needs to pay attention to interfaces. Since interfaces — just like classes — can have owner parameters, one can imagine multiple interface inheritance with multiple owners. While OGJ allows interfaces to have owners and restricts the owners of the super interfaces to be the same (just like the owner of the super class), there is not yet a formal guarantee that this solution is sound.

Annotations can be used to replace declarations of private fields owned by `This` with a single annotation `@private`. For example declaring the `secretData` field with owner `This`, as in:

```
private Data<This> secretData;
```

rather than `private Data`, conceptually means that we want a "really private" field that does not have unexpected aliases. Syntactic sugar (for example Java Annotations) can be used to allow fields to be declared as `@private Data` to carry the meaning of object-encapsulation, rather than only name-based encapsulation.

Last but not least, I would like to develop a set of OGJ design patterns for programmers wishing to make use of ownership in their programs.

Appendix A

Featherweight Generic Java Type System

In this appendix I briefly outline the complete FGJ type system [IPW01a].

Theorem 7. ***FGJ Type Soundness***. *If* $\emptyset;\emptyset \vdash \texttt{e} : \texttt{T}$ *and* $\texttt{e} \rightarrow^* \texttt{e}'$ *with* $\texttt{e}'$ *a normal form, then* $\texttt{e}'$ *is either (1) an FGJ value* v *with* $\emptyset;\emptyset \vdash \texttt{v} : \texttt{S}$ *and* $\emptyset \vdash \texttt{S} <: \texttt{T}$ *or (2) an expression containing* $(\texttt{P})\texttt{new N}(\overline{\texttt{e}})$ *where* $\emptyset \vdash \texttt{N} <: \texttt{P}$.

```
T ::= X  |  N
N ::= C<T̄>
L ::= class C<X̄◁N̄>◁N {T̄ f̄; K M̄}
K ::= C(T̄ f̄) { super(f̄); this.f̄=f̄; }
M ::= <X̄◁N̄> T m(T̄ x̄) { return e; }
e ::= x | e.f | e.m<T̄>(ē) | new N(ē) | (N)e

X ranges over the type variables.
N ranges over the nonvariable types.
```

CT class table: a mapping from class names C to class declarations L.
Δ type environment: a mapping from type variables to nonvariable types.
Γ type environment: a mapping from variables to types.

Figure A.1: FGJ Syntax

Subclassing:

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

Field lookup:

$$\mathit{fields}(\texttt{Object}) = \bullet \qquad \text{(F-Object)}$$

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

Method type lookup:

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

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

Method body lookup:

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

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

Figure A.2: FGJ Auxiliary Functions

Bound of type:

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

Subtyping:

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \qquad \text{(S-Refl)}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}} \qquad \text{(S-Trans)}$$

$$\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X}) \qquad \text{(S-Var)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\ \{...\}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}} \qquad \text{(S-Class)}$$

Well-formed types:

$$\Delta \vdash \mathtt{Object\ ok} \qquad \text{(WF-Object)}$$

$$\frac{\mathtt{X} \in dom(\Delta)}{\Delta \vdash \mathtt{X\ ok}} \qquad \text{(WF-Var)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\ \{...\}} \quad \Delta \vdash \overline{\mathtt{T}}\ \mathtt{ok} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>\ ok}} \qquad \text{(WF-Class)}$$

Valid downcast:

$$\frac{dcast(\mathtt{C}, \mathtt{D}) \qquad dcast(\mathtt{D}, \mathtt{E})}{dcast(\mathtt{C}, \mathtt{E})} \qquad \frac{\mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{D<}\overline{\mathtt{T}}\mathtt{>\ \{...\}} \quad \overline{\mathtt{X}} = FV(\overline{\mathtt{T}})}{dcast(\mathtt{C}, \mathtt{D})}$$

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

Valid method overriding:

$$\frac{mtype(\mathtt{m}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{Z}} \lhd \overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U_0} \text{ implies } \overline{\mathtt{P}}, \overline{\mathtt{T}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}](\overline{\mathtt{Q}}, \overline{\mathtt{U}}) \text{ and } \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \vdash \mathtt{T_0} <: [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}]\mathtt{U_0}}{override(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \rightarrow \mathtt{T_0})}$$

Figure A.3: FGJ Subtyping and Type Well-formedness Rules

Expression typing:

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

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{T}_i} \qquad \text{(GT-Field)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}_0)) = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U} \\ \Delta \vdash \overline{\mathtt{V}}\ \mathrm{ok} \qquad \Delta \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}\mathtt{<}\overline{\mathtt{V}}\mathtt{>}(\overline{\mathtt{e}}) : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}} \qquad \text{(GT-Invk)}$$

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

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) <: \mathtt{N}}{\Delta;\Gamma \vdash \mathtt{(N)e}_0 : \mathtt{N}} \qquad \text{(GT-UCast)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathtt{N}\ \mathrm{ok} \qquad \Delta \vdash \mathtt{N} <: \mathit{bound}_\Delta(\mathtt{T}_0) \\ \mathtt{N} = \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>} \qquad \mathit{bound}_\Delta(\mathtt{T}_0) = \mathtt{D}\mathtt{<}\overline{\mathtt{U}}\mathtt{>} \qquad \mathit{dcast}(\mathtt{C}, \mathtt{D})\end{array}}{\Delta;\Gamma \vdash \mathtt{(N)e}_0 : \mathtt{N}} \qquad \text{(GT-DCast)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathtt{N}\ \mathrm{ok} \qquad \mathtt{N} = \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>} \qquad \mathit{bound}_\Delta(\mathtt{T}_0) = \mathtt{D}\mathtt{<}\overline{\mathtt{U}}\mathtt{>} \\ \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \qquad \mathit{stupid\ warning}\end{array}}{\Delta;\Gamma \vdash \mathtt{(N)e}_0 : \mathtt{N}} \qquad \text{(GT-SCast)}$$

Method typing:

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \qquad \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}, \overline{\mathtt{P}}\ \mathrm{ok} \\ \Delta;\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e}_0 : \mathtt{S} \qquad \Delta \vdash \mathtt{S} <: \mathtt{T} \\ \mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\ \{...\}} \qquad \mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T})\end{array}}{\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>\ T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e}_0\mathtt{;\ \}\ OK\ IN\ C}\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}} \qquad \text{(GT-Method)}$$

Class typing:

$$\frac{\begin{array}{c}\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}} \vdash \overline{\mathtt{N}}, \mathtt{N}, \overline{\mathtt{T}}\ \mathrm{ok} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C}\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \\ \mathtt{K = C(}\overline{\mathtt{U}}\ \overline{\mathtt{g}}\mathtt{,}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{super(}\overline{\mathtt{g}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{ = }\overline{\mathtt{f}}\mathtt{;\}}\end{array}}{\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{M}}\mathtt{\}\ OK}} \qquad \text{(GT-Class)}$$

Figure A.4: FGJ Typing Rules

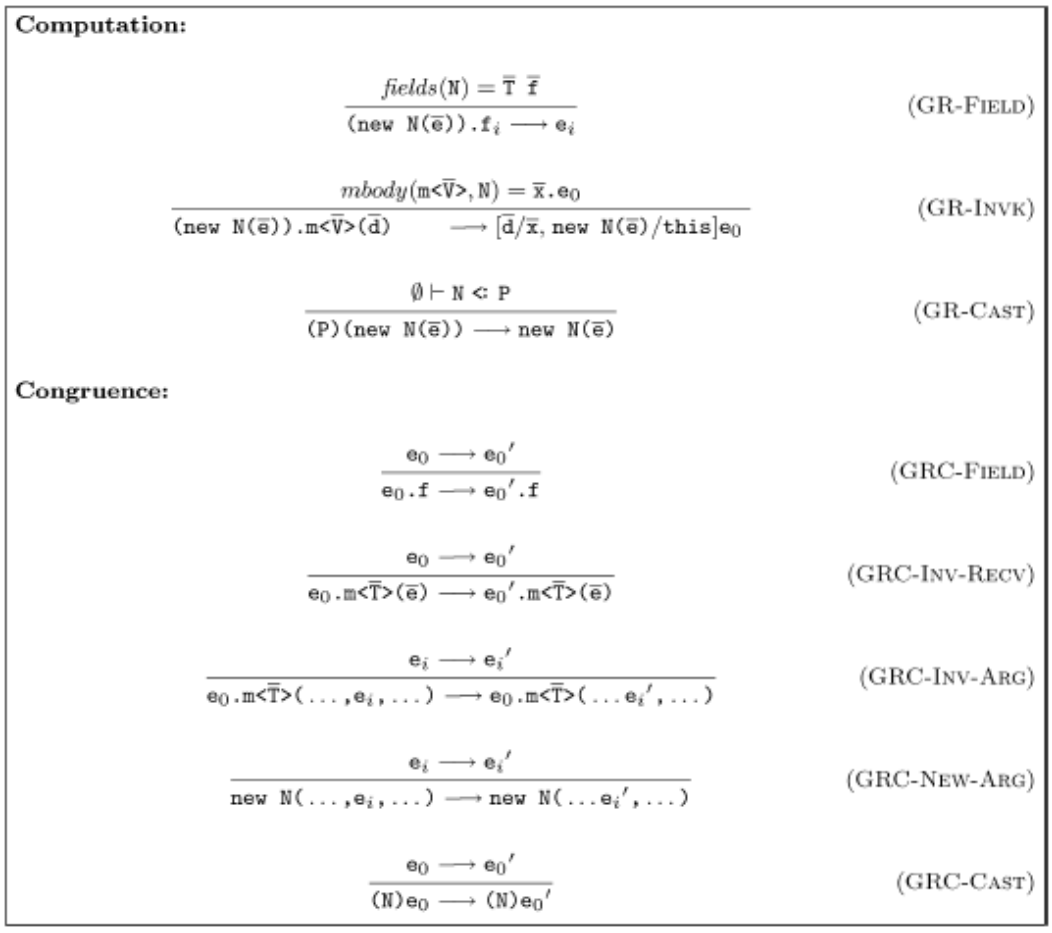

Figure A.5: FGJ Reduction Rules

Appendix B

OGJ Deep Ownership implementation using JavaCOP Rules

In this appendix I present the JavaCOP [ANMM06] rules implementing OGJ. The following was *taken verbatim* from the paper by Chris Andreae et al [ANMM06].

The full implementation is around 770 lines in 43 rules and predicates; in this section we present an overview of some of the key rules.

B.1 Classes require owner parameters

In OGJ, most programmer-defined classes are required to take an extra generic parameter to record their owner. This "owner parameter" must be declared to extend the class `World` (`ogj.World` in our implementation), and must be the last parameter provided, as in:

```
class List<Element, Owner extends World> { ...
```

In JAVACOP, we can enforce this behaviour as follows:

```
rule RequireOwnerParameter(ClassDef c){
 where(c.type.supertype == globals.objectType){
  require(hasOwnerParameter(c.type) ||
          isOwnerType(c.type)):
   error(c, "Toplevel class ("+c.type+")
             needs owner parameter");
 }
 where(hasOwnerParameter(c.type.supertype)){
  require(c.type.typarams.last ==
           c.type.supertype.typarams.last):
   error(c,"Ownership not preserved when "+
          c.type+" extends "+c.type.supertype);
 }
```

```
 where(hasOwnerParameter(c.type)){
  require(c.type.supertype == globals.objectType
         || hasOwnerParameter(c.type.supertype)):
   error(c,"Owned type extends non-owned type");
 }
}
```

This rule carries out a simple case analysis in its `where` clauses. Classes extending `java.lang.Object` must declare a new owner parameter, while classes that extend another class must monotonically preserve their superclass's owner parameter. Finally, we require that classes may not be declared with an owner parameter unless they extend either `Object` or a superclass with an owner parameter.

B.2 Preservation of ownership

The class formation rule above ensures that OGJ classes will always have an owner parameter. We then need to ensure that this information cannot be lost from an object's static type — that is, we have to prevent OGJ classes being cast to Java's raw types or to `java.lang.Object`. For example, consider the following generic list of books: its elements are public `Book` objects owned by `World`, while the `List` itself is private, owned by `This`. Note that `World` and `This` are constants instantiating the owner parameters of `List` and `Book` [Pot05].

```
List<Book<World>,This> l =
    new List<Book<World>, This>;

Object obj = l; // loses ownership information
List rawList = l; // so do raw types
```

In JAVACOP, this rule can be implemented as follows:

```
rule preventRawCasting(sub <: supr @ e){
 where(hasOwnerParameter(sub)){
  require(!supr.isRaw):
   error(e, "Cast "+sub+" to raw type "+supr);
  require(supr != globals.objectType):
   error(e,"Cast "+sub+" to Object");
}}
```

This rule, like similar rules for Confined Types and other related systems, uses JAVACOP's subtyping rule form [ANMM06] to check all explicit and implicit casts. If the cast is from a type with an owner parameter, we forbid that type from being cast to a raw type or `Object`.

B.3 Deep ownership

These two rules are sufficient to ensure that every OGJ object has an owner that is known statically. OGJ however has a more structured ownership model, known as "deep ownership", that establishes a transient owners-as-dominators property on the heap [Pot05]. This means that private data (e.g., data owned by `This`) must not be stored in publicly-accessible structures. For example, code such as:

```
List<Book<World>,This> l = new ...;
```

that declares a private list of public `Book` objects is quite permissible, but the alternative:

```
List<Book<This>,World> l = new ...;
```

must be prevented. In JAVACOP, this rule requires some quite complex constraints on generic class instantiation — especially because owners can be represented by type parameters such as (but not limited to) the `Owner` parameter of every OGJ class.

Following the OGJ formal system [Pot05], we consider each class definition, and (using JAVACOP's `forall` quantifier over a tree) recursively check that all types present within the class are well formed for deep ownership.

```
rule wellFormedTypes(ClassDef c){
 forall(Tree t : c){
  where(t.type instanceof ClassType
     && !t instanceof ClassDef){
   require(wellformed(t.type)):
    error(t,"Type "+t.type+" of "+t+"
     not well formed for deep ownership");
}}}
```

The bulk of the work then is done by the `wellformed` predicate. First, types that do not take parameters (mostly primitive types, `void`, `null`, and packages) are considered well formed, as are type variables.

```
declare wellformed(Type t){
 require(unparameterizable(t));
}

declare wellformed(TypeVar v){
 require(true);
}
```

Second, for types that do not have an owner parameter (presumably Java class types that are being used in OGJ programs), we check that all their generic type parameters (`typarams`) are well formed.

```
declare wellformed(ClassType c){
 require(!hasOwnerParameter(c));
 forall(Type prmtype: c.typarams){
  require(wellformed(prmtype));
}}
```

Finally we check that each generic type instantiation is well formed:

```
declare wellformed(ClassType c){
 require(Type ownerofC;
     ownerofC <- c.typarams.last){
  forall(Type prmtype: c.typarams){
    where(isOwnerType(prmtype)){
     require(
       isDeepOwnerSubtype(ownerofC, prmtype));
    }
    where(Type ownerofPrmType;
      ownerofPrmType <-
               prmtype.classBound.typarams.last){
      where(isOwnerType(ownerofPrmType)){
       require(isDeepOwnerSubtype(ownerofC,
                                  ownerofPrmType));
   }}}}
 forall(Type prmtype: c.typarams){
   require(wellformed(prmtype));
}}
```

This rule fetches the class's owner (`ownerofC`) from the instantiation of the last (ownership) generic type parameter, and then iterates over each of the type parameters. If the parameter is a "naked" owner parameter (i.e., it extends `World`) then we check that the parameter is outside the class's owner (via the auxiliary `isDeepOwnerSubtype` predicate); alternatively if the parameter is an OGJ class, we fetch its owner (`ownerofPrmType`) in turn from *its* last parameter, and check that that owner is valid. The `isDeepOwnerSubtype` predicate directly encodes the OGJ ownership subtyping relationship [Pot05]. To finish, we again recurse through all the type parameters.

B.4 Instance encapsulation

These three rules work together to provide and statically enforce a dynamic model where objects are deeply nested inside each other. The final rule uses this structure to enforce strong object encapsulation: private data may only be accessed from within their owners.

For example, the following code declares an encapsulated list within the `Catalogue` class:

```
class Catalogue<Owner extends World>{
 List<Book<World>,This> myList;

 int volumes() {return myList.size();}
}
```

This list can be accessed freely inside the current instance of the class but cannot be used outside it, because types owned by `This` may only be accessed (explicitly or implicitly) via `this`.

```
Catalogue cat = ...;

...cat.myList.size();
 // error as myList is private.
```

In JAVACOP, properly expressing this rule requires handling a number of cases. We begin with a rule that catches Java's "`.`" dot selection operator via JAVACOP's `Select` AST node, and then simply delegates to the JAVACOP `validThisSelect` predicate:

```
rule OGJThisSelect(Select s){
 require(validThisSelect(s)):
  warning(s,"Invalid select on "+s.type);
}
```

The `validThisSelect` predicate clauses catch various forms of dot operator. First, we permit any calls via `this` and any uses of the dot operator to construct compound types:

```
declare validThisSelect(Select s){
 require(s => [this.*]
         || s.sym instanceof TypeSymbol);
}
```

Next, we check each case of the Select node — these clauses catch method sends and variable accesses — and require that their generic type arguments do not include `This` via the `thislessType` predicate.

```
declare validThisSelect(Select s){
 require(s.type instanceof MethodType){
  require(thislessType(s.sym.type));
}

declare validThisSelect(Select s){
 require(s.sym instanceof VarSymbol){
```

```
  require(thislessType(s.sym.type));
}
```

Finally, we have a series of predicates that recursively define a "thisless" type as being a type which does not involve the owner `This` in any of its generic type parameters.

```
declare thislessType(Type t){
 require(unparameterizable(t));
}

declare thislessType(ClassType c){
 require(!(c.fullName.equals("ogj.This")));
 forall(Type t: c.allparams){
  require(thislessType(t));
}}
```

The full OGJ ruleset straightforwardly extends these cases to cover Java's other types and selection constructs.

Bibliography

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, Berlin, Heidelberg, Germany, 1996.

[AC04] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 3086, pages 1–25, Oslo, Norway, June 2004. Springer-Verlag, Berlin, Heidelberg, Germany.

[AFM97] Ole Agesen, Stephen Freund, and John Mitchell. Adding type parameterization to Java. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1997.

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 311–330, Seattle, WA, USA, November 2002. ACM Press, New York, NY, USA.

[Alm97] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, Berlin, Heidelberg, Germany, June 1997.

[Alm98] Paulo Sérgio Almeida. *Control of Object Sharing in Programming Languages*. PhD thesis, Department of Computing, Imperial College of Science, Technology, and Medicine, University of London, June 1998.

[ANC+06] Chris Andreae, James Noble, Yvonne Coady, Celina Gibbs, Jan Vitek, and Tian Zhao. Stars: Scoped types and aspects for real-time systems. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, Berlin, Heidelberg, Germany, 2006.

[ANMM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Proceedings of*

ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Portland, Oregon, USA, October 2006. ACM Press, New York, NY, USA.

[Bak90] Henry G. Baker. Unify and Conquer (Garbage, Updating, Aliasing) in Functional Languages. In *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, pages 218–226, Nice, France, June 1990.

[BDF+03] Mike Barnett, Robert DeLine, Manuel Fahndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming (FTfJP)*, Darmstadt, Germany, July 2003. Springer-Verlag, Berlin, Heidelberg, Germany.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[BLR03] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Safe runtime downcasts with ownership types. In Dave Clarke, editor, *Proceedings of International Workshop on Aliasing, Confinement, and Ownership (IWACO)*, pages 1 – 14. Utrecht University, July 2003.

[BLS03] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, pages 213–223, New Orleans, LA, USA, January 2003. ACM Press, New York, NY, USA. Invited talk by Barbara Liskov.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Shulte. The Spec# programming system: An overview. In *CASSIS*, 2004.

[BN02] Anindya Banerjee and David A. Naumann. Representation Independence, Confinement and Access Control. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, pages 166–177. ACM Press, New York, NY, USA, 2002.

[BN04a] Anindya Banerjee and David A. Naumann. Ownership Confinement Ensures Representation Independence for Object-Oriented Programs. *Journal of the ACM (JACM)*, 52(6):894–960, November 2004.

[BN04b] Mike Barnett and David Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science (LNCS), pages 54–84. Springer-Verlag, Berlin, Heidelberg, Germany, July 2004.

[BNR01] John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, Berlin, Heidelberg, Germany, June 2001.

[Boo91] Grady Booch. *Object-oriented design with applications*. The Benjamin / Cummings Publishing Company, 1991.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java programming language. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[Boy03] John Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, number 2694 in Lecture Notes in Computer Science (LNCS), pages 55–72. Springer-Verlag, Berlin, Heidelberg, Germany, 2003.

[Boy04] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, EECS, MIT, February 2004.

[Boy05] John Boyland. Why we should not add readonly to Java (yet). In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming (FTfJP)*, Glasgow, Scotland, July 2005.

[BR01] Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 56–69, Tampa Bay, FL, USA, 2001. ACM Press, New York, NY, USA.

[BR05] John Tang Boyland and William Reter. Connecting effects and uniqueness with adoption. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, 2005.

[BV99] Boris Bokowski and Jan Vitek. Confined types. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 1999.

[CD02] 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)*, pages 292–310, Seattle, WA, USA, November 2002. ACM Press, New York, NY, USA.

[CDE+05] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vivek Sarkar, and Christoph Von Praun. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[Cla02] Dave Clarke. *Object Ownership and Containment*. PhD thesis, School of CSE, UNSW, Australia, 2002.

[CPN98] 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)*, pages 48–64, Vancouver, Canada, October 1998. ACM Press, New York, NY, USA.

[CRN03] 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)*, pages 374–387, Anaheim, CA, 2003. ACM Press, New York, NY, USA.

[CW03] David Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In *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.

[CWP+03] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *Foundations of Intrusion Tolerant Systems*, pages 227–237. IEEE, 2003.

[DD85] J. Donahue and A. Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, 1985.

[DDM07] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 2007.

[DM04] Werner Dietl and Peter Müller. Exceptions in ownership type systems. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming (FTfJP)*, 2004.

[DM05] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005. `http://www.jot.fm/issues/issue_2005_10/article1`.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1998.

[FP02] Matthew Fluet and Riccardo Pucella. Phantom Types and Subtyping. In *International Conference on Theoretical Computer Science (TCS)*, pages 448–460, August 2002.

[GHJV94] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[GJS+06] Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: First-class language support for generic programming in C++. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 291–310, Portland, Oregon, USA, October 2006. ACM Press, New York, NY, USA.

[GMJ+02] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, 2002.

[Gor07] Donald Gordon. Encapsulation enforcement with dynamic ownership. Master's thesis, School of Mathematics, Statistics, and Computer Science. Victoria University of Wellington, 2007.

[GPV01] 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)*, pages 241–255, Tampa Bay, FL, USA, 2001. ACM Press, New York, NY, USA.

[Gro97] Secure Internet Programming Group. Hotjava 1.0 signature bug. `http://www.cs.princeton.edu/sip/news/april29.html`, 1997.

[GTZ98] Daniela Genius, Martin Trapp, and Wolf Zimmermann. An approach to improve locality using sandwich types. In *Proceedings of the 2nd Types in Compilation Workshop*, Kyoto, Japan, 1998.

[Hin03] Ralf Hinze. *The Fun of Programming*, chapter Fun with Phantom Types, pages 245–262. Palgrave Macmillan, 2003. Editors: Jeremy Gibbons and Oege de Moor.

[HLW+92] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The Geneva convention of the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, April 1992.

[Hog91] 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.

[HW91] D. E. Harms and B. Weide. Copying and swapping: influences on the design of reusable components. *IEEE Transactions of Software Engineering*, 17(5):424–435, 1991. IEEE CS Press.

[IPW01a] 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.

[IPW01b] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. A recipe for raw types. In *Proceedings of Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2001.

[IV02] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 441–469, Malaga, Spain, June 2002. Springer-Verlag, Berlin, Heidelberg, Germany. To appear in ACM Transactions on Programming Languages and Systems.

[KA05] Neel Krishnaswami and Jonathan Aldrich. Permission-based ownership: Encapsulating state in higher-order typed languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 96–106, Chicago, UL, USA, 2005. ACM Press, New York, NY, USA.

[KR05] Andrew Kennedy and Claudio Russo. Generalized algebraic data types and object-oriented programming. In *Proceedings of ACM Conference on Object-*

Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2005.

[KS01] Andrew Kennedy and Don Syme. The design and implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[KT01] Gunter Kniesel and Dirk Theisen. JAC - access right based encapsulation for Java. *Software: Practice and Experience*, 31(6), 2001.

[Lan92] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4), December 1992.

[LM99] Daan Leijen and Erik Meijer. Domain-Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 109–122, Berkeley, CA, USA, October 1999. USENIX Association.

[LM04] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, Berlin, Heidelberg, Germany, 2004.

[LP95] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[LP05] Yi Lu and John Potter. A type system for reachability and acyclicity. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 479–503. Springer, 2005.

[LP06a] Yi Lu and John Potter. Ownership and accessibility. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[LP06b] Yi Lu and John Potter. Protecting representation with effect encapsulation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, 2006.

[LS85] Leslie Lamport and Fred B. Schneider. Constraints: A uniform approach to aliasing and typing. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, pages 205–216, New Orleans, Louisiana, 1985.

[MBL97] Andrew C Myers, Joseph A. Bank, and Barbara Liskov. Parameterized Types for Java. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[Mic04] Sun Microsystems. Bug id 5105887. `http://bugs.sun.com/bugdatabase/view_bug.do;:WuuT?bug_id=5105887`, September 2004.

[Mil78] Robin Milner. Theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Mit06] Nick Mitchell. The runtime structure of object ownership. In Dave Thomas, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science (LNCS)*, pages 74–98, Nantes, France, July 2006. Springer-Verlag, Berlin, Heidelberg, Germany.

[MPH99] P. Müller and A. Poetzsh-Heffter. *Programming Languages and Fundamentals of Programming*, chapter Universes: a Type System for Controlling Representation Exposure. Fernuniversität Hagen, 1999. Poetzsh-Heffter, A. and Meyer, J. (editors).

[Näg06] Stefan Nägeli. Ownership in design patterns. Master's thesis, Software Component Technology Group, Department of Computer Science, ETH Zurich, 2006.

[NBT+03] James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, and Dave Clarke. Towards a Model of Encapsulation. In Dave Clarke, editor, *Proceedings of International Workshop on Aliasing, Confinement, and Ownership (IWACO)*, number 030 in UU-CS-2003. Utrecht University, July 2003.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In Eric Jul, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science (LNCS)*, pages 158–185. Springer-Verlag, Berlin, Heidelberg, Germany, July 1998.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: a program understanding tool based on type inference. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Boston, USA, May 1997.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, January 1997.

[PB05] Pratibha Permandla and Chandrasekhar Boyapati. A type system for preventing data races and deadlocks in the Java Virtual Machine language. Technical report, University of Michigan, 2005.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PNB04] Alex Potanin, James Noble, and Robert Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, 2004.

[PNC98] John Potter, James Noble, and David Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, Adelaide, Australia, November 1998. IEEE Press.

[PNCB04] 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.

[PNCB06a] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight Generic Confinement. *Journal of Functional Programming*, 16(6):793–811, September 2006.

[PNCB06b] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[PNZV05] Alex Potanin, James Noble, Tian Zhao, and Jan Vitek. A high integrity profile for memory safe programming in real-time Java. In *The 3rd workshop on Java Technologies for Real-time and Embedded Systems*, San Diego, CA, USA, 2005.

[Pot05] Alex Potanin. Ownership Generic Java Download. `http://www.mcs.vuw.ac.nz/~alex/ogj/`, 2005.

[Pug07] Bill Pugh. Find Bugs — A Bug Pattern Detector for Java. `http://www.cs.umd.edu/~pugh/java/bugs/`, 2007.

[Str06] Bjarne Stroustrup. A brief look at C++0x. `http://www.artima.com/cppsource/cpp0xP.html`, January 2006. Online article on Artima Developer.

[Sun05] Sun Microsystems. Java Development Kit. Available at: `http://java.sun.com/j2se/`, 2005.

[TEH+04] Mads Torgerson, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming

language. *Journal of Object Technology*, 3(11):97–116, December 2004. Special Issue: OOPS track at SAC 2004.

[The99] Dirk Theisen. Enhanced encapsulation in OOP — a practical approach. Master's thesis, Computer Science Department III, University of Bonn, July 1999.

[TJ92] J.-P Talpin and P. Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[VB01] Jan Vitek and Boris Bokowski. Confined Types in Java. *Software Practice & Experience*, 31(6):507–532, May 2001.

[WBW89] R. Wirfs-Brock and B. Wilkerson. Object-oriented design: a responsibility-driven approach. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1989.

[WC07] Tobias Wrigstad and Dave Clarke. Existential owners for ownership types. *Journal of Object Technology*, May 2007.

[WF94] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

[ZPA+07] Yoav Zibin, Alex Potanin, Shay Artzi, Adam Kiezun, and Michael D. Ernst. Object and reference immutability using Java generics. In *Foundations of Software Engineering*, 2007. Submitted for publication.

[ZPV06] Tian Zhao, Jens Palsberg, and Jan Vitek. Type-Based Confinement. *Journal of Functional Programming*, 16(1):83–128, 2006.