Encapsulation Enforcement with Dynamic Ownership

by

Donald Gordon

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

Victoria University of Wellington
2008

Abstract

Unrestricted aliasing is a problem endemic to object oriented programming. It allows notions of encapsulation fundamental to object oriented programming to be violated.

This thesis describes ConstrainedJava, an implementation of a language that provides alias control via a much stronger encapsulation guarantees than traditional object-oriented programming languages, integrated with a constraint system. Unlike most existing aliasing control systems, this encapsulation system integrates well with untyped dynamic languages such as ConstrainedJava. This stronger form of encapsulation has been enhanced to make it easier to write practical programs while still providing useful encapsulation guarantees.

Acknowledgments

Thanks go to my supervisor, James Noble, without whose input this thesis would never have been written.

I'm grateful for the support I received from members of the ELVIS research group – the hard questions, useful suggestions and seemingly endless proofreading were all very helpful.

Thanks are also in order for my family, friends and flatmates, who have manged to put up with and occasionally give useful advice to me during the course of my MSc.

Contents

1 Introduction **1**

2 Background **3**

2.1 Encapsulation 3

2.1.1 Law of Demeter 3

2.2 Aliasing Control 4

2.2.1 Islands 6

2.2.2 Balloon Types 6

2.2.3 Flexible Alias Protection 6

2.2.4 Ownership Types 7

2.2.5 Universes 7

2.2.6 Alias Burying 7

2.2.7 Ownership Types for Object Encapsulation 8

2.2.8 Ownership-Generic Java 8

2.3 Alias protection in dynamic languages 8

2.3.1 Dynamic Alias Protection 9

2.3.2 Object Oriented Encapsulation 10

2.4 Constraints 10

2.4.1 ThingLab 11

2.4.2 Amulet and Garnet 11

2.4.3 Forms/3 12

2.4.4 Change Propagation and Response Graphs 12

2.4.5 Turtle++ 12

3 Dynamic Ownership Structure **15**
3.1 Ownership structure 16
3.2 Core ConstrainedJava 18
3.2.1 Classes 20
3.2.2 Creating objects 21
3.2.3 Sending messages 21
3.2.4 Closures 22
3.3 Ownership in ConstrainedJava 23
3.3.1 Ownership Change 23
3.3.2 Factory methods 24
3.4 Copying Objects 25
3.5 Summary 27

4 Encapsulation Enforcement **29**
4.1 Encapsulation Guarantees 29
4.2 Message types 30
4.2.1 Interpreter structure 31
4.2.2 Internal calls 32
4.2.3 Encapsulation breaking calls 33
4.2.4 External calls 34
4.3 Method types 34
4.3.1 Unrestricted methods 34
4.3.2 Externally independent methods 35
Pure methods 35
Oneway methods 36
4.4 Message send rules 36
4.5 Encapsulation Guarantees 37
4.5.1 Pure methods 38
4.5.2 One-way and unrestricted methods 39

5 Encapsulation Enforcement Extensions **41**
5.1 Interface objects 41

5.1.1 Encapsulation Guarantees 44
5.2 Lending – The equals problem 45
5.2.1 Dynamically scoped object access 45
5.2.2 Temporary ownership transfer 47
5.2.3 Lending all access rights 51
5.2.4 Restricted access loan 52
5.3 Extensions for lending ownership 53
5.3.1 Permit 54
5.3.2 Equivalent 55
5.3.3 Encapsulation Guarantees 56
5.4 Summary 57

6 Encapsulation Enforcement Evaluation 59
6.1 Object Oriented Patterns in ConstrainedJava 59
6.1.1 Proxy 59
6.1.2 Iterator 60
6.1.3 Visitor 60
6.1.4 Composite 61
6.1.5 Factory Method 61
6.1.6 Singleton 62
6.1.7 Observer 62
6.2 Language features 62
6.2.1 Labelling Method Types 63
6.2.2 Closures 65
6.2.3 Inner Classes 66
6.2.4 Calling Java 67
6.3 Implementation 67
6.4 Performance 68

7 Constraints 71
7.1 Introduction 71
7.2 Constraint Lifecyle 72

7.2.1 Establishing a constraint . . . 73
7.2.2 Gathering dependencies . . . 74
7.2.3 Detecting changes . . . 74
7.2.4 Scheduling re-evaluation . . . 76
7.2.5 Constraint Evaluation . . . 76
7.2.6 Activation . . . 77
7.3 Overhead . . . 77
7.3.1 Monitoring . . . 78
7.4 Discussion . . . 79
7.4.1 Ownership-directed simplification . . . 79
7.4.2 Native code . . . 79
7.4.3 Change detection . . . 80
7.4.4 Interaction with the ownership system . . . 80
7.4.5 Scheduling performance . . . 81
7.5 Summary . . . 81

8 Conclusions **83**
8.1 Summary . . . 83
8.2 Contributions . . . 84
8.3 Comparison with previous work . . . 84
8.4 Future work . . . 86

A Message send rules **87**

B Benchmark Code **91**

List of Figures

3.1 Encapsulation: ownership and visibility 17
3.2 Ownership vs References 19
3.3 Sheep clone of an object 26

5.1 The interface problem 42
5.2 Export: the interface solution 42
5.3 The Equals Problem 46
5.4 The Equals Problem: permit 48
5.5 The Equals Problem: equivalent 50
5.6 The Equals Problem: restricted equivalent and permit 52

7.1 Detecting changes: reducing scheduler calls 75

Chapter 1

Introduction

The problem of unrestricted aliasing is endemic to object-oriented programs. Aliasing occurs when, in an object-oriented program, multiple references exist pointing to the same object. If disparate parts of the program hold references to the same object, they can use those references to perform operations using that object's public interface, often including modifying the object. If the object was not intended to be shared in this way, errors can result.

Many approaches have been proposed to solve the problem of unrestricted aliasing [1, 5, 17, 22, 27, 26, 6]. These typically impose restrictions on which objects can hold or store references to other objects. Some notion of ownership has often been used with these systems to decide which objects can hold references to which other objects.

One proposed approach that does not work this way is Dynamic Aliasing Protection [25]. This system eschews restrictions on holding references, instead using an ownership system to restrict the types of messages that can be sent to the references.

This thesis describes ConstrainedJava, which implements an ownership and alias protection system based on Dynamic Aliasing Protection. Along with implementing the base system, ConstrainedJava extends Dynamic Aliasing Protection with a number of features that make writing real

programs within the constraints imposed by the alias protection system easier.

ConstrainedJava also provides a simple one-way constraint system. This makes it easier to propagate state between different parts of a program, and again relieves some of the difficulty placed upon a programmer by having to work within the aliasing protection system.

This thesis is organised as follows:

Chapter 2 describes existing approaches to the problem of unrestricted aliasing and a number of existing constraint systems

Chapter 3 outlines the ownership structure provided by ConstrainedJava

Chapter 4 details the initial form of ConstrainedJava's encapsulation enforcement system, based upon Dynamic Aliasing Protection [25]

Chapter 5 presents the additions we've made to the simple encapsulation enforcement system to support the writing of practical programs

Chapter 6 discusses the ConstrainedJava implementation, and the practicality of writing programs using it

Chapter 7 introduces the one-way constraint system that is part of ConstrainedJava

Chapter 8 provides a summary of the contributions of this thesis, and suggestions for future extensions

Chapter 2

Background

2.1 Encapsulation

One of the most important facilities provided by object oriented programming languages is encapsulation; a way for objects to keep some information and interfaces private and inaccessible to other parts of the program. Typically (in languages such as Java [2], C++ [18] and Smalltalk [14]) this facility is provided by allowing methods and fields of an object to be marked private, making them accessible only from that object or others of the same type.

These simple encapsulation facilities are quite restrictive, however; they provide a coarse grained approach to access control. Either any object with a reference to you can call a public method, or any object of the same type can call any method. Any further control must be implemented by the programmer manually by making sure that references to the object do not leak to parts of the program that are not supposed to access it directly.

2.1.1 Law of Demeter

The Law of Demeter [20] supports better encapsulation and modularity than that supported natively by most object oriented programming lan-

guages. It states:

> For all classes C, and for all methods M attached to C, all objects to which M sends a message must be instances of classes associated with the following classes:
>
> 1. The argument classes of M (including C).
> 2. The instance variable classes (i.e. fields) of C.

Following the Law of Demeter means an object avoids sending messages to objects that are not part of it or not passed to it as a method argument. Messages can only be sent to other objects indirectly. Each call must be from an object to one of its parts, or a method parameter, which in turn may call more such wrapper methods until finally the method on the intended target object is called.

Experimental work [3] has found that a metric (Response For a Class) that is reduced by following the Law of Demeter has a positive correlation with faults in software.

2.2 Aliasing Control

Aliasing occurs in object-oriented systems when multiple references – aliases – exist which point to the same object. Object-oriented languages typically do not impose access controls on entire objects: therefore, maintaining a reference to an object allows unrestricted access to that object's public interface. The Law of Demeter tries to rectify this problem by avoiding sending messages to references other than object parts or method arguments.

The ability to send messages to any reference (unrestricted aliasing) can become a problem when part of a program ends up with a reference to an object which is part of the internal mutable representation of some other aggregate object when this object was not intended to be shared.

A classic example of the problems with unrestricted aliasing is the code-signing hole [21] in an early version of the Sun Java Development Kit. In this version, when an application asks for the list of digital signatures that apply to it, a reference to the mutable array used by the interpreter to keep track of the application's signatures is returned, instead of a copy. The application can then modify this array to include signatures of other loaded classes, behaviour not expected by the Java Security Manager's developer.

Unrestricted aliases to an object are only a problem if those aliases can be used to access that object's mutable state. An immutable object's state cannot be changed, and thus many references to it may be safely held.

The unrestricted aliasing problem is widespread and pervasive in object-oriented programs, due to a lack of restrictions on how objects can reference one another.

Unrestricted aliasing is fundamentally a problem with encapsulation. Typically, object-oriented languages provide an encapsulation mechanism consisting of a facility to mark members of an object or class public or private. This access control is fairly broad. It only allows a very limited specification of which objects may call certain methods. The other part of encapsulation – making sure references to the object are not handed out to inappropriate parts of the program – is usually not directly supported by the language, and is therefore spread throughout the code on an ad-hoc basis.

Many systems have been proposed to deal with this problem of unrestricted aliasing. Most of these existing systems rely on language facilities which are often absent in object oriented dynamic languages – explicit typing, a single compile time, and a rigid notion of class. These same features are common to most class-based object oriented languages, such as Java [2], C++ [18] and C♯ [10].

2.2.1 Islands

Islands [17] enforce encapsulation in a similar way to balloon types: groups of objects, known as Islands, are defined. Each Island has a bridge object; static references to objects within the island other than the bridge from objects outside the island are disallowed.

2.2.2 Balloon Types

The Balloon Types system [1] allows classes to be declared as being balloons. Balloon objects cannot have more than one reference to them held at any one time, and objects not encapsulated by a balloon object may not refer to the objects that the balloon encapsulates. Therefore, balloon objects can not be aliased, and the objects encapsulated by a balloon may not be aliased by objects outside of the balloon. Balloon Types enforces these restriction by a compile-time full program analysis, unlike most other static systems which merely enforce simple local rules at compile-time.

2.2.3 Flexible Alias Protection

Flexible Alias Protection (FLAP) [26] is a system for enforcing encapsulation. FLAP defines a set of modes (rep, arg and free mode) that can be used to annotate variable definitions and object constructions, which are then propagated through the program. This set of modes are used to statically enforce a set of invariants:

- *No Representation Exposure:* Component objects which make up an aggregate object's representation (rep mode) should not be returned to the rest of the system.

- *No Argument Dependence:* An expression referring to an object which is an argument of an aggregate object (arg mode) cannot be used to access that object's mutable state.

- *No Role Confusion:* Expressions of a mode other than free cannot be assigned to a variable of another mode.

These invariants are enforced by ensuring that references held in one mode may be converted to references held in another mode only in certain circumstances, and disallowing certain operations with references held in a certain mode. For instance, references held in arg mode may only be used to call clean methods, which are not allowed to access mutable state.

2.2.4 Ownership Types

Ownership types [9] is a static type system that provides ownership information. Objects own object contexts – types are annotated with context declarations, to produce ownership types. Then, variables with ownership types referring to different contexts cannot be referring to the same object. This ownership information is then used to provide a mechanism to limit the visibility of object references. Later work [8] extends this to allow support for interface objects, borrowing, and numerous other extensions.

2.2.5 Universes

Universes [22] controls representation exposure by partitioning a program's object into universes – components within the system. Universes are hierarchical, and are enclosed by a root universe.

2.2.6 Alias Burying

Alias Burying [6] uses static intraprocedural analysis to enforce a number of invariants on procedure parameters and return values based upon annotations applied to these procedures. Available annotations include *unique*, which guarantees that a reference is unaliased, *borrow*, which guarantees that the procedure passed the reference will not cause more aliases to be generated, and variations upon these.

2.2.7 Ownership Types for Object Encapsulation

Ownership Types for Object Encapsulation [5] is a type system based on Ownership Types [9] to enforce object encapsulation. In an attempt to make common patterns such as iterators implementable, it allows privileged access from inner classes to their outer classes. It still allows reasoning about classes, as a class and its inner classes are considered to be one unit, a module. However, this approach does reduce the granularity of the encapsulation controls provided.

2.2.8 Ownership-Generic Java

Ownership Generic Java[27] provides a static ownership system on top of the Java language, utilising Java's existing type checking and generic parameters to provide a simple extension to the language to support ownership. This is achieved by adding an owner type parameter to every class, and ensuring via subtyping that this owner information is preserved.

2.3 Alias protection in dynamic languages

The problem with these existing approaches to alias protection is they're tied to class-based static languages such as C++[18] and Java[2]. They all impose their restrictions statically at compile-time. This presents a problem for dynamic languages such as Smalltalk[14] and Ruby[30], which have no whole-program compile-time at which to perform verification, and in the case of prototype-based languages such as Self[31] and NewtonScript[29], no classes. Clearly, therefore, a different approach is needed to solve the problem of unrestricted aliasing in dynamic languages.

2.3.1 Dynamic Alias Protection

Dynamic alias protection [25] is an idea for enforcing encapsulation in a program that has an ownership structure like that described in chapter 3. It grew out of earlier work on Flexible Alias Protection (section 2.2.3). Despite the name, Dynamic Alias Protection does not restrict aliasing; rather, it removes the harmful effects of aliasing by ensuring that the mutable state of aliased objects cannot be accessed by any object in the system other than that object's owner.

The Dynamic alias protection paper proposes enforcing encapsulation by categorising the objects within an aggregate object into the aggregate's representation – the objects within the aggregate that make up its mutable state, and may not be accessed from outside, and the aggregate's arguments – the objects which may be accessed from outside the aggregate, provided they are treated as immutable.

Encapsulation can then be enforced by imposing three invariants: representation encapsulation, argument independence and no role confusion. These are the same invariants as those used by Flexible Alias Protection (section 2.2.3), restated in terms of the Dynamic Alias Protection ownership model.

An aggregate object's representation encapsulation is violated when the mutable objects making that aggregate object's representation are accessed directly by other objects in the system. This requires bypassing the interface provided by the aggregate object. For example, accessing mutable node objects that are part of a linked list by objects other than the linked list of which those nodes are a part would be a violation of the list's representation encapsulation.

Argument dependence occurs when an aggregate object depends on its arguments' mutable state. For example, a list depending on the state of the items contained within it, when said items were not owned by the list, would be exhibiting argument dependence.

This scheme of allowing access to arguments and parts of an aggregate

object is similar to the restrictions imposed when following the Law of Demeter, although flexible alias protection imposes more restrictions on messages sent to arguments than Demeter does.

Dynamic Alias Protection has not previously been implemented.

2.3.2 Object Oriented Encapsulation

The object-oriented encapsulation system[28] provides an encapsulation enforcement system more suited to dynamic languages. It allows classes to restrict how their subclasses can utilise them, and allows the production of references with associated access policies allowing only certain sets of methods to be called using them.

2.4 Constraints

Many programs have dependencies between the runtime state of their parts, which are only described as implicit properties of the program's implementation, and must be explicitly managed by the programmer.

A constraint system allows programmers to define explicit relationships between the state of parts of a program – relationships that would normally have been coded implicitly, often by explicitly updating one field when one or more others change.

A typical relationship a constraint system might enforce could be a relationship between widget locations – widget a's left hand edge should be 40 pixels to the right of widget b's left hand edge:

$$a.x = b.x + 40$$

Constraint systems come in several varieties. A one-way constraint system would interpret the $=$ symbol as assignment: setting the value of $a.x$ to be equal to $b.x + 40$, and updated when $b.x$ changes. A two-way constraint system would consider the equation as an invariant to maintain,

and would be able to resolve the constraint by modifying either $a.x$ or $b.x$ if the other was changed. Two-way constraint systems must employ more complex constraint solvers such as the DeltaBlue [11] algorithm.

A number of constraint systems have been developed. Here we review related work on imperative object-oriented constraint systems.

2.4.1 ThingLab

ThingLab [4] provides a two-way constraint system on top of Smalltalk-80. It allows constraints to be defined between objects, and some parts of structure to be shared between objects.

2.4.2 Amulet and Garnet

Amulet [23] (and its predecessor, Garnet) provides a simple one-way constraint system. Each implements a custom object system on top of its base language. Constraints are expressed by placing a block of code into a field instead of a simple value; when the field is read, the code is evaluated and the result returned. The constraint system caches these return values and monitors the fields read by the code block to detect cache invalidations.

The constraint system in Amulet/Garnet is used extensively in the user interface toolkit it provides. Garnet's radio button widget uses 58 constraints internally, and the Lapidary graphical editor contains over 16,700 constraints. The graphics system is connected to the object system, ensuring that widgets are automatically updated when field values relating to them change.

The problem of a constraint's output being sent to a single field is solved by Amulet through allowing the code that is evaluated to determine the value of the constraint to modify other fields as a side-effect. Therefore, a constraint that produced a co-ordinate pair might be placed in a field containing the x co-ordinate, and update the y co-ordinate as a side-effect.

Garnet and Amulet allow the use of pointers in constraints [32]. This makes the re-use of constraints in similar situations very easy. For example, a feedback object needing to be positioned beside the currently selected item in a menu could have this done with a constraint of the form:

$$feedback.position = self.obj\text{-}over.position + offset$$

This relies on the reference *self.obj-over* to be able to change, something not possible with constraint systems that do not allow pointer variables in constraints.

2.4.3 Forms/3

Forms/3 [7] is a visual language utilising a one-way constraint system based on a spreadsheet paradigm. The spreadsheet paradigm provides a constraint system familiar to many computer users. Forms/3 allows programs to be specified as forms, containing within them sets of cells.

Interaction is supported by two concepts: that of time, and that of event queues. The concept of time allows animation, and the idea that cells can hold different values at different times.

Forms/3 re-evaluates formulae contained in cells lazily. Graphical elements can be contained within cells, and parameterised by formulae.

2.4.4 Change Propagation and Response Graphs

Change Propagation and Response Graphs [16] provides a mechanism to propagate changes between components in a program, based on specified relationships between them. The system also provides multiple view consistency, undo-redo, versioning and cooperative work facilities.

2.4.5 Turtle++

Turtle++ [19] is a constraint imperative language based on Turtle [15], implemented on top of C++ using templates and operator overloading.

It allows constrained variables to be defined, and multiway constraints placed upon them with a natural C++ syntax. New constraint solvers can be plugged in, and user-defined constraints can be easily added.

Chapter 3

Dynamic Ownership Structure

We have developed a language, Constrained Java, to experiment with Dynamic Ownership. Dynamic Ownership is our attempt to solve the problem of alias protection in a form that is applicable to dynamic languages.

Unlike other alias protection schemes, such as Balloons [1] and Islands [17], Dynamic Ownership places no restrictions on which objects or classes in the system may hold pointers to other objects or classes in the system. There are a number of features typical of static languages that dynamic languages do not possess, and which Dynamic Ownership therefore cannot rely on. Dynamic languages often have a weak notion of typing, so class-based restrictions cannot be used. Many dynamic languages allow code to be added to a running program. This means that we cannot merely run a checker over the entire program to check for violations of our ownership rules during a compile step: the code that constitutes the whole program may be augmented after this occurs. This is especially true in languages like Self [31], Smalltalk [14], Ruby [30], and even Java [2].

Our model for dynamic ownership provides alias protection and encapsulation enforcement by maintaining a notion of object ownership, and then placing restrictions on messages sent between objects based on this idea of ownership. The restrictions are based solely on the idea of object ownership; ordinary references between objects are not restricted in any

way, and do not affect the behaviour of the encapsulation enforcement. Unrestricted references allow, for instance, container objects such as lists which hold references to a set of objects but do not own them (see figure 3.2). As the encapsulation restrictions are based on runtime behaviour rather than compile-time structure, they are ideally suited to implementation in a dynamic language.

This chapter describes the ownership structure provided by the Dynamic Ownership system. The next chapter describes how the information provided by this structure is employed to dynamically enforce encapsulation.

3.1 Ownership structure

Dynamic Ownership imbues a notion of object ownership into a program by giving each object an *owner*, which refers to some other object in the system which owns it.

An object a in the ownership tree is said to *own* an object b when b.owner = a. For example, in our example of the house (figure 3.1), the Lounge *owns* the Television.

An object a in the ownership tree is said to *be owned by* an object b when a.owner = b. For example, in our example of the house, the RemoteControl is *owned by* the Lounge.

An object a in the ownership tree is said to *contain* an object b if there is some path from b to a following owner pointers. In figure 3.1 the House *contains* the Television, but does not *own* it.

An object a is said to be *visible* to an object b if there exists some object c such that *c owns a* and *c contains b*. This means that both a and b are encapsulated within c, and b can access a without breaking through encapsulation boundaries. In the example figure, the House, Kitchen, Street and SecondHouse are all *visible* to the Lounge. This is an incoming visibility; the Street is visible to the Lounge, but the Lounge is not visible

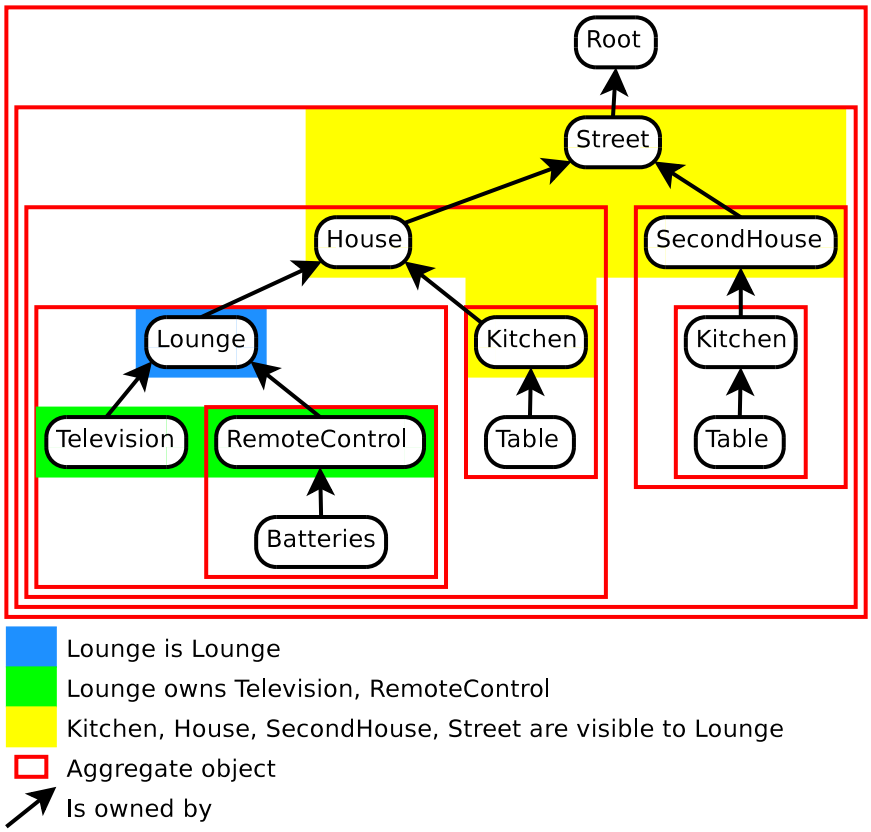

Figure 3.1: Encapsulation: ownership and visibility

to the Street. Lounge would not be breaking through an encapsulation boundary to access SecondHouse, as it is owned by Street; however, direct access to the SecondHouse's Kitchen from Lounge would be bypassing the encapsulation boundary provided by SecondHouse.

Objects which would otherwise have no owner, such as the first objects created by a program, are owned by a special node in the ownership tree called the *root* owner.

The objects in this scheme can be thought of as a tree of encapsulated objects, being comprised of the set of the other objects they *contain* that make up their representation. Every node in the ownership tree then marks an encapsulation boundary: an object's owner is considered the *interface* through which other parts of the system should interact with the object.

The tree of ownership pointers need bear no relation to the graph of other pointers in the system. For example, figure 3.2 shows a system with a house object containing a number of rooms, including a lounge and a kitchen. The house owns a list and the rooms within the house, but not the links in the list. We think of the house as the interface to the list and its contents, and the list as the interface to the list's structure (the links), but not the objects held in the list. As each object's ownership information is explicitly stored in the object itself, this information is retained when the object (such as a room) is placed in a container object (such as the house's list of rooms) – the ownership information is not lost or changed.

3.2 Core ConstrainedJava

We have designed and implemented the ConstrainedJava language to experiment with the Dynamic Ownership system, and the encapsulation enforcement described in the next chapter.

ConstrainedJava is based on the BeanShell [24] language, with extensions to handle ownership, encapsulation enforcement and constraints. ConstrainedJava is not statically typed.

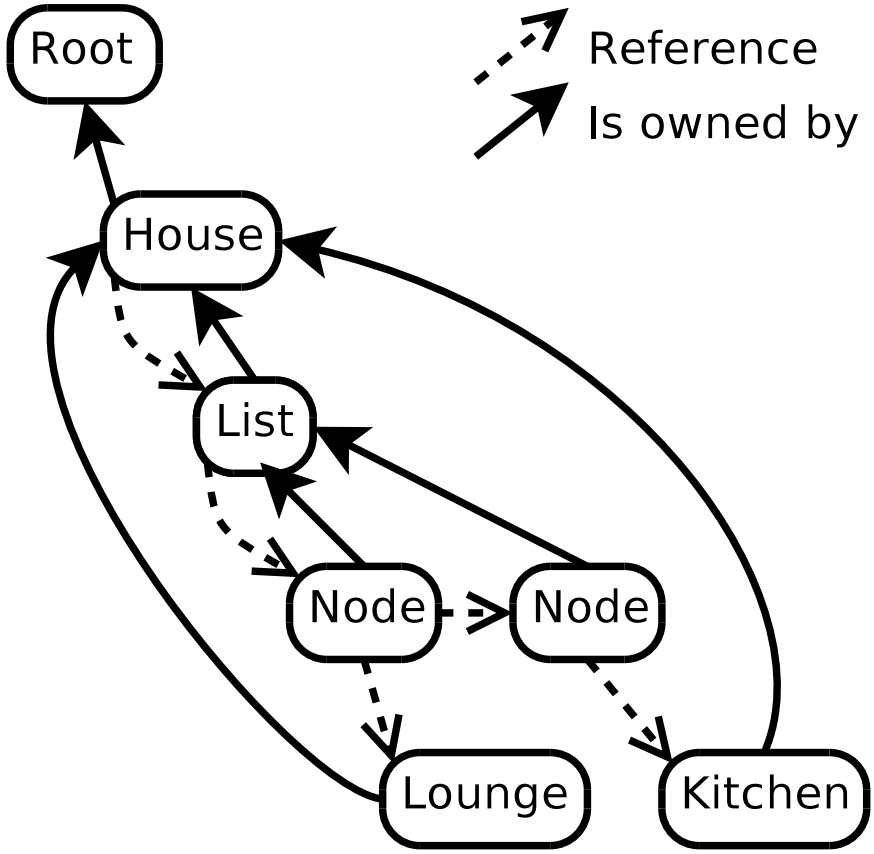

Figure 3.2: Ownership vs References

ConstrainedJava gets most of its syntax from BeanShell, which in turn attempts to be similar to Java. Syntactically, the language appears like a dynamically typed Java. Classes, methods, fields and the like are declared in a similar manner to Java, except that types can be omitted entirely or replaced with the "anything" type var. The following code example demonstrates this with a simple factorial function:

```
class MathStuff {
  ...
  factorial(x) {
    if (x < 2) return 1;
    return x * factorial(x - 1);
  }
}
```

Additions to the ConstrainedJava language beyond the base BeanShell functionality include Smalltalk-style blocks, and a means of interacting with the ownership system.

3.2.1 Classes

ConstrainedJava supports classes and single inheritance. Interfaces are not supported, as dynamic typing means they're not needed. Classes are defined with the class keyword, followed by a classname, and optionally the extends keyword along with the name of the class to inherit from. Fields can be declared inside them, with the keyword var or the name of a type, followed by the name of the field. Methods are declared by specifying an optional return type, followed by the method name, the arguments, and the method body.

The class's constructor is merely a method with the same name as the class and no return type.

For example, the following class implements a simple rectangle that can draw itself:

```
class Rectangle extends Drawable {
  var topLeft;
  var bottomRight;

  Rectangle(_topLeft, _bottomRight) {
    topLeft = _topLeft; bottomRight = _bottomRight;
  }

  draw(g) {
    g.drawRect(topLeft.getX(), topLeft.getY(),
      bottomRight.getX(), bottomRight.getY());
  }
}
```

The class defines two fields, topLeft and bottomRight. It also defines a constructor taking default values for those fields, and a method to draw the rectangle when passed a java.awt.Graphics object.

3.2.2 Creating objects

An object is constructed in the same way it would be in ordinary Java: using the new keyword. A new object's owner is initially the object that called its constructor. In the following example, the RemoteControl object's owner would be the TelevisionSet object that created it:

```
class TelevisionSet {
  var myRemote = new RemoteControl(this); ...
```

3.2.3 Sending messages

Message sending is also syntactically similar to Java. Message sends are dynamically checked to ensure they don't violate the rules of the encapsulation enforcement part of Constrained Java (chapter 4).

The following code demonstrates the syntax for sending messages:

```
someObject.callMethod();

class Thing {
  callMethod() {
    ...
} }
```

3.2.4 Closures

ConstrainedJava provides the ability to define closures. A closure, when created, acts like an object with a single method, value(). The code executes in the scope of the method in which it was defined, and is able to access the local variables defined in that method. The value method returns the result of the last expression, unless a return statement is executed, in which case the method declaring the closure returns the value provided to the return statement in the closure.

For example, a method that returns the first item in a list greater than some specified value could use a closure as follows:

```
class MathStuff {
  ...
  firstAbove(list, a) {
    list.forEach(block(b) {
      if (b.above(a)) return a;
    });
    return null; // if no match
  }
}
```

In this code, the forEach method on the list is passed a block. The block is defined to take one parameter, and return a value depending on

a condition. When the block executes the statement "return a", the stack will be unwound, the forEach loop will be terminated, and the caller of the firstAbove method will have the value of the variable a returned to it. If no match is found, the method returns null.

3.3 Ownership in ConstrainedJava

ConstrainedJava implements the Dynamic Ownership structure described in section 3.1.

An owner pointer is present in every object, which can be read by calling the owner() method. Operations are provided to make use of and change these owner pointers.

For example, the code below informs an object's owner that one of its children has changed, when that child's setChanged() method is called.

```
...
setChanged() {
  owner().setChildChanged(this);
}
...
```

Note that an object does not have control over which object owns it; therefore, if the owner() method is used in this way, care must be taken to ensure that the owner is of an appropriate type.

3.3.1 Ownership Change

Dynamic Ownership allows object ownership to be changed at runtime. Ownership change causes an object to move to a new location in the ownership tree. We check that this does not create a cycle in the ownership graph by ensuring that the object whose ownership is being changed does not contain its new owner; this ensures the ownership graph remains a tree.

Operations which affect the ownership of an object may only be performed by the object's current owner.

Changes of ownership may be initiated with the gift method, which when sent to an object by its owner causes that object's owner to be changed to the object specified as the method's only parameter:

```
class thingy {
...
  line = new Line(p1, p2);
  // this owns line
  line.gift(drawing);
  // now, drawing owns line
  drawing.add(line);
}
```

Changing an object's ownership moves it within the ownership tree, changing the object it is encapsulated by. This change occurs completely independently of any references held by any objects, including the previous owner, the new owner or the object whose ownership is being changed may hold. Dynamic Ownership does not require or enforce the tree of ownership pointers to have any relationship with any other references in the program.

3.3.2 Factory methods

A common case of ownership change is a factory method – one which returns an object for use by another part of the system. Factory methods are commonly used to create a new object (or retrieve one from a pool), perform some extra initialisation on it, and then return the new object for use by the factory method's caller.

ConstrainedJava's ownership system supports factory methods through the *factory* method modifier, which transfers the ownership of the returned

object to the factory method's caller. If the object containing the factory method does not own the object returned by that method, an OwnershipError exception will be generated.

The example method below creates a new Widget object, adds it to a list of widgets, and then returns the new object. As the method is marked as being a factory method, the object that called the newWidget method will gain ownership of the new Widget object returned by the method.

```
class WidgetFactory {
  ...
  factory newWidget(a) {
    w = new Widget(a);
    widgetList.add(w);
    return w;
  }
}
```

This facility for factory methods could be emulated with the gift method described in the previous section – it does not add any capabilities to the ownership system. However, the requirement to transfer ownership of an object to a method's caller is present sufficiently often that it is nonetheless a useful feature to have.

3.4 Copying Objects

Having ownership information available allows the implementation of an ownership-directed clone, or *sheep clone* [25].

The sheep clone is somewhere between a shallow clone, copying only the target object, and a deep clone, recursively copying the target and all objects it refers to. It acts as a "do what I mean" clone for aggregate objects, only copying objects which are contained by the object of which copying is initially requested.

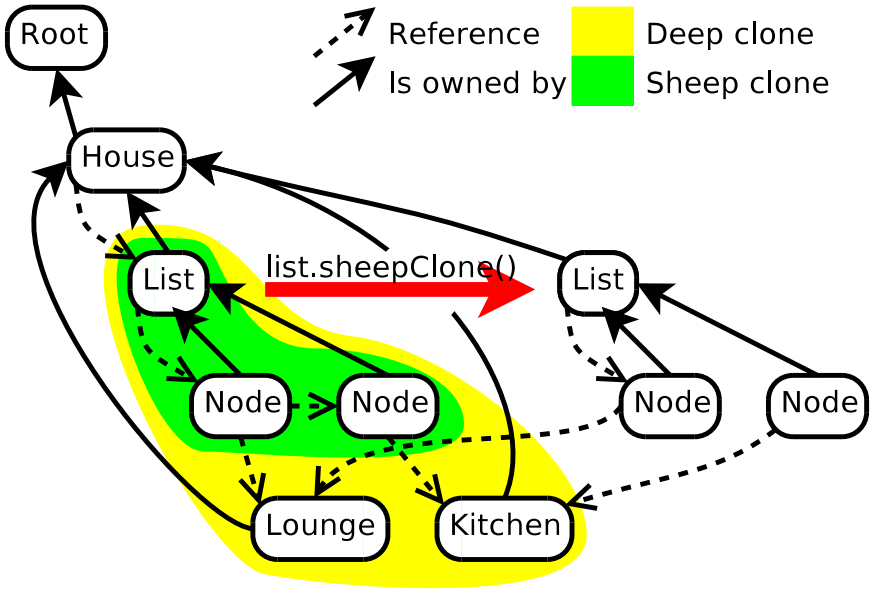

Figure 3.3: Sheep clone of an object

Like a deep clone, it recursively follows references from the cloned object to find other objects as candidates for cloning. But unlike a deep clone, not all of these objects are cloned. The ownership tree is consulted; only references to objects *contained* by the object which is the subject of the clone are followed. References to objects in other parts of the ownership tree are retained in the cloned object graph.

For example, in figure 3.3, a sheep clone of the List object is requested. The sheep clone operation follows the link from the list to the first node, and the first node to the second node, and each time, successfully checks that the new object found is contained within the list. When the sheep copy operation follows the references from the nodes to the Lounge and Kitchen, it checks if they're contained within the list by following owner pointers. As this check fails, the Lounge and Kitchen objects are not marked for copying.

Then the objects identified for copying by the last step are copied. The

ownership structure of the original object tree is replicated, as is the set of references between the copied objects. References to objects outside the set of copied objects are retained, so the copied nodes still have pointers to the same lounge and kitchen objects as the original nodes.

3.5 Summary

ConstrainedJava adds an ownership system to a Java-like dynamic language, BeanShell. Each object in the system has an owner pointer pointing at some other object; the owner pointers are constrained to form a tree. The ownership structure is exposed to user programs, and may be manipulated and queried.

The next chapter shows how we employ the ownership system to enforce encapsulation, by imposing limitations on message sends between objects based on those objects' relative positions in the ownership tree.

Chapter 4

Encapsulation Enforcement

In this chapter, ConstrainedJava's system for enforcing encapsulation is described. The encapsulation enforcement is based on the dynamic ownership structure described in chapter 3.

4.1 Encapsulation Guarantees

ConstrainedJava combines the ownership structure provided by Dynamic Ownership with a set of invariants to provide object encapsulation. The invariants are enforced by restricting the types of messages that can be sent between objects depending on their ownership relationships.

The base invariants (taken from Dynamic Alias Protection [25]) that ConstrainedJava enforces are representation encapsulation and external independence.

Representation encapsulation requires that an object may only be accessed by messages passing though its interface: to access the state of objects that an object encapsulates (the encapsulating object's *representation*), the call sequence must go through the encapsulating object, as it owns the objects that make up its representation.

External independence means that an object must not be dependent on the mutable state of objects that are external to it. We consider an object which is able to gain any information at all derived from another object's mutable state as being dependent on that object's mutable state. Objects external to an encapsulating object are defined as any object in the system that the object holds or can obtain a reference to, but does not own or encapsulate. This includes references passed as parameters, stored in fields, and returned by method calls. We sometimes refer to these external objects as an object's *arguments*.

These invariants together effectively enforce encapsulation by ensuring that the mutable state of objects $p_1 ... p_n$ that form part of an object o can only be accessed by o sending messages to them. The ownership system is employed to determine that an object p is part of an object o because o *owns* p, and a set of rules on message sends are used to realise the invariants.

The invariants impose similar restrictions to those of the Law of Demeter (see section 2.1.1). The major differences are due to ConstrainedJava having a more explicit idea of the relationship between an object and its parts, and therefore being able to be more permissive with calls that do not return information about mutable state.

No restrictions are required by the invariants above, or imposed by the rules described later that realise them, about objects being disallowed from holding certain references.

This lack of restrictions on references is a major difference from most systems designed to provide aliasing protection. ConstrainedJava places no restrictions on passing or storing references; only the types of method called on those references are restricted.

4.2 Message types

In order to achieve representation encapsulation and external independence, ConstrainedJava classifies message sends into three categories based

on the relative positions of the message sender and receiver in the ownership tree. The categories are: internal calls, encapsulation breaking calls, and external calls; these categories are each described below.

4.2.1 Interpreter structure

The code in the next sections is intended as pseudocode for an interpreter of the message send rules, to unambiguously express their semantics. This technique of providing an interpreter to describe a dynamic language is quite common, being used to define Smalltalk [14]. We only concern ourselves with message dispatch: other parts of the interpreter are standard – inheritance, running code bodies, control flow, variables – so we do not consider them. In section 4.5 we provide a case analysis that demonstrates that these rules enforce the invariants identified in section 4.1.

We consider all the message checks performed below to be indicative of the structure of a function in an interpreter called to determine if a message send is allowed, from code similar to the following. The sendMessage function sends a message to a method named target from an object sender to another object receiver, with arguments args. The isPure parameter is an auxiliary parameter which tracks where the message is able to return information about mutable state – it's set when the message is being sent from a method marked *pure* (pure methods are described in section 4.3.2).

```
sendMessage(sender, receiver, target, isPure, args) {
  // call from sender to method target on receiver
  // ...
  if (checkMessageSend(sender, receiver, target, isPure)) // ok
    dispatchMessage(sender, receiver, target, args);
  else
    throw new OwnershipException(sender, receiver, target);
  // ...
}
```

```
checkMessageSend(sender, receiver, target, isPure) {
  messageType = getMessageType(sender, receiver);
  methodType = target.getMethodType();
  if (isPure && methodType != EXTERNALLY_INDEPENDENT)
    return false;
  allowed = getAllowedMethodTypes(messageType);
  if (allowed == ALLOW_NONE) return false;
  if (allowed == ALLOW_ALL) return true;
  if (methodType == EXTERNALLY_INDEPENDENT &&
      allowed == ALLOW_EXTERNALLY_INDEPENDENT)
    return true;
  return false;
}
```

The full set of rules, including chapter 5's extensions, are reproduced in appendix A.

The code samples categorising message types in the following three sections would be placed in the getMessageType() method referenced in the sample interpreter code above.

4.2.2 Internal calls

Internal calls are those from an object to itself (a *self call*), or to an object owned by the calling object (an *owner call*). By definition, an internal call cannot cause the principles of representation encapsulation or external dependence to be violated; an internal call is one from an object to itself or some part of its representation.

```
getMessageType(sender, receiver) {
  if (sender.is(receiver) ||
      sender.owns(receiver)) return INTERNAL_CALL;
```

```
  // ... other cases
}

is(other) {
  return this == other;
}
owns(other) {
  return other.owner == this;
}
```

4.2.3 Encapsulation breaking calls

Encapsulation breaking calls are those from an object to another object which is not *visible* to the calling object. An object a is said to be *visible* to an object b if there exists some object c such that *c owns b* and *c contains a*.

Encapsulation breaking calls violate both of the encapsulation invariants: they are calls to objects a which bypass the owner b of those objects, thus bypassing the encapsulation provided by object b, and thus violating the representation encapsulation invariant.

```
getMessageType(sender, receiver) {
  if (!receiver.visibleTo(sender))
    return ENCAPSULATION_BREAKING;
  // ... other cases
}

visibleTo(other) { // is this visible to other?
  return this.owner.contains(other);
}

contains(other) {
  if (other.is(this)) return true;
```

```
    if (other == null) return false;
    return this.contains(other.owner);
  }
```

4.2.4 External calls

External calls are those that do not fit into the above categories: they are calls from an object to another object that is *visible* to it, but not *owned* by it. They're calls to objects which aren't part of the representation of the sending object, and are therefore external to the sending object – they are the sending object's *arguments*.

```
getMessageType(sender, receiver) {
  if (receiver.visibleTo(sender) &&
      !sender.is(receiver) &&
      !sender.owns(receiver)) return EXTERNAL_CALL;
  // ... other cases
}
```

4.3 Method types

Dynamic Ownership's encapsulation enforcement system works by applying a set of rules to message sends. In order to specify this set of rules, methods must be classified into two groups: unrestricted, and externally independent. Externally independent methods are further subdivided into two types: oneway and pure methods.

4.3.1 Unrestricted methods

Normal, or unrestricted methods, are those which are able to read and write mutable state, and to return some value or exception. A normal method can perform any operation on the object it is part of.

Access to normal methods must be restricted, as they allow any action to be undertaken. If any part of the program could call any unrestricted method, then encapsulation boundaries would be bypassed, thus violating the principles of representation encapsulation and external independence.

4.3.2 Externally independent methods

Externally independent methods are defined as not returning any information to the caller about any object's mutable state. An equals() method that compares the mutable state of an object with another is not externally independent, as it returns information derived from mutable state.

A method returning a hash of an immutable String object would be externally independent, as it does not return information about or derived from mutable state. Likewise, an addElement method on a list would be externally independent, as while it accesses mutable state, it returns no information about or derived from it.

Dynamic Ownership Enforcement achieves this by further classifying them into two types: pure and one-way methods.

Pure methods

Pure methods are defined as being unable to return information about mutable state. This means that they are unable to access non-final fields, or send messages that would cause information about non-final fields to be returned to the pure method (i.e. messages to externally dependant methods). When a pure method sends a message, the isPure parameter in the pseudocode sendMessage() function in section 4.2.1 would be set, ensuring this.

Uses for pure messages include retrieving information about immutable objects, such as a character from an immutable string object or a co-ordinate from an immutable point object.

Oneway methods

Oneway methods are defined as being able to perform any action, like a normal method, including calling normal methods, apart from returning a value or exception.

As oneway methods are allowed to access mutable state just as normal messages may, they are useful for gathering output such as updates to window display, sending test results in a test harness, implementing an assert facility, or providing additions to an HTML document to be sent to a web browser. For example, the method to add items to a list could be marked oneway, as it does not need to return any information.

Unlike pure methods, messages sent from oneway methods would not set the isPure flag in calls to sendMessage (section 4.2.1), as this does not affect the oneway method's external dependence – it will still be unable to return information derived from mutable state.

4.4 Message send rules

ConstrainedJava enforces ownership restrictions by only allowing certain message send operations. To do this, we categorise message sends, and the methods they are sent to, and then apply a set of rules to see if the message is allowed.

The code samples illustrating the message send rules in the following sections would be placed in the getAllowedMethodTypes() method called by the sample interpreter code in section 4.2.1.

If the message is an internal call, then it is allowed regardless of the message type. As internal calls are made by an object to itself or its representation, they do not violate the principles of representation encapsulation or external dependence.

```
if (messageType == INTERNAL_CALL)
  return ALLOW_ALL;
```

These are the only unrestricted messages that may be sent. Other messages cross encapsulation boundaries, and thus must be subject to additional restrictions.

Obviously, if a message is classed as *encapsulation breaking*, then it cannot be allowed to be sent, regardless of the method type of the callee; such calls violate both of the flexible aliasing principles.

```
if (messageType == ENCAPSULATION_BREAKING)
  return ALLOW_NONE;
```

The other possible message classification is that of an *external call*. As an external call is by definition not made to the calling object's representation, it must be made to some object that is external to the calling object.

The principle of external independence states that an object must not depend on the mutable state of objects that are external to it – objects not contained within its representation. As external calls are only made to objects not part of the sender's representation, they must not cause the calling object to learn of the callee's mutable state.

```
if (messageType == EXTERNAL_CALL)
  return ALLOW_EXTERNALLY_INDEPENDENT;
```

4.5 Encapsulation Guarantees

In this section, we show how the rules in sections 4.2 to 4.4 satisfy the encapsulation guarantees stated in section 4.1.

We recapitulate these guarantees here:

Representation encapsulation requires that an object's mutable state may only be retrieved by messages passing though its interface: to access the mutable state of objects that make up an encapsulating object (the encapsulating object's *representation*), the call sequence must go

through the encapsulating object, as it owns the objects that make up its representation.

External independence means that an object must not be dependent on the mutable state of objects that are external to it. We consider an object which is able to gain any information at all derived from another object's mutable state as being dependent on that object's mutable state. Objects external to an encapsulating object are defined as any object in the system that the object holds or can obtain a reference to, but does not own or encapsulate. This includes references passed as parameters, stored in fields, and returned by method calls. We sometimes refer to these external objects as an object's *arguments*.

Our argument proceeds as follows: we perform a case analysis of all possible message sends. First we consider messages sent from pure methods, which have special restrictions placed on them, and then we consider messages sent from all other method types. We classify messages by the type of the sending and receiving methods, and the ownership relations between the sending and receiving objects.

4.5.1 Pure methods

Pure methods are a special case, as they are unable to send messages to unrestricted methods.

When a message is sent from a pure method in an object to any object that is not visible or owned by the object containing this method, this message is disallowed. If they were allowed, such messages would violate the representation encapsulation of the owner of the receiving object.

When a message is sent from a pure method in an object to an unrestricted method in any object, this message is disallowed. Unrestricted methods can return information about mutable state, which pure methods are not allowed to access.

All other messages from pure methods must be sent to externally independent methods in objects either owned by or visible to the sending objects. These messages do not break the representation encapsulation of the receiving object, as they do not allow information about its mutable state to be returned. They don't break the external independence of the sending object, as they return no information about mutable state to it.

4.5.2 One-way and unrestricted methods

Non-pure methods may send messages of any type – unrestricted, or externally independent – within the bounds of the message send rules.

When a message is sent from an object to itself, the message is allowed. As this method is from an object to itself, it does not cross any encapsulation boundaries, and therefore does not break the representation encapsulation rule. Also, as it is not to an external object, it does not cause the sender to become externally dependent.

When a message is sent from an object to an object it owns, the message is allowed. This does not break the representation encapsulation of the receiving object or its owner, as the receiving object is part of the sender's representation. It does not cause the sender to become externally dependent, as the receiving object is not external to the sender.

When a message is sent from an object to an unrestricted method on an object it does not own, the message is disallowed. Allowing it would violate the representation encapsulation of the owner of the receiving object, as that owner object would be bypassed. Additionally, if the receiving object was not contained by the sending object, then the sender would become externally dependent on the mutable state of the receiving object.

When a message is sent from an object to an externally independent method on an object visible to the sending object, the message is allowed. This does not break representation encapsulation, as the definition of *visible* (see section 3.1) ensures that the receiving object is within the representation

of an object that contains the sending object. And external independence is maintained, as when the method returns it is unable to return any information derived from mutable state, which would cause the sending object to become dependent on the receiver.

When a message is sent from an object to an externally independent method on an object not visible to or owned by the sending object, the message is disallowed. Allowing such messages would violate the representation encapsulation of the owner of the receiving object, by bypassing it. Such messages, if allowed, would not violate external independence, as the call is made to an externally independent method which cannot return information derived from mutable state.

Thus, no case of message sends violates the invariants.

Chapter 5

Encapsulation Enforcement Extensions

ConstrainedJava introduces a number of extensions to the base encapsulation enforcement system described in chapter 4. While trying to use this system, we discovered a number of problems that made writing real programs difficult. These extensions resolve many of these problems, making it easier to write real programs within the constraints of the ownership system. They provide support for interface objects such as iterators, and make it possible to access method parameters in certain well-defined circumstances allowing Java-style equals() methods to be implemented. Section 5.3 then defines the final form of the parameter access extensions and their interaction with invariants provided by the encapsulation enforcement system.

5.1 Interface objects

Interface objects, such as iterators present a problem with the message send rules. The interface objects will be created by objects such as lists, but the object owning the list can't send unrestricted messages to them unless it owns them, and if ownership of the interface object is transferred to the list's owner, then the iterator can't access the list's mutable state (figure 5.1).

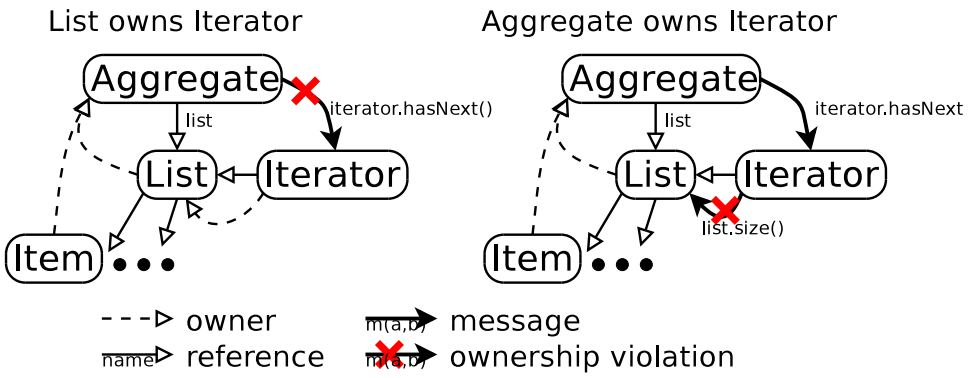

Figure 5.1: The interface problem

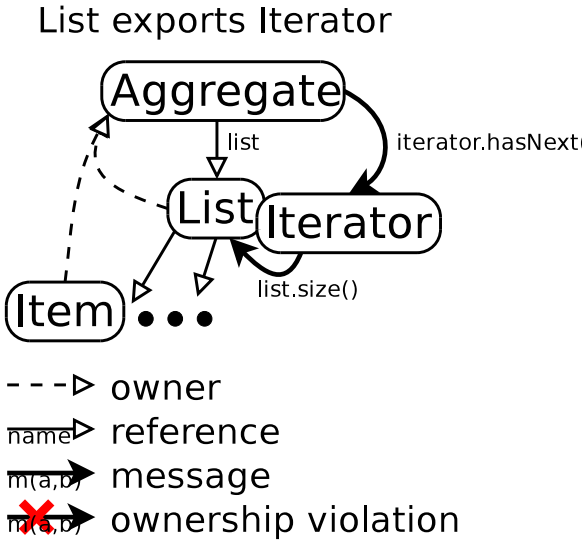

Figure 5.2: Export: the interface solution

Dealing with interface objects such as iterators requires some sort of export operation: a way for the iterator object to send unrestricted messages to the list, and be sent unrestricted messages by at minimum the list's owner.

We have augmented the Dynamic Ownership model to provide such an export operation. When an object (such as a list) exports another object (such as an iterator), the exported object no longer has a separate ownership identity. The exported object occupies the same location in the ownership tree as its former owner; any objects it owned are now effectively also owned by the object which exported it; the ownership system treats it as the same object as all the other objects sharing that location in the ownership tree.

Adding an export operation means that objects no longer have a simple owner field; instead they are given an ownership context field. This field contains a pointer to an *ownership context* object, that in turn contains a pointer to some other ownership context. So the object nodes in the ownership graph of a running program are replaced with ownership context nodes, each of which may have one or more objects associated with it.

When one object exports another, the exported object ends up sharing the exporting object's ownership context. The ownership structure and the corresponding rules for deciding upon a message type in section 4.2 require some changes to handle these ownership contexts. The helper methods on objects used in that section are changed to support these new context objects:

```
is(other) { // notion of identity changes
  return this.context == other.context;
}

owns(other) {
  return other.context.owner == this.context;
}
```

```
contains(other) {
  if (other.is(this)) return true;
  if (other == null) return false;
  return this.contains(other.context.owner);
}
```

In the eyes of the ownership system, the exported and exporting objects are now the same, as they share the same ownership context.

An object cannot be un-exported; once it has been exported, it is inextricably linked to the object that exported it. Exporting is a transitive relation. If some object a owns object b, and object b has already exported object c, then when a exports b, all three objects will share the same ownership context. Therefore, they will occupy the same place in the ownership tree where object a originally was.

This neatly solves the interface problem. An exported object will have the same rights to access its exporter and the objects it contains as the exporter itself has. Other objects in the system will have the same rights to access it as they have to access its exporter. For example, in figure 5.2, the aggregate object is able to send unrestricted messages to the iterator, and the iterator is able to send unrestricted messages to both the list and the items in the list. In effect, the iterator becomes part of the interface of the list object – the list's owner can mutate the list through either the list or the iterator.

5.1.1 Encapsulation Guarantees

Exporting doesn't break the guarantees made by the ownership system. While it changes how an object can be owned, it still maintains the tree structure, associated rules and guarantees as described in section 4.5. The effect of the change is to allow several objects to form the encapsulation boundary to a set of objects they jointly own, rather than a single object

forming this encapsulation boundary. Calls between these boundary objects and from one of them to any of the objects they jointly own are classified as internal calls.

However, exporting should be used sparingly, as it creates a set of related objects between which all messages are classed as *internal calls*. If every object in a program was exported, then every object would occupy the same position in the ownership tree, all messages sent between them would be classed as internal calls, and no encapsulation guarantees would be provided at all.

5.2 Lending – The equals problem

To compare two objects in Java for equality, the program send an *equals* message to one of them, passing a reference to the other object as a parameter. Usually, both of these objects will be owned by another object sending the equals message. This means that neither of the objects being compared owns the other, and therefore they can only send non-argument dependent messages to each other. This means that neither of the objects can access the other's mutable state, and therefore cannot compare any of that mutable state with their own. Figure 5.3 demonstrates the problem.

Several attempts were made at solving this problem. These are discussed below, then section 5.3 presents the final design we chose for ConstrainedJava in detail, and argues that it does not affect the invariants.

5.2.1 Dynamically scoped object access

The first solution we tried was to add one extra rule to the message send rules. If an unrestricted method tries to call another unrestricted method, and the message send rules would normally disallow it, then the call stack is checked to see if the call would have been allowed if the sender was one of the objects on the call stack. If so, then the message send is allowed.

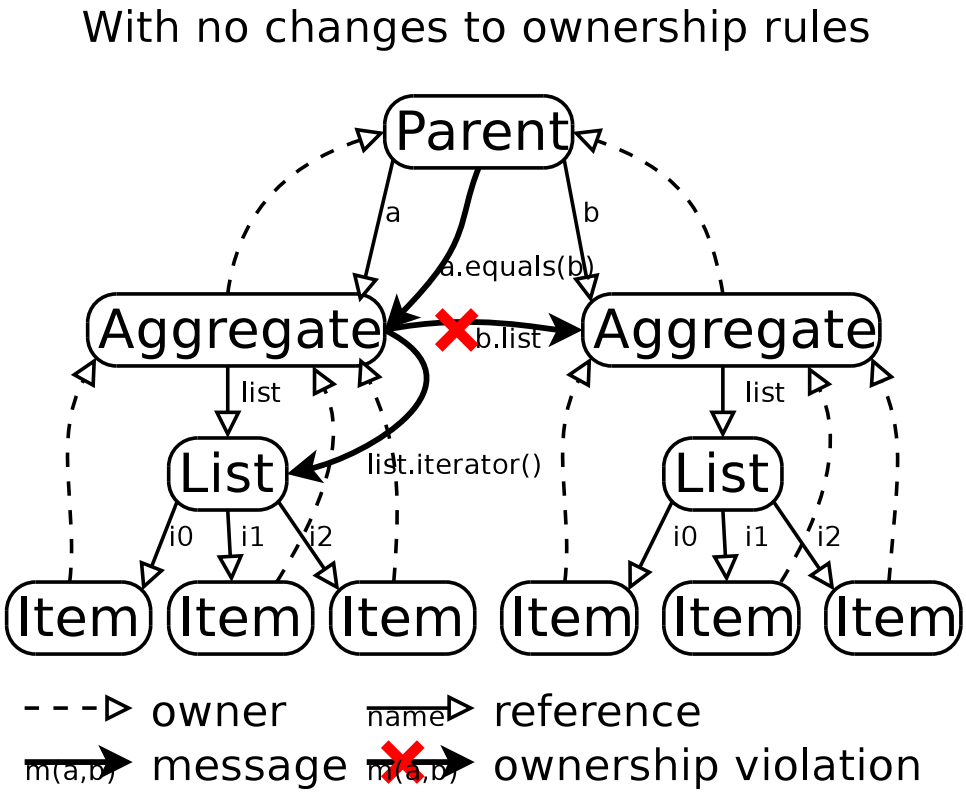

Figure 5.3: The Equals Problem

This solution has the disadvantage that it effectively circumvents the rules about argument-dependant method calls. Calls between objects in a ConstrainedJava program typically start some distance up the ownership tree and then make their way down one level at a time. Therefore, if this solution was implemented, full mutable access to all those objects (and therefore, their immediate children in the ownership tree) would be allowed, effectively allowing unrestricted internal calls to be made to many objects that would otherwise only be able to be sent external calls. All message sends would be in effect carrying with them a grant of all the access permissions of the caller to the message recipient. This would greatly reduce the guarantees provided by the encapsulation enforcement system.

5.2.2 Temporary ownership transfer

Our second attempt at solving this problem (see figure 5.4) was to add a modifier to method parameters, *permit*, which would make the object passed in the marked parameter temporarily owned by the object that method was a member of. This ownership transfer has very limited scope; it's only allowed if the message sender owns the permit parameter, and is only visible within the thread in which the method call occurs, for the duration of the execution of that method.

```
boolean equals(permit o) { ... }
```

In order to avoid surprises for programmers, arguments in the method call must also be marked with the permit keyword.

```
if (o1.equals(permit o2)) { ... }
```

While an argument is marked as permit, all code executed in the thread until the method with the permit argument returns will treat the permit

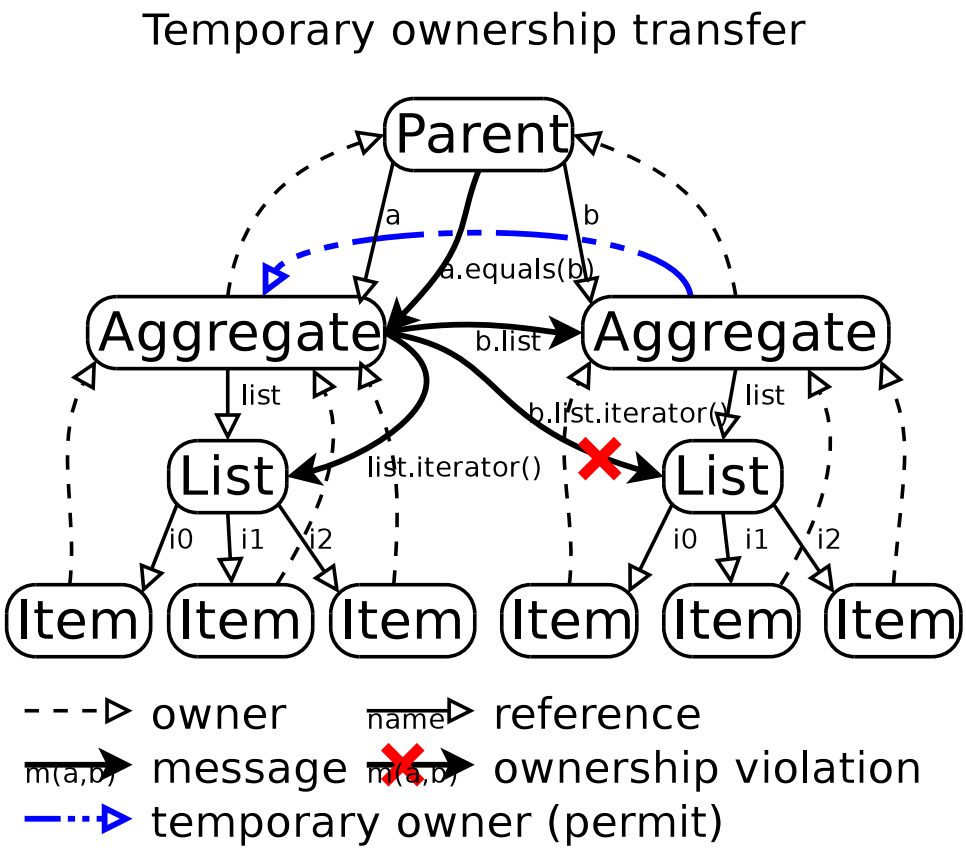

Figure 5.4: The Equals Problem: permit

object's owner as being the permitting object. This change is not visible to other threads, and normal ownership rules still apply to them.

The restricted scope of temporary ownership transfer solves the problem of gaining unintended access to higher parts of the ownership tree, but only works for relatively simple comparisons. In particular, it does not allow sufficient access for comparisons of larger aggregate objects.

In the following equals method, an aggregate object compares items in a list with items in an equivalent list in the comparee.

```
boolean equals(permit o) {
  i0 = list.iterator();
  i1 = o.list.iterator();
  while(i0.hasNext()) {
    if (!i1.hasNext()) return false;
    if (!i0.next().equals(i1.next)) return false;
  }
  if (i1.hasNext()) return false;
  return true;
}

...
  // in an object owning both a and b
  a.equals(permit b);
```

Since the owner of a also owns b, the temporary ownership change is allowed. This only allows a to access the external interface of b. For instance, the call to o.list.iterator() would not be allowed, as according to the ownership rules, iterator() causes changes to mutable state, and *list.owner != a*. So permit does not give sufficient access to allow the comparison of these aggregate objects.

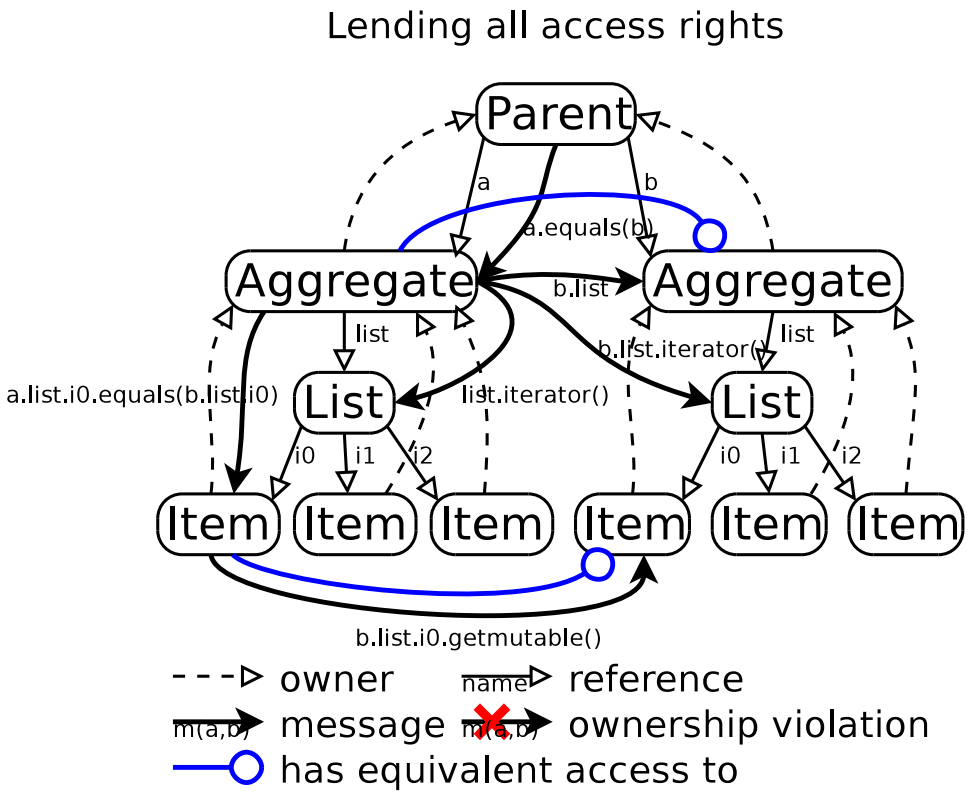

Figure 5.5: The Equals Problem: equivalent

5.2.3 Lending all access rights

Our third attempt to solve this problem provides a better solution.

This approach involves granting access to parameter objects equivalent to that acquired when an object is exported (see figure 5.5). Method parameters marked *equivalent* are added to an access control list associated with the object and method call to which this access is granted (which we call the *accessing* method). Then, when a message is sent from the method called with the equivalent parameter, to the object in that parameter, if the send would have been unsuccessful under the normal ownership rules it is checked again, this time treating the object marked equivalent as the sender.

Granting equivalence with an object a is only allowed in certain circumstances: if the granting object is a, if it owns a, or if it has been granted equivalence with a. The access granted is only usable by the object it was granted to; objects sent messages by the object to which access is granted do not automatically receive the same equivalence rights.

For example, to call equals() on an object granting it equivalence with the object which the comparison is taking place with:

```
boolean equals(equivalent x) {
 ...
}

...
  a.equals(equivalent b);
```

Unfortunately, equivalence as stated allows an object o, which owns objects a and b, to grant an access equivalent to that which b has – access to parts of b which o did not have access to. This breaks the encapsulation invariants: it violates the representation encapsulation of object b.

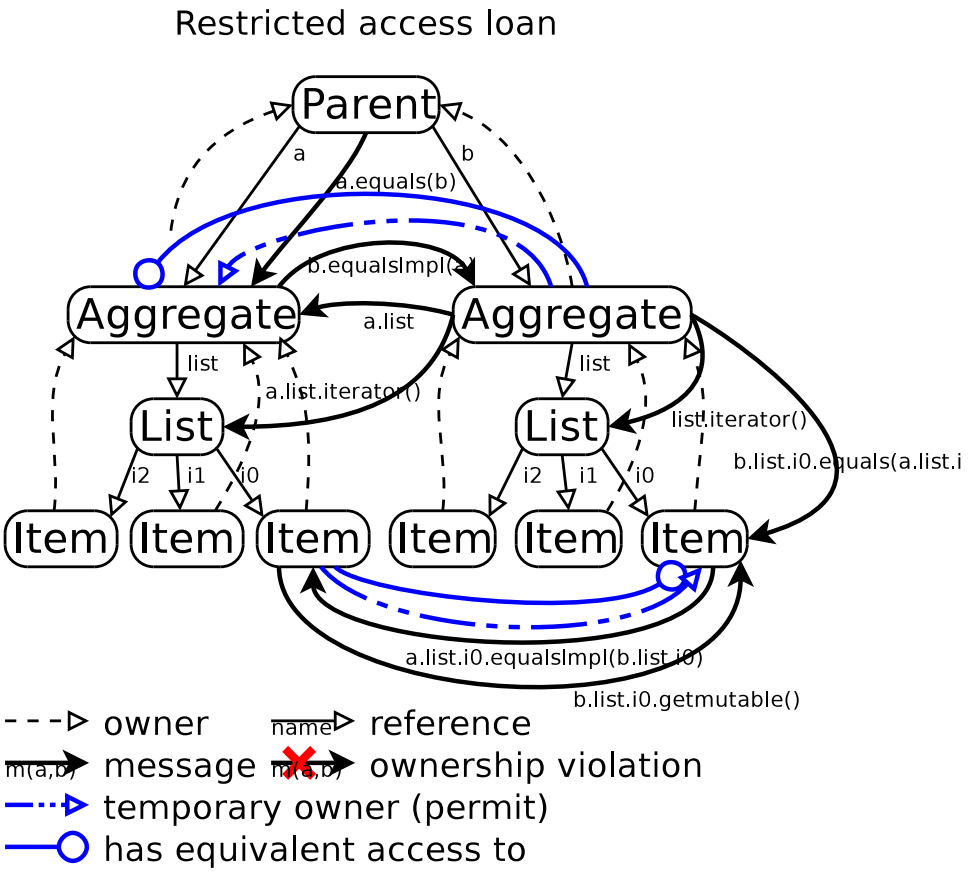

Figure 5.6: The Equals Problem: restricted equivalent and permit

5.2.4 Restricted access loan

The final solution to the equals problem implemented by ConstrainedJava combines the permit and equivalent solutions to offer the best of both worlds: full support for comparing complex aggregate objects, and only carefully constrained local bypassing of the encapsulation invariants (see figure 5.6).

This solution does not use the original form of the equivalent keyword defined in section 5.2.3. It uses a modified version with tighter restrictions on which objects may pass an object as equivalent. An object a calling a

method on an object b, and passing in a parameter p marked as equivalent may only do so if $a == p$. In other words, the only object able to permit other objects access to its representation is itself. This corresponds to the final definition of permit and equivalent in section 5.3.

The equals method must become more complicated to support this restricted access loan. Now two calls are involved, one to a first equals method granting permit access to the single parameter, and then another call to a method on that parameter object to actually perform the comparison, granting full *equivalent* access to the first object.

```
boolean equals(permit x) {
 ... // optional check that x is
     // an object we don't mind
     // accessing our internals
 return x.equalsImpl(equivalent this);
}
boolean equalsImpl(equivalent x) {
 ...
}

...
  a.equals(permit b);
```

5.3 Extensions for lending ownership

We now define the language extensions that enable message parameters to be accessed, by lending object ownership.

ConstrainedJava provides two ways of lending ownership: one for when object a wishes to provide access permission identical to that gained by owning object b *(permit)*, and one for when object a wishes to provide access equivalent to being inside object a *(equivalent)*.

5.3.1 Permit

The *permit* keyword, applied to a method parameter, indicates that the object on which the method is called should have access to the object passed as a permit parameter equivalent to the calling object. This access is limited in scope; it is only valid between the method call and its return, and only usable within the thread that made the call. Furthermore, it may only be granted by the object owning the parameter marked permit. The permit keyword must be specified both in the call and the parameter prototype in the method the call is to:

```
class a {
  var b;
  // ...
    c = new HashGenerator();
    c.makeHash(permit b);
  // ...
}

class HashGenerator {
  makeHash(permit o) {
    return md5String(o.toString());
  }
  //...
}
```

Support for permit is added to the message send rules by keeping track of extra objects that have temporary ownership for the ownership contexts passed as permit parameters, and changing the owns() method used by the message send rules:

```
owns(other) {
  if (other.context.owner == this.context)
```

```
    return true;
  return other.context.permitted.contains(this.context);
}
```

The facility provided by permit is similar to the ability to borrow an object provided by Alias Burying [6], which uses intraprocedural static analysis to determine that a borrowed parameter to a procedure is no more aliased when the procedure returns than when it was called. By comparison, permit causes the object marked permit to be temporarily accessible by the method to which it is passed, and revokes this access when the method call has completed.

5.3.2 Equivalent

The *equivalent* keyword is used in a similar manner to the permit facility. The difference is in the access that it provides. Equivalent allows the called object access to the parameter object equivalent to being that object; thus, its internals and owned objects can be freely manipulated. An ownership error is generated if the parameter object is not the same as the object sending the message; this ensures that the sending object is not able to grant access that it does not already possess. Like permit, equivalent status only applies for the duration of the method call, in the thread in which the call was made.

```
class Course {
  // ...
    c = new MarkAuditor();
    c.auditMarks(equivalent this);
  // ...
}

class MarkAuditor {
  // ...
```

```
    auditMarks(equivalent course) {
      course.getStudents().forEach(
        block(s) { checkMark(s, course.getMark(s)) } );
    }
    checkMark(student, mark) {
      // still has equivalent access
    }
    //...
  }
```

To add the necessary checks, ownership contexts must contain a list of objects they are equivalent to in the current thread, and the is() method used by the message send rules needs changing:

```
is(other) { // notion of identity changes
  if (this.context == other.context)
    return true;
  for(ctx : this.equivalentTo)
    if (ctx.is(other)) return true;
  return false;
}
```

5.3.3 Encapsulation Guarantees

The permit and equivalent facilities provided by ConstrainedJava both allow limited bypassing of the encapsulation invariants. However, this bypassing is explicitly extremely limited in scope. The accessing method is only granted that access for a short period of time – while it is running. So the accessing method is effectively only allowed to see a snapshot of the external object's mutable state. This is information that could be extracted by the accessing method's caller, stored in an immutable object, and passed to the accessing method; the *equivalent* and *permit* features merely provide a convenient way to bypass this costly step.

5.4 Summary

To recap, Dynamic Ownership enforces encapsulation by providing two guarantees: representation encapsulation and external independence.

Representation encapsulation disallows direct access to an object's *representation* – the objects that comprise an object's mutable state can only be accessed by the object owning them; other parts of the system must use that owner object's interface to indirectly access them.

Dynamic Ownership considers an object's representation to be the objects it *owns*. Objects which are part of a container but are not owned by it do not count as part of the representation of the container: only of the object that owns them.

Dynamic Ownership classifies methods that can access and return information about mutable state as unrestricted. It classifies calls from an aggregate to itself, or to its representation as internal calls. Internal calls to unrestricted methods are allowed: they are from an object's interface or itself to itself; they're not to an external object, and are bypassing the object's interface. Therefore they are allowed, and non-internal calls to unrestricted methods are not allowed. In addition, calls to externally independent methods are only allowed if the receiver is visible to the sender – so the representation encapsulation of the receiver's owner is not violated, as the call does not bypass the receiver's owner. As this covers all the possible method types, these restrictions are sufficient to maintain representation encapsulation.

External independence requires that an object is unable to send messages that return information about mutable state to objects it has a reference to but does not own. This is the other side of representation encapsulation; a message sent to an object not owned by the message sender must be bypassing that object's interface, and is therefore breaking representation encapsulation. Therefore, as we preserve representation encapsulation we must also be preserving external independence.

Exporting does not change this in any significant sense, as an exported object effectively merges with the object that exported it in the eyes of the ownership system; calls between it and the object that exported it are classed as internal calls, and its children and the exporting object's children are merged into one set.

Lending ownership does cause external dependence and allows the bypassing of representation encapsulation, but it does so within a very limited scope, in a small, localised part of the object graph. Furthermore, this only occurs when calls have gone through the interfaces of both objects involved, and they have both consented to this dependence and representation access – and it finishes when the first of those calls returns.

Chapter 6

Encapsulation Enforcement Evaluation

In this chapter, we discuss issues relating to writing programs within the restrictions imposed by the encapsulation enforcement, including ConstrainedJava's approach to several common Java patterns, and more general design patterns for object oriented code. Also discussed is the implementation of dynamic ownership and encapsulation enforcement within ConstrainedJava.

6.1 Object Oriented Patterns in ConstrainedJava

We describe how a number of common design patterns [12] interact with the object ownership and encapsulation enforcement provided by ConstrainedJava.

6.1.1 Proxy

Proxy objects act as stand-ins for some other object, providing access to its facilities while imposing extra requirements, like not instantiating the

proxied object until it is required, or disallowing access to some of its methods.

A proxy object must own or export the object it is providing access to, if it needs to call unrestricted methods within the object. If the proxy object does own the proxied object, without exporting it, objects not contained by the proxy will be unable to send any messages to the object the proxy is encapsulating.

6.1.2 Iterator

An iterator is similar to a proxy in that it provides access to some other object, usually a collection. But unlike a proxy, it must do this without usurping ownership of that object. Collections may have several iterators in use at once, and only one of them would be allowed to own the collection.

The solution to this problem provided by our ownership system is the export facility, which allows the iterator to become part of the collection's public interface, able to send and receive messages as if it were the collection object itself.

6.1.3 Visitor

A Visitor is an object that performs some operation when passed an object. Visitors are themselves typically passed to some aggregate object. This aggregate causes some method on the visitor to be called for each of some set of other objects; maybe by those objects themselves – for example, the nodes in a graph.

The problem here concerns the nature of the calls back to the visitor object. As the visitor is likely to be owned by the owner of the aggregate objects whose elements are being visited, calls to externally independent methods could be made to the visitor object. In this case, however, the visitor would have no ability to make calls to the elements at all, and only externally independent calls to the aggregate that owns the elements.

If the visitor called back, by another oneway method, the object that owns it and the aggregate being visited, that would allow this owning object to send messages to the aggregate, manipulating the element. Additionally, if the visitor object was merely a closure that was part of the object owning the aggregate, then a call from the visitor back to the aggregate would not be necessary. Calling the aggregate to manipulate the objects it contains is very similar to the style of programming used when conforming to the Law of Demeter.

6.1.4 Composite

Composite has no special problems with the encapsulation system – the digraph of references from composites to leaves and other composites would be nicely mirrored by the owner pointers going in the opposite direction. However, if it is used with the visitor pattern, the restrictions mentioned in section 6.1.3 would apply.

6.1.5 Factory Method

A factory method instantiates an object on behalf of some calling object, often using runtime information to make decisions about the type of object to create.

Factory methods typically change mutable state and return an object; therefore, they count as normal methods and can only be called from the object they are part of, or that object's owner. Dynamic Ownership provides the factory method modifier (see section 3.3.2) which transfers the ownership of the object returned by the factory method to the calling object.

6.1.6 Singleton

The singleton pattern ensures that a program only ever creates one instance of a particular class.

While this is not in itself a problem for the ownership system, it does influence what a singleton class can do. If the singleton is to be accessed directly by several objects which do not contain each other, then many objects in the system will only be able to call externally independent methods on the singleton object, as they do not contain the singleton.

In some cases this is not a problem. Operations such as sending messages to be logged, or other output, can be performed with oneway methods, which are externally independent. But maintaining a cache accessible by the whole program requires unrestricted methods to be called. ConstrainedJava does not currently provide a mechanism to support this.

6.1.7 Observer

The observer pattern does not pose any particular problems to our ownership system. While the observer is *visible* to the observed object, oneway "I've changed" messages can be sent back to the observer. The message that subscribed the observer to updates would have to be sent by the observed object's owner, but this can be passed down the ownership tree, law-of-demeter style. An alternative to the observer pattern is to use the constraint system described in chapter 7.

6.2 Language features

A number of language features need special consideration in how they interact with the ownership and encapsulation enforcement systems.

6.2.1 Labelling Method Types

ConstrainedJava requires externally independent methods to be labelled with a *pure* or *oneway* method modifier. For example, the add method in a list might be marked *oneway:*

```
class List { // ...
  public oneway add(o) {
  //...
  }
}
```

Checks are performed at runtime to ensure a *pure* or *oneway* method conforms to the constraints implied by that label; if a pure method accesses mutable state, an OwnershipException is thrown, and anything returned by a oneway method is eaten by the interpreter and not passed through to that method's caller. Therefore, it would be possible to partially or completely forego the labelling requirement altogether and rely on the runtime checks to ensure that the method either does not access mutable state or does not return any result. Defaulting to assuming *pure* mode if the message send rules require a method to be externally independent would be one way of doing this.

Not requiring such labelling turns out to be a bad idea, however. It makes these methods pure or oneway polymorphic – in some situation they are restricted, and in others they are not. Experience with using ConstrainedJava to write programs shows that it is very easy to write methods marked *pure* that access mutable state by mistake. When they're not labelled as pure, ownership errors in an unlabelled method are less obviously the cause of a problem: it is not forced to run in pure mode unless the message send rules require it to. As most messages are classified as internal calls (see section 6.4), problems with code in such methods will only show up intermittently, when they are called by some other object.

Oneway mode will not cause ownership errors to be thrown – returned values or exceptions are eaten by the interpreter when the method returns, and an error message is printed to warn the programmer that a oneway message attempted to return a value or throw an exception. Methods declared as pure, however, will cause ownership errors to be thrown when non-final fields are accessed, or non-externally independent methods are called. For example, in the following class there are two pure methods, one which accesses a non-final field, which is disallowed, and one which delegates this to an externally independent oneway method, which is allowed:

```
class Foo {

  final f;
  var m;

  public pure doStuffBad() {
    m++;       // disallowed
    i = f + 4; // ok
    return i;  // ok
  }

  public pure doStuffGood() {
    incM();    // ok
    i = f + 4; // ok
    return i;  // ok
  }

  private oneway incM() {
    m++;
  }
}
```

6.2.2 Closures

ConstrainedJava's blocks (section 3.2.4) provide the same facilities as that provided by Smalltalk's blocks. They allow an object to be created with a single value() method, which runs in the scope in which the block was defined.

```
a(b) {

  b.visitNodes(block oneway (x) {
      if (x.equals("")) { return false; }
    });
  return true;
}
```

ConstrainedJava uses exporting (section 5.1) to deal with closures. A closure never has a separate ownership identity – it inherits the ownership context of the object that created it, effectively being exported by its creator. Thus, closures maintain access to the object that created them, and messages sent to them have the same restrictions imposed on them that messages sent to their creator object have.

Some common uses of blocks are allowed by the message send rules; some others, in particular, many Smalltalk-style control structures do not.

Using closures with a forEach method on a collection to iterate through that collection requires the closure to be marked oneway. The collection's call back to the closure counts as an external call, and is thus allowed. For example, in the following code snippet (where block() {..} declares a closure in ConstrainedJava):

```
var sumlist(list) {
  sum = 0;
  list.forEach( block oneway (x) { sum += x } );
  return sum;
}
```

In this example, when the object representing the closure block oneway (x) {...} is created, ConstrainedJava automatically causes it to exported by its owner – effectively, for ownership purposes, it becomes a part of the object it was declared in.

Emulating more Smalltalk-style control structures is harder. Implementing a Smalltalk-style if using blocks and ifTrue and ifFalse methods on singleton True and False objects does not, however, work. While an ifFalse message can be sent to the singleton boolean object, if that object is True, then it cannot send a message back to the closure passed to it, as the closure would not be visible to the shared singleton True object, which would be owned by the root object.

6.2.3 Inner Classes

Inner classes present a problem with our system as currently specified, as they act as a special object with unrestricted access to their source objects. Exporting them would give them this access, but then they wouldn't have a separate ownership identity.

Exporting does make sense when an inner class is being used in a manner analogous to a closure, a common use for anonymous inner classes in Java. Exporting is in fact the way that ConstrainedJava deals with closures. But exporting conflates the concepts of allowing other parts of the system to access an object with giving the exported object privileged access to the internals of its outer object.

Ownership Types for Object Encapsulation [5] (see section 2.2.4) deals with this problem by allowing objects, when instantiated, to be passed extra ownership information. This then enables the instantiation of inner classes that are owned by some other object, but have an extra ownership parameter allowing them access to the internals of the outer object that created them. Constraints can be set forcing the owner of the inner class to be the owner of the outer object, or some object owned by that owner.

The current implementation of ConstrainedJava does not support inner classes.

6.2.4 Calling Java

ConstrainedJava inherits from BeanShell the ability to call native Java code. Java objects can be instantiated in the same way as ConstrainedJava objects.

As ConstrainedJava calls native Java methods using reflection, it has no control over the behaviour of native Java code. Native Java objects created by ConstrainedJava code are wrapped to give them owner pointers, but these wrappers can be lost when Java code deals with native objects. It is therefore recommended to avoid using Java native objects if possible, or compartmentalise their use to a small part of your program.

ConstrainedJava also provides the ability to cast objects to a native Java interface type, to allow native Java code to call ConstrainedJava code back.

```
import java.awt.*;
  ...
  var x = new Window();
  x.addActionListener((java.awt.event.ActionListener)this);
```

6.3 Implementation

The ConstrainedJava language is a modified version of the BeanShell Java source interpreter, with extensions to support our Dynamic Ownership system.

To add Dynamic Ownership to the language, a number of changes have been made.

ConstrainedJava tracks the ownership of each object, by adding an owner pointer. Ownership tracking is complicated by the need to allow for the export operation, which causes two or more objects to share the same location in the ownership tree. This precludes the use of a simple

owner field. ConstrainedJava implements ownership tracking by giving each object or set of exported objects an Ownership Context, which stores the objects' owner pointer. This scheme is described in more detail in section 5.1.

ConstrainedJava provides two mechanisms for specifying ownership change: as a message or as a method modifier. A *gift* message can be sent to an object, with the object's new owner as a parameter. Alternatively, a method can be marked with the modifier *factory*, which causes the ownership of any object returned to be transferred to the calling object.

Oneway and pure methods are marked as such by being declared with the oneway and pure modifiers, for the reasons discussed in section 6.2.1.

Checks have been added to facilitate message restrictions. Every method call is checked to ensure that the message send restrictions are not violated, traversing the ownership tree as necessary.

In addition, methods which have been marked pure must not be allowed to access mutable state. This is implemented by setting a pure-mode flag for the current thread, and checking field accesses when it's on to ensure that only final fields are accessed.

Oneway methods are accommodated by forcing them to return void, and ignoring any exceptions they throw.

Permit and equivalent are implemented by adding lists of ownership contexts whose objects have permit or equivalent access to the ownership context the lists are associated with.

6.4 Performance

ConstrainedJava was instrumented to record information about the performance of the ownership system while running a number of demonstration and benchmark programs.

The major overhead added by the ConstrainedJava system is that every field access and message send requires the ownership system to check what

restrictions may apply to it.

Most (77%) of messages (this includes field accesses) are sent to an object directly owned by the message sender. Only 22% are flagged as being restricted to being *pure* or *oneway*; the remaining 1.2% are calls within an object.

These statistics were gathered running a number of GUI demos, several constraint-based, totalling 1700 lines of code.

Checking if a message sender *a* directly owns the receiver *b* is a relatively cheap operation; it requires checking that *b.owner* == *a* – that the receiver's owner pointer points to the sender. This operation becomes more expensive if the sender, *a*, is marked *equivalent* to some object, as the list of objects which *a* may gain access to the children of must also be checked through.

The test to see if an object's owner is equal to some object b (*a.owner* == *b*) is normally quite cheap. However, if object *b* has been marked equivalent to some other object *c*, then object *b* acquires a per-thread equivalence set that must be checked when the *a.owner* == *b* test is made and fails. This involves cycling through the members of the equivalence set and making comparisons like *a.owner* == *c*.

Instrumentation of the *owns* check shows, however, that of the calls made to it, only 2.4% of them require iterating through the equivalence set, and only 0.14% of them end up iterating through the equivalence set without finding a match. So the overhead added by the *equivalent* check does not appear to be great, and this is in a program that uses constraints – and thus, calls to *equals()* with a parameter marked *equivalent* – heavily.

A simple 70 line program (listed in appendix B) to perform inserts into a sorted list was benchmarked on a 2.8GHz Pentium 4 running NetBSD and Sun's Java 1.5.0 HotSpot JVM, with both the original Beanshell 1.3.0 on which the ConstrainedJava prototype implementation was based, the new Beanshell 2.0b4 from which some changes were taken, and the Constrained-Java prototype implementation, both with encapsulation enforcement on and turned off. Averages are over the last 40 of a loop of 47 runs, with an

inner loop that performs 4000 inserts.

Interpreter	Minimum time/run (ms)	Average time/run (ms)
BeanShell 1.3.0	259	267.35
BeanShell 2.0b4	3566	3616.975
ConstrainedJava (no EE)	367	379.925
ConstrainedJava (EE)	389	403.575

Table 6.1: Performance

The data (table 6.1) shows that maintaining the ownership information, which is still done even with encapsulation enforcement turned off, causes the program to run 41% slower. However, adding message send checks only reduces performance by a further 6%.

The reason for the much slower BeanShell 2.0b4 result is the new structure for handling BeanShell-generated objects. This is not a change carried over to ConstrainedJava, due to the extreme performance penalty it exacts.

Chapter 7

Constraints

In this chapter, ConstrainedJava's one-way constraint system is described.

7.1 Introduction

Enforcing encapsulation can make programming harder. Common techniques, like using the observer pattern (section 6.1.7) become more complex. Propagating changes in state from one part of the program to another requires some object which contains both objects between which state is being transferred, along with explicit support in that object for shepherding those state changes between different parts of the system.

A constraint system allows programmers to make explicit relationships between the state of parts of a program that would have otherwise been coded implicitly, updating one field when another changed.

Constraint systems are relevant to our ownership system, as the ownership restrictions attempt to ensure that objects do not become dependent on the state of internal parts of other objects which are external to them. Constraint systems can allow different parts of the program to depend on each other in a more controlled and explicit fashion.

ConstrainedJava provides a one-way constraint system. One-way constraint systems are only able to propagate updates in one direction, and are

thus much simpler to implement than multi-way constraint systems. For example, the constraint

$$a.x = b.x + 40$$

would cause $a.x$ to be updated when $b.x$'s value changed, but not the other way around. The constraint has a direction, an identifiable source and sink.

Most one-way constraint systems specify the sink as a field in an object. The field is updated when the value of the source expression changes. Some mechanism may be provided to notify the object that its field value has changed, allowing for instance a GUI element to be redrawn when the fields describing its state are changed.

Using fields directly as constraint sinks can be undesirable from an encapsulation standpoint, however. If a constraint sink must be specified as a field in an object, then only values exposable as a single field can be used as the target for a constraint. The field must be considered part of the public interface of the object if other objects are able to set up a constraint targetting it. Some mechanism must be provided to allow some action, e.g. redrawing part of the UI, when fields are changed in this manner.

ConstrainedJava's one-way constraint system requires both the source and sink of a constraint to be specified as a block (see section 3.2.4) of code, so any action can be performed to retrieve or set a value – a simple field access, a method call, or something more complex.

7.2 Constraint Lifecyle

One-way constraints have three parts: a source, a sink and a control object. ConstrainedJava provides a mechanism to specify the source and sink as blocks, and associate them with each other, to form a constraint, generating a control object in the process.

When the constraint is established, ConstrainedJava gathers the set of fields which the constraint's source depends on, by executing the source.

When writes are detected to these fields, the source block is re-evaluated. If the source block's return value has changed, then the sink block is called. A scheduling mechanism is used to attempt to reduce the number of unnecessary evaluations.

7.2.1 Establishing a constraint

A constraint in ConstrainedJava is comprised of a pair of blocks, representing the source and sink of the constraint, and a control object. ConstrainedJava expects the source block to take no parameters and return some value. This value is what is passed onto the sink block. There are no extra restrictions, other than encapsulation enforcement, applied to a constraint source block.

While there is no restriction on the source block having side-effects, these are not recommended.

As an example, a source block to monitor the position of a slider could be as follows:

```
...
source = block () {
  slider.getValue();
};
...
```

ConstrainedJava expects the sink of a constraint, like the source, to be specified as a block. The block must take one parameter, and its return value is ignored.

Using a block as the sink of a constraint instead of a reference to a field has the advantage that more complex data types produced by the sink can be dealt with by the sink. Also, as methods can be called from a block, the

object being changed as a result of the constraint has the chance to detect that it has been changed. This means that an extra mechanism to deal with detecting changes in order to propagate them through the system, for instance by redrawing GUI widgets, is not required.

The source block is, until the constraint is created, just an ordinary block. To form a constraint, a sink block must be attached to it.

```
...
sink = block (x) {
  VolumeControl.setVolume(x);
};
constraint = source.listen(sink);
...
```

7.2.2 Gathering dependencies

When a constraint is created or updated, the source block is executed, and field accesses are monitored in order to ascertain which fields it reads, and thus what parts of the system its output depends on.

The list of fields read is then passed to the part of the system that detects changes that might require constraint re-evaluation.

Not every field read is part of this list. Final fields are ignored, as their value cannot change. Field access monitoring is suspended when an external call (either one-way or pure) is made, as neither can provide the source block with any information about the program's mutable state.

7.2.3 Detecting changes

During normal code execution, when a field is written to, a check is made to see if the field is one that a constraint depends upon. This monitoring of field writes is not required when a pure method is executing, as such methods are unable to write to fields.

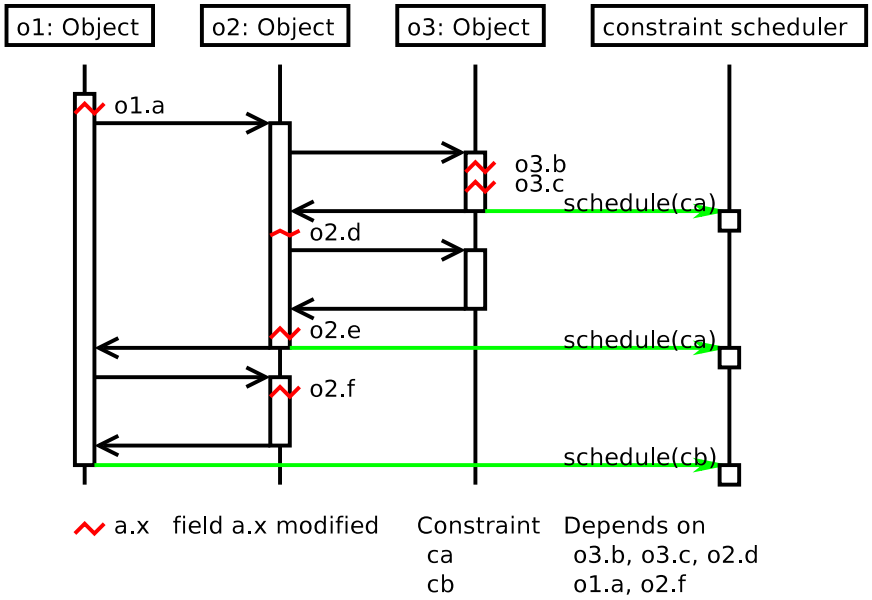

Figure 7.1: Detecting changes: reducing scheduler calls

If a constraint depends on a field that is written to by the program, and the value written is different to that field's previous value, then the constraint must be marked for re-evaluation.

ConstrainedJava attempts to batch multiple updates together, so that a constraint is not re-evaluated part of the way through an update of the data it depends on. This improves performance, but assumes that a method that modified a monitored object will finish execution relatively quickly. This tends to be true for code executed in the UI thread of a Java AWT or Swing based application, but when it is not true, constraint semantics would be affected. So far we have found this tradeoff to be acceptable.

Batching the updates together is achieved by storing a setting in the callstack when a particular constraint's dependent field is modified. When a method returns, it checks if there are any such flags set for the current callstack level. For each candidate constraint it finds, it checks if this constraint has been invalidated by a method call in the callstack that has not yet completed. If no method in the current callstack has invalidated the

constraint, then it is scheduled for re-evaluation.

This means that once a method invalidates a constraint, it is only scheduled for re-evaluation when that method returns. If that method calls other methods that would also invalidate the constraint, it is still only scheduled for re-evaluation when the first method to invalidate the constraint returns.

For example, in figure 7.1, constraint `ca` depends on the fields `o3.b`, `o3.c` and `o2.d`. When o3.b and o3.c are modified, no other method has yet modified a field depended on by `ca`, so the constraint is scheduled for re-evaluation when the method that modified them returns.

However, as the first method on `o1` modifies `o1.a` before calling the method that modifies `o2.f`, constraint `cb` (which depends on both fields) is only scheduled when the first method to invalidate it returns.

7.2.4 Scheduling re-evaluation

As ConstrainedJava evaluates constraints eagerly, some effort must be made to reduce unnecessary re-evaluations. ConstrainedJava uses two methods to achieve this: only marking constraints for re-evaluation during a method return (as described in the previous section), and the way it schedules constraint evaluation.

When a constraint is scheduled for evaluation, it is added to a queue. An AWT event is used to process constraints from the queue. When one or more constraints are available to be evaluated, an AWT event is scheduled. This executes in the AWT thread used by the Java AWT and Swing user interface frameworks, between user interface events. It evaluates all pending constraints, including any which are scheduled for re-evaluation while the event is executing.

7.2.5 Constraint Evaluation

Constraints are evaluated by executing the source block, gathering dependencies while doing so, and then comparing its return value to that

returned by the previous run of the source block. If the value returned has changed, then the sink block is called with the return value of the constraint source block passed in as a parameter.

Detection of a change in the return value of the monitored block is accomplished by comparing it to the previously returned value via an object identity comparison (Java ==) and then calling the new object's equals() method. Therefore, the monitored code must return a different object if a change in its return value is to be noticed.

7.2.6 Activation

A constraint is activated when it is created, but it can be rescinded and reinstated if desired.

This is accomplished by calling the rescind() and reinstate() methods on the constraint object returned by the listen() method used to establish the constraint.

For instance, when the mouse cursor entered a region, a constraint causing some widget in that area to be updated could be reinstated, and rescinded again when the mouse left the region:

```
...
if (region.contains(mousePos))
  widgetMouseConstr.reinstate();
else
  widgetMouseConstr.rescind();
...
```

7.3 Overhead

The constraint system imposes two types of overhead: storage and processing time.

When constraints are not in use, the only noticeable overhead is more storage use, as all the Variable objects store a list of constraints that depend on them. This overhead is not great compared to the existing overhead imposed by BeanShell; a Variable object (used by BeanShell to represent a variable or field) already has ten nonstatic fields, and the constraint system only adds one more.

When constraints are in use, overhead is introduced at several points.

7.3.1 Monitoring

When code is executed to evaluate a constraint, all field accesses are monitored, and accessed fields are added to a list associated with the constraint. Variables which represent fields which are or have been depended on by a constraint also store a list of constraints that depend on them, to facilitate notification when the fields change.

Unfortunately this scheme can create large lists of fields to be monitored. An example of this being particularly problematic is monitoring changes to a value in a linked list. Since n mutable fields in the form of next pointers must be read in order to read the nth element of the list, monitoring a single item in a large list can cause a large number of next pointers to be monitored. If several items in the list are monitored, this number grows very quickly.

Pruning this list of fields to monitor would appear to be desirable, due to the storage costs and possibilities of excessive constraint evaluation. Several strategies have been tried, but the costs of each have proved too great.

In fact, the scheduling improvements detailed in section 7.2.4 have reduced overhead to a level where the existing problem of too much constraint re-evaluation has essentially gone away.

7.4 Discussion

7.4.1 Ownership-directed simplification

It had been hoped that the ownership tree would provide useful extra information with which to optimise the constraint system, but so far this approach has proved fruitless.

One approach to monitor list pruning we tried was to monitor entire objects, instead of fields, and forward changes up the ownership tree. This meant that entire aggregate objects could be monitored rather than the set of sub-objects that were part of them. When an object change notification is signalled, that object's owner also signals a change, and this continues up the ownership tree.

This approach had much greater storage efficiency, as the set of objects read is much smaller than the set of fields read. However, this approach had very poor performance; field writes caused changed messages to be propagated up the ownership tree. Due to the low granularity of monitored areas, many more constraints were re-evaluated than was necessary.

The ability to describe regions in objects, and which of these regions different operations affect, like the system used in FX [13] would improve this situation. Forwarding change notifications up the ownership tree does not fit well with the concept of object regions. As object ownership is an orthogonal property to the fields in which references to an object is held, there is no obvious way to retain region information when a change notification is forwarded up the ownership tree.

In the end, simpler means to reduce overhead have been more successful, in particular the scheduling described in sections 7.2.3 and 7.2.4.

7.4.2 Native code

As the code for native Java objects is executed by the host JVM, not the ConstrainedJava interpreter, field access cannot be monitored. Thus, con-

straints depending on the state of native Java objects will not be evaluated unless this state is manually propagated into ConstrainedJava objects, for example in response to Java AWT events. Helper classes to perform this copying for selected AWT controls have been implemented.

7.4.3 Change detection

Constraint source must return a different object to the one originally returned if the information in it changes. The object returned is compared with the previous object returned using the standard Java equals() method. If the reference returned is to the same object as that returned by the previous call to the constraint source, then comparing them with equals() would be comparing that object with itself, which should always return true.

7.4.4 Interaction with the ownership system

The constraint system must interact with the ownership system as it sends messages to objects in the system: to execute code whose return value is being monitored, equals messages to compare that return value with the previous one, and to call the block(s) that propagate the value to its destination.

ConstrainedJava sends the messages from the ownership context of the object which created the monitored block in the first place. As ConstrainedJava has the ability to share a monitored block between several constraints, this does make some sense. However, it means that whichever object calls the listen method is able to cause code to run in a different ownership context. Our solution is to treat the listen method as an unrestricted method in the same ownership context as the block, which calls the value methods of the source and sink blocks.

7.4.5 Scheduling performance

Evaluating the constraints as a batch between GUI events has the advantage that GUI changes caused by events, such as repaints of parts of the UI, all happen at once. There are no partial updates where some constraints are satisfied but others are not. The GUI feels subjectively smoother than it does when constraints are evaluated in parallel with the user interface thread.

7.5 Summary

Constraints help programming with our encapsulation enforcement system, as they allow objects to depend on the state of another object without violating the external dependence rule: an object owning both the depending and dependant object can initiate the constraint that ties them together.

ConstrainedJava implements a simple one-way constraint system, which allows parts of the running program to be notified when the mutable state of other parts changes, subject to ownership restrictions. This is integrated with the language using closures, and utilises a number of optimisations to improve performance.

Chapter 8

Conclusions

8.1 Summary

In this thesis we have presented ConstrainedJava, an implementation of the Dynamic Alias Protection [25] alias protection system. ConstrainedJava provides much more effective encapsulation facilities than traditional object oriented languages such as Java.

Firstly, the ownership structure utilised by ConstrainedJava was presented. This provides the structural basis on which the encapsulation enforcement system is based.

Then the encapsulation enforcement system, which restricts message sends based on information taken from the ownership structure, was presented. The encapsulation invariants were described. The different method and message classifications were described along with the rules that utilise them. An argument was made that these rules preserved the encapsulation invariants.

The restrictions imposed by Dynamic Alias Protection made writing practical programs hard. Extensions to the DAP model we devised were described that made writing real programs much easier.

We then demonstrated this benefit by describing how several common object-oriented design patterns would be implemented within a Con-

strainedJava program. Interaction with other language features, implementation details and performance information was presented.

The constraint system built for ConstrainedJava was also presented. We described the interface it provides, the implementation optimisations that make it usable for GUI programming, and its interaction with the ownership and encapsulation system.

8.2 Contributions

We have implemented an ownership and encapsulation enforcement system based on the Dynamic Alias Protection proposal [25] inside a Java-like dynamic object-oriented language, BeanShell [24].

The simple encapsulation enforcement and ownership structure of Dynamic Alias Protection was then extended to make it possible to write real programs. Our extensions included support for interface objects and closures (export), the ability to lend other objects the rights conferred by object ownership (permit and equivalent), and the ability to permanently change the owner of an object. These extensions, while easing the job of the programmer, do not violate the encapsulation invariants taken from Dynamic Alias Protection.

We have also implemented a simple one-way constraint system on top of the same language, to simplify the propagation of mutable state between parts of an object-oriented program. This constraint system is integrated with the ownership and encapsulation system provided by the ConstrainedJava language.

8.3 Comparison with previous work

The structure of the ownership and base encapsulation enforcement system detailed in chapters 3 and 4 is based on the concepts of ownership

and the alias protection rules described in the Dynamic Alias Protection proposal [25], which was not previously implemented.

A number of solutions to the problem of iterator objects in existing static alias protection or ownership systems have been proposed. ConstrainedJava's export operation (see section 5.1) bears some similarity with these systems. Boyapati et al [5] provide a scheme wherein an object and instances of inner classes associated with it share the same ownership properties, in the same way we allow a collection and iterator to share an ownership context; however, they provide no way to instantiate inner classes with their own ownership contexts.

Dave Clarke's extensions [8] to Ownership Types [9] does allow arbitrary objects to share representation contexts, which provides a facility much more similar to ConstrainedJava's export facility.

The ability to lend ownership of an object by marking a method parameter as permit (see section 5.3.1) is comparable to borrowing as defined in Alias Burying [6]. Borrowing guarantees that when the method passed an object marked borrow returns, the borrowed object will not be more aliased than it was when the method was called. In ConstrainedJava, when a method passed a permit mode parameter returns, it loses any privileged access granted by the permit mode that it did not already have.

The constraint system provided by ConstrainedJava has some similarity to that provided by Amulet [23]. Like Amulet's system, ConstrainedJava treats the source part of a constraint as an arbitrary block of code whose execution is monitored to determine which slots in the system it depends on. Unlike Amulet's approach, however, ConstrainedJava treats the target of a constraint as another block of code, whereas Amulet requires that constraints are placed in object slots. This has the advantage that any update can be performed by the target code, however it also means that fields updated by that code cannot be guaranteed to be up-to-date when the constraint's dependencies change, a guarantee Amulet provides.

8.4 Future work

A number of directions are available for future work.

We would like to produce a formalism for the ownership and enforcement system provided by ConstrainedJava, to formally prove that the encapsulation invariants from Chapter 4 are maintained.

Another area of work to be explored is finding a natural way to incorporate support for inner classes into the ownership system.

We'd also like to implement the ConstrainedJava ownership and encapsulation enforcement system on top of a more mainstream language than BeanShell.

Appendix A

Message send rules

The full set of message send rules, incorporating the original Dynamic Aliasing Protection rules from chapter 4 and our extensions to them from chapter 5. These are intended to be invoked from interpreter code similar to that provided in the sendMessage() method below, a replica of the code from section 4.2.1.

```
sendMessage(sender, receiver, target, isPure, args) {
  // call from sender object to method target on receiver
  // sender, receiver are OwnershipContexts
  // target is some sort of Method object
  // ...
  if (checkMessageSend(sender, receiver, target, isPure))
    dispatchMessage(sender, receiver, target, args); // ok
  else
    throw new OwnershipException(sender, receiver, target);
  // ...
}

checkMessageSend(sender, receiver, target, isPure) {
  messageType = getMessageType(sender, receiver);
  methodType = target.getMethodType();
```

```
  if (isPure && methodType != EXTERNALLY_INDEPENDENT)
    return false;
  allowed = getAllowedMethodTypes(messageType);
  if (allowed == ALLOW_NONE) return false;
  if (allowed == ALLOW_ALL) return true;
  if (methodType == EXTERNALLY_INDEPENDENT &&
      allowed == ALLOW_EXTERNALLY_INDEPENDENT)
    return true;
  return false;
}

getMessageType(sender, receiver) {
  if (sender.is(receiver) ||
    sender.owns(receiver)) return INTERNAL_CALL;

  if (receiver.visibleTo(sender))
    return EXTERNAL_CALL;

  // else
  return ENCAPSULATION_BREAKING;
}

getAllowedMethodTypes(messageType) {
  if (messageType == INTERNAL_CALL)
    return ALLOW_ALL;

  if (messageType == ENCAPSULATION_BREAKING)
    return ALLOW_NONE;

  if (messageType == EXTERNAL_CALL)
    return ALLOW_EXTERNALLY_INDEPENDENT;
```

```
}

class OwnershipContext {
  var owner; // points to another context
  var objects; // list of object(s) this is context for
  // list of objects permitted to access this one
  var permitted;
  // list of objects this object can impersonate
  var equivalentTo;
  //...

  is(other) {
    if (this == other)
      return true;
    for(ctx : equivalentTo)
      if (ctx == other) return true;
    return false;
  }

  visibleTo(other) { // is this visible to other?
    return owner.contains(other);
  }

  contains(other) {
    if (other.is(this)) return true;
    if (other == null) return false;
    return this.contains(other.owner);
  }

  owns(other) {
    if (this.is(other.owner))
```

```
        return true;
      return other.permitted.contains(this);
    }
    //...
  }
```

Appendix B

Benchmark Code

This is the benchmark code timed to produce table 6.1, demonstrating the performance impact of ConstrainedJava's encapsulation enforcement and ownership system on a simple program.

```
class Node {
    var next = null;
    var value = null;
}

class SortList {

    var head = null;

    add(x) {
        var ptr = head;
        var prev = null;
        var node = new Node();
        node.value = x;
        while (ptr != null && ptr.value < x) {
            prev = ptr; ptr = ptr.next;
        }
```

```
        if (prev == null) {
            node.next = head;
            head = node;
        } else {
            node.next = prev.next;
            prev.next = node;
        }
    }

    av() {
        var sum = 0;
        var num = 0;
        var ptr = head;
        while(ptr != null) {
            sum += ptr.value;
            ptr = ptr.next;
            num++;
        }
        return sum/(1.0*num);
    }

    dev() {
        var av = av();
        var sum = 0;
        var num = 0;
        var ptr = head;
        while(ptr != null) {
            sum += Math.pow(ptr.value-av,2);
            ptr = ptr.next; num++;
        }
        return Math.sqrt(sum/(1.0*num));
```

```
    }

    min() {
        return head.value;
    }
}



min = 99999999;
times = new SortList();
for(int i = 0; i < 47; i++) {
    System.gc();
    now = System.currentTimeMillis();
    l = new SortList();
    for (int j = 4000; j > 0; j--) {
        l.add(j);
    }
    then = System.currentTimeMillis();
    time = then - now;
    if (i>6)times.add(time);
    System.err.print(" "+time);
}
System.err.println("");

System.err.println(" min "+times.min()+" av "+
                   times.av()+" dev "+times.dev());
```

Bibliography

[1] ALMEIDA, P. S. Balloon types: Controlling sharing of state in data types. In *ECOOP Proceedings* (June 1997), pp. 32–59.

[2] ARNOLD, K., GOSLING, J., AND HOLMES, D. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, 2000.

[3] BASILI, V. R., BRIAND, L. C., AND MELO, W. L. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng. 22*, 10 (1996), 751–761.

[4] BORNING, A., AND DUISBERG, R. Constraint-based tools for building user interfaces. *ACM Trans. Graph. 5*, 4 (1986), 345–374.

[5] BOYAPATI, C., LISKOV, B., AND SHRIRA, L. Ownership types for object encapsulation. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2003), ACM Press, pp. 213–223.

[6] BOYLAND, J. Alias burying: unique variables without destructive reads. *Softw. Pract. Exper. 31*, 6 (2001), 533–553.

[7] BURNETT, M. M., AND AMBLER, A. L. Interactive visual data abstraction in a declarative visual programming language. *Journal of Visual Languages and Computing 5*, 1 (1994), 29–60.

[8] CLARKE, D. *Object Ownership & Containment*. PhD thesis, University of New South Wales, 2001.

[9] CLARKE, D. G., POTTER, J. M., AND NOBLE, J. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1998), ACM Press, pp. 48–64.

[10] ECMA. *ECMA-334: C♯ Language Specification*, second ed. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, Dec. 2002.

[11] FREEMAN-BENSON, B. N., MALONEY, J., AND BORNING, A. An incremental constraint solver. *Commun. ACM 33*, 1 (1990), 54–63.

[12] GAMMA, E., HELM, R., JOHNSON, R. E., AND VLISSIDES, J. *Design Patterns*. Addison-Wesley, 1995.

[13] GIFFORD, D. K., JOUVELOT, P., LUCASSEN, J. M., AND SHELDON, M. A. FX-87 Reference Manual. Tech. Rep. TR-407, MIT Lab. for Computer Science, September 1987.

[14] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[15] GRABMÜLLER, M., AND HOFSTEDT, P. Turtle: A Constraint Imperative Programming Language. In *Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence* (Cambridge, UK, December 2003), F. Coenen, A. Preece, and A. Macintosh, Eds., Research and Development in Intelligent Systems, British Computer Society, Springer-Verlag.

[16] GRUNDY, J. C., HOSKING, J. G., AND MUGRIDGE, W. B. Supporting flexible consistency management via discrete change description propagation. *Softw. Pract. Exper. 26*, 9 (1996), 1053–1083.

[17] HOGG, J. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 1991), ACM Press, pp. 271–285.

[18] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2003.

[19] KRZIKALLA, O. Constraint imperative programming with C++. In *MultiCPL'03 Proceedings* (September 2003), pp. 55–66.

[20] LIEBERHERR, K., HOLLAND, I., AND RIEL, A. Object-oriented programming: an objective sense of style. *SIGPLAN Not. 23*, 11 (1988), 323–334.

[21] MCGRAW, G., AND FELTEN, E. W. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, 1999.

[22] MÜLLER, P., AND POETZSCH-HEFFTER, A. Universes: A type system for controlling representation exposure. In *Programming Languages and Fundamentals of Programming* (1999), A. Poetzsch-Heffter and J. Meyer, Eds., vol. 263 of *Technical Report*, Fernuniversität Hagen.

[23] MYERS, B. A., MCDANIEL, R., MILLER, R., ZANDEN, B. V., GIUSE, D., KOSBIE, D., AND MICKISH, A. The Prototype-Instance Object Systems in Amulet and Garnet. In *Prototype-Based Programming*, J. Noble, A. Taivalsaari, and I. Moore, Eds. Springer-Verlag, 1999, pp. 141–176.

[24] NIEMEYER, P. *BeanShell*. `http://www.beanshell.org/`.

[25] NOBLE, J., CLARKE, D., AND POTTER, J. Object ownership for dynamic alias protection. In *TOOLS '99: Proceedings of the 32nd International Conference on Technology of Object-Oriented Languages* (Washington, DC, USA, 1999), IEEE Computer Society, p. 176.

[26] NOBLE, J., VITEK, J., AND POTTER, J. Flexible Alias Protection. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming* (London, UK, 1998), Springer-Verlag, pp. 158–185.

[27] POTANIN, A., NOBLE, J., CLARKE, D., AND BIDDLE, R. Defaulting Generic Java to Ownership. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs in European Conference on Object-Oriented Programming* (Oslo, Norway, June 2004), Springer-Verlag.

[28] SCHÄRLI, N., BLACK, A. P., AND DUCASSE, S. Object-oriented encapsulation for dynamically typed languages. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2004), ACM Press, pp. 130–149.

[29] SMITH, W. R. NewtonScript: Prototypes on the Palm. In *Prototype-Based Programming*, J. Noble, A. Taivalsaari, and I. Moore, Eds. Springer-Verlag, 1999, pp. 109–140.

[30] THOMAS, D., AND HUNT, A. *Programming Ruby*, 2nd ed. Addison Wesley, 2005.

[31] UNGAR, D., AND SMITH, R. B. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1987), ACM Press, pp. 227–242.

[32] ZANDEN, B. V., MYERS, B. A., GIUSE, D., AND SZEKELY, P. The importance of pointer variables in constraint models. In *UIST '91: Proceedings of the 4th annual ACM symposium on User interface software and technology* (New York, NY, USA, 1991), ACM Press, pp. 155–164.