Dynamic Ownership in a Dynamic Language

Donald Gordon
Innaworks Development Ltd
Wellington, New Zealand
don@dis.org.nz

James Noble
Compter Science,
Victoria University of Wellington, New Zealand
kjx@mcs.vuw.ac.nz

Abstract

Object aliasing causes as many problems in dynamic languages as it does in static languages. Most existing work on aliasing, such as ownership types, relies on static checking based on annotations and type declarations. We introduce ConstraintedJava, a scripting language based on BeanShell that provides dynamic ownership checking. Dynamic ownership provides the encapsulation benefits of static ownership types but preserves the flexibility of dynamic languages.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages—Constructs and Features

General Terms Languages

1. Introduction

Dynamic languages are more flexible than static languages. Closures, comprehensions, polymorphism, reflection and message sending were all supported in dynamic languages long before their static counterparts. Dynamic languages are also safer than static languages: null pointers, type errors, stack overflows, and message not understood errors that can crash programs (or entire machines) in static languages are prevented or handled gracefully in dynamic languages.

In this paper we describe our efforts in applying dynamic techniques to catch errors in program design — by encapsulating abstractions against aliasing. Consider the following example of browser-side scripting implementing a web-based document handling API:

```
class DocumentProxy {
 // ... public API goes here
  private var docServerSocket = new ServerSocket(...);
  public hole() {return docServerSocket}
}
```


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DLS'07, October 22, 2007, Montréal, Québec, Canada.

Copyright © 2007 ACM 978-1-59593-868-8/07/0010. . . $5.00

The DocumentProxy class offers a public API to a back-end document server, storing the network socket in a private docServerSocket field. By ignorance, weakness, or deliberate fault, a programmer could write a method that exports an alias to this socket, breaking the DocumentProxy abstraction and potentially corrupting the back end database. Note that access restrictions such as **public** or **private** qualifiers do not help here, because they prevent access to variables but not to the objects held by those variables.

2. Dynamic Ownership

Dynamic Ownership [23] is our proposal to strengthen encapsulation in dynamic languages. Most other alias protection schemes, such as Balloons [1], Islands [14], and Ownership Types [7], rely on features of static languages that dynamic languages lack. Many dynamic languages allow code to be added to a running program. This means that we cannot 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 [12], Ruby [29], and even Java [2].

Our model for dynamic ownership provides alias protection and encapsulation enforcement by maintaining a dynamic notion of object ownership at runtime, and then placing restrictions on messages sent between objects based on their ownership. 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. Dynamic ownership can also be extended easily to support features such as ownership-based ("sheep") clone, exported interfaces, and ownership transfer that are difficult to support in static ownership systems. As the encapsulation restrictions are based on runtime behaviour rather than compile-time structure, they are ideally suited to implementation in a dynamic language.

2.1 Encapsulation Guarantees

Dynamic Ownership enforces two invariants: representation encapsulation and external independence [23].

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 message invocation 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 sends. We sometimes refer to these external objects as an object's *arguments*.

Note that these invariants only restrict the type of messages sent between objects. They do not restrict in any way the passing of references to methods, or the storage of references in fields or local variables. This lack of restrictions on references is a major difference from many type systems designed to provide aliasing protection.

2.2 The Ownership Tree

To enforce these invariants, Dynamic Ownership gives each object an *owner*: some other object in the system which owns it. This ownership relation forms a tree (the *ownership tree*) as each object has only one owner, and ownership may not be cyclic.

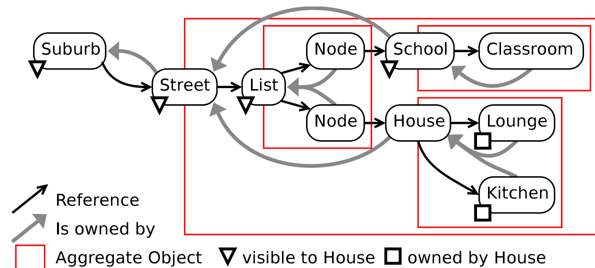

Figure 1. Encapsulation: ownership, visibility and aggregation

We write b.**owner** = a when an object a in the ownership tree *owns* another object b. For example, in figure 1, the House *owns* the Lounge. Alternatively, b is *owned by* a when b.**owner** = a. For example, in our example of the suburb, the Nodes are *owned by* the List. 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 1 the Street *contains* the Lounge, but does not *own* it. Objects which would otherwise have no owner, such as the first objects created by a program, are owned by the topmost node in the ownership tree called the *root* owner.

Dynamic ownership programs can be thought of as a tree of encapsulated objects, the set of the other objects they *contain* making up their representation. Every object in the ownership tree 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 ownership tree places no constraints on the graph of other pointers in the system. For example, figure 1 shows a system with a street object containing a number of buildings, a house and a school. The street owns a list and the buildings within the street, but not the nodes that make up the list's internal representation. We think of the street as the interface to the list and its contents, and the list as the interface to the list's structure (the nodes), but not the objects held in the list (the buildings). As each object's ownership information is explicitly stored in the object itself, this information is retained when the object (such as a building) is placed in a container object (such as the street's list of buildings) — the ownership information is not lost or changed.

Because ownership and pointers are independent, an object's ownership can be changed dynamically. A room could be moved between buildings, then its ownership changed to reflect its new status. Permission to change ownership lies with the object's owner.

2.3 Visibility

Dynamic ownership uses the ownership tree to restrict interactions between objects. We say that an object a is only *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 any encapsulation boundaries. Visible objects can be any number of links *up* the ownership tree but at most one link *down*, because going down the tree requires crossing into encapsulation boundaries.

In figure 1, the Suburb, Street, List and School are all *visible* to the House. This is an incoming visibility; the Suburb is visible to the House, but the House is not visible to the Suburb. House would not be breaking through an encapsulation boundary to access School, as School is owned by Street, and Street contains House. Direct access to the School's Classroom from House would breach School's encapsulation, however.

Dynamic Ownership uses visibility directly to guarantee representation encapsulation. The rule is a simple one: messages are sent only if the receiving object is visible to the sending object. Any other send would require crossing an encapsulation boundary, so a system providing dynamic ownership must ensure each message send is to a visible object.

2.4 Message Sends

External independence is a more subtle condition: it requires that objects only depend on the mutable state of their owners, or of the objects they themselves contain. In figure 1, School is visible to House (so House could send a message to School),

but if that message's result relies on School's state, then it will establish an external dependency of House upon School. On the other hand, House may depend on the state of the Lounge or Kitchen that it owns. By sending messages to Lounge or Kitchen, House may also depend on the state an objects owned by the Lounge or Kitchen, because it transitively contains all of those objects.

To enforce external independence, Dynamic Ownership further classifies message sends. If the target of a message send is **this** or is owned by self then we say the send is *internal*, while sends to other visible objects are *external*. External sends are prevented from establishing dependencies on mutable state. Dynamic Ownership recognise two kinds of *externally independent* messages that do not establish dependencies — *pure* messages that do not access state, and *oneway* messages that do not return results.

Pure messages Pure messages 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 send (i.e. other messages that may establish external dependencies).

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 messages Oneway messages are defined as being able to perform any action, like a normal message send, including sending normal methods, or reading and changing non-final fields, apart from returning a value or throwing an exception.

As oneway messages are allowed to access mutable state they are useful for gathering output such as updates to window display, sending test results in a test harness, implementing an assertion facility, or providing additions to an HTML document to be sent to a web browser. Adding items to a list could be a oneway message, as it does not need to return any information.

Message Dispatch Dynamic Ownership thus requires two-dimensional check whenever a message is dispatched: depending on the relationship between sender and receiver (internal, external, or not visible) and the type of message (unrestricted, pure, or oneway). This check dynamically ensures the encapsulation invariants [13].

Table 1 shows which categories of message sends are permitted ($\star$) or forbidden ($\times$). An internal send (to the sender itself, or to an object the sender owns) is always permitted, unless the currently executing invocation is pure. A pure method invocation, or an outgoing send (to a visible object that is not owned by the sender) may make only pure or oneway sends. Finally, any kind of message send to a receiver that is not visible to the sender is forbidden.

Relationship of Sender and Receiver	**Message Type**		
	Normal (Unrestricted)	Externally Independent	
		Pure	Oneway
Internal (from normal or oneway)	$\star$	$\star$	$\star$
Internal (from pure method)	$\times$	$\star$	$\star$
Outgoing (visible but not owned)	$\times$	$\star$	$\star$
Not visible	$\times$	$\times$	$\times$

Table 1. Dynamic ownership message dispatch.

Adding such a restrictive message dispatch check to a dynamic language reduces the flexibility and power of the language, as well as (hopefully) increasing safety. The project we are engaged in is first to design languages supporting Dynamic Ownership, and then to investigate whether (and in what contexts) its benefits overcome its liabilities.

3. ConstrainedJava

We have designed and implemented the ConstrainedJava language as a proof of concept design incorporating Dynamic Ownership [13]. ConstrainedJava is built as an extension to BeanShell [22], a dynamically typed dialect of Java. To BeanShell's object model, we added support for managing object owners, and a few programmer visible methods (**owner**, **gift**, **export**) and method qualifiers (**factory**, **pure**, **oneway**).

3.1 Classes and Objects

ConstrainedJava classes, methods, fields and the like are declared similarly 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);
}}
```

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 class name, 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. A 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.

3.1.1 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 Lounge object's owner would be the House object that created it:

```
class House {
  var myKitchen = new Kitchen(); ...
```

3.1.2 Sending Messages

Message sending is also syntactically similar to Java — although message sends are dynamically checked to ensure they don't violate the rules of the encapsulation enforcement part of Constrained Java (see section 2.1).

```
  someObject.callMethod();

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

3.1.3 Blocks

ConstrainedJava provides Smalltalk-like blocks, i.e. closures. A block 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 block returns the value provided to the return statement in the block.

For example, a method that returned 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 sender of the firstAbove method will have the value of the variable a returned to it. If no match is found, the method returns **null**.

3.2 Ownership in ConstrainedJava

ConstrainedJava implements the Dynamic Ownership structure described in section 2.2. An owner pointer is present in every object, which can be read by sending 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, by sending a notifyChanged() method.

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

An object's **owner**() returns a reference that is treated as any other object reference: it may be stored elsewhere, passed to other objects and receive other messages. Note that an object does not have control over which object owns it: if messages are sent to an object's owner, care must be taken to ensure that the owner is able to handle the messages sent to it.

3.2.1 Ownership Change

Dynamic Ownership allows object ownership to be changed at runtime. Programs may need to change objects' ownership either to reflect changes in the domain (as in a Kitchen moving between buildings) or to support implementation efficiency (moving nodes to merge linked lists).

Changes of ownership are 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 within. We check that this does not create a cycle in the ownership tree by ensuring that the object whose ownership is being changed

does not contain its new owner: this ensures the ownership tree remains a tree. Operations that affect the ownership of an object may only be performed by the object's current owner.

Ownership change occurs 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 — but of course changing an object's ownership will have consequences for its visibility, and thus for which messages it can receive.

3.2.2 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 sender.

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 sender. If the object containing the factory method does not own the object returned by that method, the program will throw an OwnershipError.

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 sent 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;
}}
```

Note that after the newWidget method returns, the factory will retain a reference to the newly created widget, even though the factory no longer owns the widget. Assuming the widget is still visible to the factory, the factory will be able to send oneway or pure messages to the widget, for example to allocate or deallocate caches, or to redraw the screen.

This facility for factory methods could be emulated with the **gift** method described in the previous section The requirement to transfer ownership of an object to a method's sender is present sufficiently often that it is nonetheless a useful feature to have.

3.2.3 Pure and Oneway Methods

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) {
  // ...
}}
```

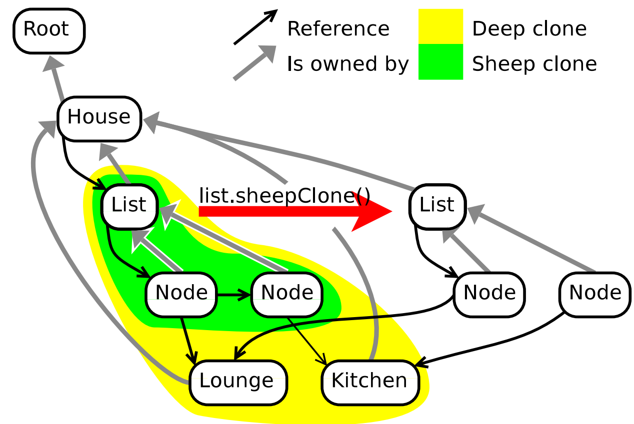

Figure 2. Sheep clone of an object

if it does not need to return a result.

Similarly, a message which returns an object's hash code could be labelled **pure** — assuming the method only accesses final fields or sends other **pure** methods:

```
class Student {
  final int number; // set by constructors
  final String surname; // set by constructors

  public pure hashCode() {
    return number + surname.hashCode();
}}
```

Neither **oneway** nor **pure** modifiers impose any compile-time restrictions. If a pure method accesses mutable state or sends a normal method, an OwnershipException is thrown, while return values and exceptions from **oneway** methods are discarded and not returned to the sender.

3.3 Cloning Objects

Having ownership information available allows the implementation of an ownership-directed *sheep clone* [23].

A sheep clone is somewhere between a shallow clone (cloning only the target object) and a deep clone (cloning the transitive closure of the target and all objects it refers to). An ownership based *sheep clone* acts as a "do what I mean" clone for aggregate objects, cloning the objects which are contained by the target object.

Like a deep clone, a sheep clone 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 2, 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 clone 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 cloning.

Then the objects identified for cloning by the last step are duplicated. The ownership structure of the original object tree is replicated, as are the references between the cloned objects. References to objects outside the set of cloned objects are retained, so the cloned nodes still have pointers to the same lounge and kitchen objects as the original nodes.

3.4 Exported objects

Interface objects, such as iterators, present a problem with the message send rules in section 2.4. 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 3).

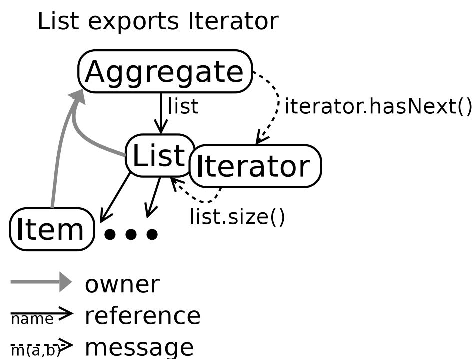

Figure 4. Export: the interface solution

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. 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 b has already exported object c , then when a exports b , all three objects will share the same ownership.

```
class List {
  ...
  factory iterator() {
    it = new Iterator(this);
    it.export();
    return it;
}}
```

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 4, 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 links 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.

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, and associated rules and guarantees. 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. Messages between these boundary objects, and from any one of them to any of the objects they jointly own are classified as internal calls.

Internally, adding an export operation means that objects' owner fields are supplemented by an *ownership context* field, referring to an ownership context object, which may have one or more objects associated with it. When one object exports another, the exported object shares the exporting object's ownership context. The rules for message sends from table 1 are reinterpreted in terms of these contexts. In the eyes of the ownership system, the exported and exporting objects are now the same, as they share the same ownership context.

Exporting should be used sparingly, however, 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.

4. Implementation

The ConstrainedJava language is a modified version of the BeanShell interpreter, with extensions to support Dynamic Ownership. Adding Dynamic Ownership to the BeanShell required a number of changes. 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, with ownership contexts supplementing owner fields, as described above.

Ownership checks Checks have been added to facilitate message restrictions. Every method call is checked to ensure

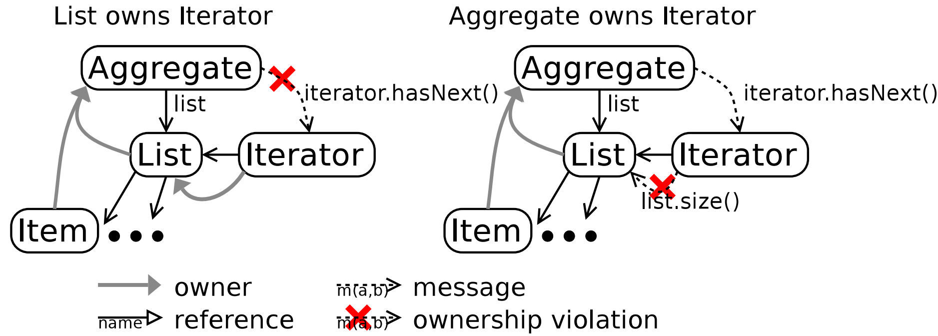

Figure 3. The interface problem

that the message send restrictions are not violated, traversing the ownership tree as necessary, and throwing an exception if the check fails..

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 each field access to ensure that pure mode only accesses final fields.

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

Finally, **gift**, **export**, and **factory** method sends are checked so that they may only be sent to an object by that object's owner (or returned by the owner, in the case of **factory** methods) or an exception is thrown.

Calling Java ConstrainedJava inherits from BeanShell the ability to call native Java code, and to instantiated native Java objects, as if they were ConstrainedJava objects. Because ConstrainedJava uses reflection to call Java methods (rather, say, than interpreting method bytecodes), ConstrainedJava 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, losing the ownership checks. We recommend programmers compartmentalise Java objects to a small part of the program, and they need to be aware that ConstrainedJava cannot retrofit ownership checking into those objects.

4.1 Performance

Our implementation of ConstrainedJava is only a proof of concept: it has not been optimised for performance. To gain some impression of the overhead imposed by ownership checking, we instrumented ConstrainedJava 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.

First, we gathered some ownership statistics by running a series of small benchmarks totalling 1700 lines of code. While these benchmarks are not large enough to be sure the results will generalise to large systems, we think the results are indicative. Most (77%) of messages (this includes field accesses) are sent to an object directly owned by the message sender — these are internal sends of unrestricted messages. Only 22% are external sends, restricted to being externally independent — pure or oneway. The remaining 1.2% are (unrestricted) internal sends within an object. 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 is the sender.

A simple 70 line program [13, 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 ConstrainedJava prototype implementation, both with encapsulation enforcement on and turned off. For comparison, we also include results from other scripting languages. Averages are over the last 40 iterations of a loop of 47 runs, with an inner loop that performs 4000 inserts.

Interpreter	Min	Ave
ConstrainedJava (no EE)	367	380
ConstrainedJava (EE)	389	404
BeanShell 1.3.0	259	267
BeanShell 2.0b4	3566	3617
JavaScript	20	21
Python 2.4.3	49	51
OpenJFX-200707201531	631	637

Table 2. Performance (all times in milliseconds)

The data (table 2) show that maintaining the ownership information, which is still done even with encapsulation enforcement turned off, causes the program to run 41% slower. Adding message send checks only reduces performance by a further 6%.

The reason for the much slower BeanShell 2.0b4 result is a new structure for handling BeanShell-generated objects. We

include these figures to show that while ConstrainedJava's performance is slower that BeanShell 1.3, it is still significantly quicker than BeanShell 2.0. Comparison with other scripting languages shows that while both BeanShell and ConstrainedJava are much slower than optimised interpreters such as JavaScript, the performance is comparable with the recently released language JavaFX Script.

5. Discussion

In this section we discuss and evaluate some features of ConstrainedJava, based on our experience so far.

5.1 Labelling Methods

ConstrainedJava requires externally independent methods to be labelled with a **pure** or **oneway** method modifier. Checks are then 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 sender. Therefore, it would be possible to partially or completely forego method labels and rely on 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 sends (see section 4.1) problems with code in such methods will only show up intermittently, when they are sent by some other object.

Oneway mode will not cause ownership errors to be thrown, rather exceptions or return values are silently discarded. Methods declared as pure, however, will cause ownership errors to be thrown when non-final fields are accessed, or non-externally independent methods are sent. For example, in the following class there are two pure methods, one which accesses a non-final field, which is disallowed dynamically, 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++;
}}
```

The point here is that the increment m++ is treated as a normal message send that reads and writes state, so it cannot be called from a pure method, however encapsulating it inside the **oneway** incM helper method masks those effects from the calling thread, and makes an extra, effectively outer-level, call on the Foo object to update the field.

This ambiguity, however, does illustrate one design choice in ConstrainedJava: to annotate methods, rather than message sends: there is no distinction in the caller between sends to pure, oneway, and normal methods. An alternative design could distinguish these syntactically within the sending method, rather than at the receiver, however exploring that part of the design space remains as future work. This would also remove another issue with this design, which is that different subclasses can implement the same messages in different ways — a oneway method can override a normal method that overrides a pure method — and in any combination. We have designed our programs so that this issue has not arisen, but it could lead to problems where dynamic dispatch ends up selecting different types of methods without warning.

5.2 Blocks

ConstrainedJava's blocks (section 3.1.3) provide the same facilities as 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 3.4) to deal with blocks. A block never has a separate owner – it inherits the ownership context of the object that created it, effectively being exported by its creator. Thus, blocks 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 blocks with a forEach method on a collection to iterate through that collection requires the

block to be marked **oneway**. The collection's call back to the block counts as an external send, and is thus allowed. For example, in the following code snippet:

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

when the object representing the block **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 ifTrue message can be sent to a singleton boolean object, if that object is True, then it cannot send a message back to the block passed to it, as the block would not be visible to the shared singleton True object, which would be owned by the root object.

5.3 Binary Methods

The issue with blocks illustrates a wider problem the Dynamic Ownership model has with binary methods — that is, methods involving two or more objects. In static languages, such methods can also cause problems for object-oriented type systems [5]. Consider a simple equals method:

```
class Avatar {
 int level = 1;
 Weapon weapon = new Sword();
 boolean equals(Avatar other) {
  return (level == other.level) &&
    (weapon.equals(other.weapon));
}}
```

The equals method cannot be pure, because it references the mutable level and strength fields; not can it be one way because it must return a result to its caller — we show an (unnecessary) explicit **boolean** return type here. The method also needs to read and return the mutable state of its other argument. Both objects must be mutually visible to each other, so they must be very closely related indeed in the ownership hierarchy. This can cause problems in practice, especially as a recursive comparison of two complex objects is practically impossible — if each Avatar owns their Weapon object then the recursive call weapon.equals(other.weapon) will always fail, because other.weapon will not be visible from **this**.weapon, assuming **this** and other are sibling objects.

Our implementation of ConstrainedJava includes several experimental mechanisms to address this problem, by granting a method temporary access ("borrowing" [4]) to objects passed as method arguments — much as a method always gains internal access to **this**) [13]. We plan to continue to investigate support for binary methods in future language designs.

5.4 Patterns in ConstrainedJava

Finally we describe how a number of common design patterns [11] interact with the object ownership and encapsulation enforcement provided by ConstrainedJava.

5.4.1 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 send 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.

5.4.2 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.

5.4.3 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, sends to externally independent methods could be made to the visitor object. In this case, however, the visitor would have no ability to make sends to the elements at all, and only send externally independent messages 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 back from the visitor 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 [17].

5.4.4 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. If it is used with the visitor pattern, however, the restrictions mentioned in section 5.4.3 would apply.

5.4.5 Factory Method

A factory method instantiates an object on behalf of some other 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 sent from the object they are part of, or that object's owner. Dynamic Ownership provides the factory method modifier (see section 3.2.2) which transfers the ownership of the object returned by the factory method to the object sending then message.

5.4.6 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 send 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 sent. ConstrainedJava does not currently provide a mechanism to support this.

5.4.7 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 [17].

6. Related Work

Many systems have been proposed to deal with the 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++ [15] and C♯ [10].

Islands [14] enforce encapsulation via programmer annotations on fields and methods. 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.

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.

Flexible Alias Protection (FLAP) [24] 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 send clean methods, which are not allowed to access mutable state. Dynamic Alias Protection was first proposed as a dynamic analogue to Flexible Alias Protection [23]. ConstrainedJava's Dynamic Ownership is the first implementation of this proposal, extends it in significant ways (exporting and ownership transfer), and demonstrates that the performance overheads are manageable.

Ownership types [7] are static type systems that provide 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 refer to the same object. This ownership information is then used to provide a mechanism to limit the visibility of object references. Later work [6] extends this to allow support for interface objects, borrowing, and numerous other extensions. For example, SafeJava [3] extends Ownership Types to allow instances of inner classes to be exported; while Ownership Generic Java [26] 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.

Universe Types [20] extend Ownership Type by permitting read-only access to any object, but only read-write access to owned objects (internal sends). This prevents representation exposure, but does not prevent argument dependence — although in Universes, objects' invariants may not depend on external objects, mitigating that problem. Recent Universes implementations [9] provide dynamic ownership checking, to support dynamic "ownership casts" in an otherwise static system, and ownership transfer based on uniqueness [19]. In contrast, ConstrainedJava is a purely dynamic language, and object transfers are unaffected by any references to objects being transferred. The problem with all these approaches that follow on from flexible alias protection is that they are only suitable for statically checked languages — dynamic ownership aims to fill that gap.

The object-oriented encapsulation system [27] provides a flexible encapsulation enforcement system suited to dynamic languages. Classes can restrict how their subclasses can utilise them, and references have access policies allowing only certain sets of methods to be called using them. This work addresses different problems to Dynamic Ownership, since (like other encapsulation systems that are not ownership related) it controls only access to the names of methods and fields, but not to the objects stored in those fields.

Finally ConstrainedJava is related to other object-oriented dynamic languages that maintain tree structures of objects. Amulet and Garnet, for example, maintained an explicit object tree that modelled a UI widget tree [21]; NewtonScript's dual inheritance was often used similarly [28]. Similar structures were used in scripting languages for adventure games [16] and music [25, 30]. While these languages support an object tree structure, very few use it to provide any kind of encapsulation or ownership support to programmers.

One exception is the object-oriented multi-user "adventure" LambdaMOO [8, 18] that gives objects, attributes and verbs (methods) unix-like "permissions". The main use of these permissions is access control in the multi-user environment: if a programmer does not have write permission to an object or method, they cannot modify that object or method's code. Attribute permissions provide name-based read or write protection, both per-field and per-user, while other permissions allow objects to host new objects created inside them, or ensure that attributes of new objects will belong to those objects (rather than the factory that created them). Compared with ConstrainedJava, LambdaMOO's permissions are richer because they take multiple users into account, and control editing code as well as data, however LambdaMOO's objects (while organised into a tree) are not encapsulated within that tree.

Figure 5 roughly compares some of these (and some other) systems balance of expressiveness and safety.

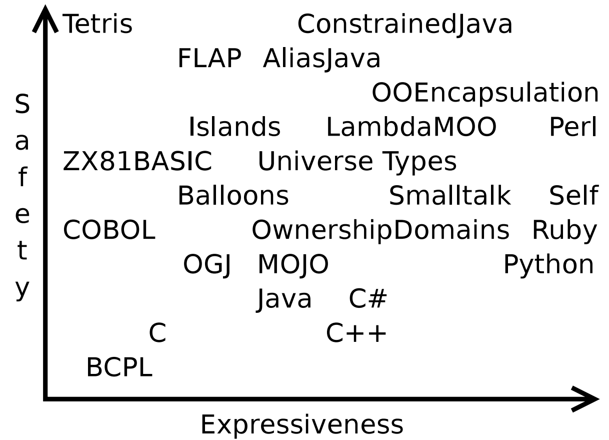

Figure 5. Safety vs Expressiveness

7. Conclusion

We have presented the design and implementation of ConstrainedJava, the first language to support Dynamic Ownership. ConstrainedJava includes a number of dynamic checks that ensure objects cannot suffer from representation exposure or external dependence, while maintaining the flexibility of a dynamic language. Dynamically recording ownership structure, for example, allows ConstrainedJava to support an ownership-based "sheep" clone, ownership transfer, and exported interfaces, with acceptable performance.

Ideally there are a number of further issues we hope to address in this design space. Given the development of scripting languages since this project began, we would like to implement the Dynamic Ownership and encapsulation enforcement system on top of a more mainstream language than BeanShell. It could be useful to produce a formalism for the ownership and enforcement system provided by ConstrainedJava, to prove that the encapsulation invariants are maintained by the language. Finally, we hope to gain more experience with dynamic ownership support in dynamic languages.

Acknowledgements

We are grateful to the anonymous reviewers, and to Dave Ungar, for their many robust comments. Thanks also to Stephen Nelson, who wrote the Python and Java FX Script versions of the performance benchmarks at very short notice. This work was supported by the Royal Society of New Zealand Marsden Fund.

References

[1] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP Proceedings*, pages 32–59, June 1997.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, 2000.

[3] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, New York, NY, USA, 2003. ACM Press.

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

[5] Kim B. Bruce, L. Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *TAPOS*, 1(3):221–242, 1995.

[6] Dave Clarke. *Object Ownership & Containment*. PhD thesis, University of New South Wales, 2001.

[7] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, New York, NY, USA, 1998. ACM Press.

[8] Pavel Curtis. *LambdaMOO Programmer's Manual*, March 1997.

[9] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

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

[11] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[12] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[13] Donald Gordon. Encapsulation enforcement with dynamic ownership. Master's thesis, Victoria University of Wellington, 2007.

[14] John Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA*, pages 271–285, New York, NY, USA, 1991. ACM Press.

[15] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2003.

[16] P. David Lebling, Marc S. Blank, and Timothy A. Anderson. Zork: A computerized fantasy simulation game. *IEEE Computer*, 12(4):51–59, April 1979.

[17] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: an objective sense of style. In *OOPSLA*, 1988.

[18] Colin McCormick. Colin's way easy intro guide to moo programming, March 1996.

[19] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *OOPSLA*, 2007. To appear.

[20] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for controlling representation exposure. Programming languages and fundamentals of programming, Fernuniversität Hagen, 1999.

[21] Brad A. Myers, Rich McDaniel, Rob Miller, Brad Vander Zanden, Dario Giuse, David Kosbie, and Andrew Mickish. The Prototype-Instance Object Systems in Amulet and Garnet. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming*, pages 141–176. Springer-Verlag, 1999.

[22] Pat Niemeyer. *BeanShell*. `http://www.beanshell.org/`.

[23] James Noble, David Clarke, and John Potter. Object ownership for dynamic alias protection. In *TOOLS*, page 176, Washington, DC, USA, 1999. IEEE Computer Society.

[24] James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In *ECCOP*, pages 158–185, London, UK, 1998. Springer-Verlag.

[25] L. Polansky, D. Rosenboom, and P Burk. HMSL: Overview (version 3.1) and notes on intelligent instrument design. In *International Computer Music Conference*, 1987.

[26] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for Generic Java. In *OOPSLA*, 2006.

[27] Nathanael Schärli, Andrew P. Black, and Stéphane Ducasse. Object-oriented encapsulation for dynamically typed languages. In *OOPSLA*, pages 130–149, New York, NY, USA, 2004. ACM Press.

[28] Walter R. Smith. Using a prototype-based language for user interface: The Newton project's experience. In *OOPSLA*, 1995.

[29] David Thomas and Andy Hunt. *Programming Ruby*. Addison Wesley, 2nd edition, 2005.

[30] Tim Thompson. Keynote — a language and extensible graphics editor for music. *Computing Systems*, 3(2):331–357, 1990.

[31] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA*, pages 227–242, New York, NY, USA, 1987. ACM Press.