Johan Östlund

# Realizing External Uniqueness

*... or how I learned to stop worrying (about representation exposure) and love the owner*

Master thesis at the department of Computer and Systems Sciences at Stockholm University (corresponds to 20 credits)

**Summary.** External Uniqueness was proposed as a solution to an abstraction problem inherent in most uniqueness proposals theretofore. External Uniqueness merges Ownership Types and Uniqueness in a way that loosens some of the restrictiveness in deep ownership, without compromising encapsulation. However, there is very little experience in realizing and using External Uniqueness and Ownership Types in practice. In this thesis we take the first steps into this uncharted area and, where applicable and possible, present remedies to inherent shortcomings in the existing proposals.

"... Our present attitudes and laws governing the ownership and use of land represent an abuse of the concept of private property...
Today you can murder land for private profit. You can leave the corpse for all to see and nobody calls the cops."
– Paul Brooks, The Pursuit of Wilderness (1971)

# Acknowledgments

I would like to take this opportunity to thank my supervisor Tobias Wrigstad for his untiring commitment, enthusiasm and support.

Also, I would like to thank Donald Knuth and Leslie Lamport for the typesetting of this thesis.

# Contents

# List of Figures

# 1 Introduction

Aliasing occurs whenever there exists more than one reference to an object (Hogg, Lea, Wills, deChampeaux, and Holt 1992). Aliasing may lead to unpredictable behavior. Take a look at the following code example. What is the output?

```
x.f = 1; y.f = 2; print(x.f);
```

What we know from this rather meager code segment is insufficient to answer that question. There are two possible answers depending on whether `x` and `y` are aliases or not. If `x` and `y` point to the same object the output will be `2`; otherwise `1`.

Over the years several proposals have been presented concerning alias management(Noble, Vitek, and Potter 1998; Clarke 2001; Hogg 1991; Aldrich, Kostadinov, and Chambers 2002; Hogg, Lea, Wills, deChampeaux, and Holt 1992). These proposals, however, suffer from differing yet often severe problems, such as breaking of abstraction and being too restrictive for real life applications (Clarke, Noble, and Potter 1998b; Clarke, Noble, and Potter 1999; Noble 2000; Liskov, Boyapati, and Shrira 2003; Clarke and Wrigstad 2003). We believe that External Uniqueness (Clarke and Wrigstad 2003) may solve these problems by redefining uniqueness to not include innocuous internal references but only active ones by utilizing the properties of deep ownership (uniqueness and deep ownership are discussed in Sections 2.2 and 2.4 respectively.) There is, however, little experience regarding the practical use of External Uniqueness. We hope that this thesis will shed some light on this unexplored area.

## 1.1 Purpose

Deep ownership types has been called too restrictive for actual programming. There is however little evidence to support or refute this claim as very few actual programs have been written in a language with deep ownership.

The purpose of this work is to gain experience in using the constructs of the Joline language (external uniqueness, owner-polymorphic methods, scoped regions, deep ownership) for writing non-trivial applications in order to evaluate their compatibility with normal programming practice.

We consider the applicability of the proposed constructs, constraints due to enforced encapsulation, additional syntactic baggage due to ownership annotations and borrowing, and chosen defaults for things such as string literals.

## 1.2 Methodology

In order to fulfill the purpose stated above we perform a qualitative study on the implementation and usage of External Uniqueness. We have implemented a compiler with support for External Uniqueness, and that process in conjunction with the implementation of non–trivial programs using External Uniqueness will reveal issues inherent in External Uniqueness and `Joline`, which is an implementation of External Uniqueness in a Java-like language, discussed in Section 4.

## 1.3 Contributions

This thesis presents the first experiences of practical use of External Uniqueness. We think these experiences are of interest and may be so to others as well, mostly because the lack of practical experience in this field. Our greatest contribution, however, we believe is the `Joline` compiler, which is the first compiler of its kind to implement deep ownership. Our `Joline` compiler enables other researchers to seamlessly carry on testing their theses, and further explore the practical usability of External Uniqueness and Ownership Types.

# 2 Background

## 2.1 Aliasing

Aliasing occurs when a single memory area is referenced by more than one reference simultaneously (Hogg, Lea, Wills, deChampeaux, and Holt 1992), or more practically, in object-oriented programming, when there exists two or more references to a single object. Aliasing is not always a problem, in fact, sometimes it is even wanted, and as long as it is controlled it may be beneficial, or required even. Aliasing problems, however, may occur unexpectedly and seemingly on their own, at which point aliasing may become a problem. Consider the code in Figure 2.1 (stolen from Clarke and Wrigstad (2003).) If `f1` and `f2` refer to the same file object this will generate an exception when the read method is called on `f2`, even though the code seems perfectly fine statically.

Several attempts to deal with the problems with aliasing have been made. These are categorized as *alias detection*, *alias advertisement*, *alias prevention* and *alias control* (described below.) All these techniques try to solve, or at least lessen the effects of unexpected aliasing. However, as Hogg et al. (1992) declare, all these proposals have inherent drawbacks and often there is a compromise between static checkability and ability to write useful code.

### 2.1.1 Ways of Dealing with Aliasing

In The Geneva Convention on the Treatment of Object Aliasing (Hogg, Lea, Wills, deChampeaux, and Holt 1992) the authors categorize the different ways of dealing with aliasing. These categories are described in short here. Ownership Types is not present in the categorization by Hogg et al. but deserves to be mentioned here nonetheless.

*Alias detection* – tries to detect aliasing patters in a program, either statically or dynamically. The problem here is that the static analysis will find a

```
public void example(File f1, File f2)
{
    f1.close(); // Closes f1
    f2.read();  // Requires f2 to be open
}
```

**Fig. 2.1.** Aliasing of file objects

lot of may–alias situations, which will have to be treated conservatively as must–aliases. This, of course, may lead to an abundance of situations being incorrectly labeled as aliasing.

*Alias advertisement* – method annotations may be used to reveal whether an object is aliased or not. This reveals the internal implementation of an object, not making it quite the black–box entity it is supposed to be. This can of course guide the programmer to write less erroneous programs, but as we shall see shortly, revealing internal implementation leads to other problems.

*Alias prevention* – statically guarantees that aliasing cannot occur within a given context. Again the problem is that a certain level of conservativeness is needed. Several alias prevention proposals have been presented, however, with constraints disallowing a programmer from implementing many common constructs (Clarke, Noble, and Potter 1998b; Clarke, Noble, and Potter 1999; Liskov, Boyapati, and Shrira 2003). One implementation of alias prevention is alias–free or unique references (Wadler 1990; Hogg 1991; Baker 1995; Almeida 1997; Noble, Vitek, and Potter 1998; Boyland 2001; Boyland, Noble, and Retert 2001; Boyapati and Rinard 2001; Aldrich, Kostadinov, and Chambers 2002; Clarke and Wrigstad 2003), which syntactically guarantee there be only one reference to an object. Unique references will be discussed in more detail shortly.

*Alias control* – strives to limit the harmful effects of aliasing. Examples of alias control schemes are read–only references (MacLennan 1982; Hogg 1991; Kent and Howse 1996; Noble, Vitek, and Potter 1998; Müller and Poetzsch-Heffter 2000; Boyland, Noble, and Retert 2001; Skoglund and Wrigstad 2001; Kniesel and Theisen 2001; Birka and Ernst 2004) where references may be made immutable, thus making aliasing less affecting, and the proxy pattern which may be used to replace destructive methods with non destructive methods by returning a copy instead of the object itself (Gamma, Helm, Johnson, and Vlissides 1994). An example is the `String` class in `Java` which has no destructive methods, and any change to a string object results in a new copy altered in the desired way (Gosling, Joy, Steele, and Bracha 2000).

```
public void example(unique File f1, unique File f2)
{
    f1.close(); // Closes f1
    f2.read();  // Requires f2 to be open
}
```

**Fig. 2.2.** Unique file objects

*Ownership Types* – uses owners (other objects) to form the types of objects. An object owned by a certain object is not type compatible with an object owned by another object, and thus they cannot be aliases (Clarke and Drossopoulou 2002). Ownership Types will be discussed in more detail shortly.

## 2.2 Uniqueness

Recent research states that up to 85 percent of all the objects in a general program are unique (Potanin and Noble 2002), i.e. each is referred to by one and only one reference at any time. That suggests that aliasing should not be such a great problem. However, experience tells us that when aliasing does occur, it often results in unexpected behavior. One way of dealing with aliasing is to make references unique. Several different treatments of unique references or unsharable objects have been proposed by Wadler (1990), Hogg (1991), Baker (1995), Almeida (1997), Noble et al. (1998), Boyland (2001), Boyland et al. (2001), Boyapati et al. (2001), Aldrich et al. (2002) and Clarke and Wrigstad (2003).

In Uniqueness fields that are unique are also annotated as such. Unique references are really simple to grasp. Either a variable holds the only reference to an object or it holds null. This may enable the compiler to generate more efficient code, for instance when a unique variable is assigned a new value the previous object may be disposed of by a garbage collector, unconditionally, since no other reference to it can exist, and the number of possible aliasing situations is drastically decreased, thus making it feasible for the compiler to check the code for unwanted aliasing. Consider the code in Figure 2.2 compared to Figure 2.1. In this example the arguments cannot be aliased since they are unique and therefore the problem discussed in the previous section cannot occur (of course, in this example, having only one of the arguments being unique would suffice.) The example is also depicted in the object graph in Figure 2.3.

In many applications, although being unique, objects need to be moved from one variable to another. Moving a unique object means assigning it to

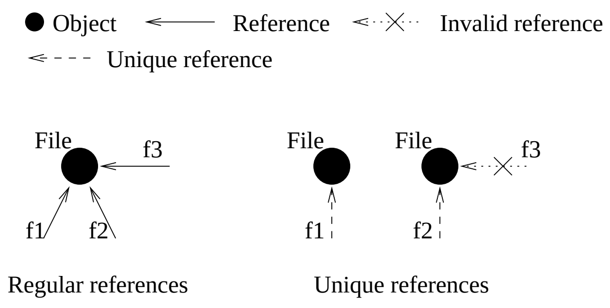


**Fig. 2.3.** Regular and unique references

another variable while making sure to preserve the uniqueness invariant. In order to maintain uniqueness in such situations there must be a way of ensuring that the unique object is not referred to by more than one reference after the move. Several ways of dealing with this have been proposed. A radical way to deal with this could be to swap the contents of two variables whenever an assignment is performed, as proposed by Harms and Weide (1991). Boyland (2001) proposes alias burying as another way to deal with movement of unique objects. Alias burying settles for *effective uniqueness*, where objects may actually be aliased as long as all references but one are “dead” when that variable is read. Destructive read is a third approach which is simple and intuitive. A destructive read means that the contents of the right hand side reference is nullified whenever assigned to another reference. This may be done explicitly, by the use of some operator, or implicitly as for instance in Hogg’s Islands (1991). Destructive reads will be explained more thoroughly when discussing External Uniqueness.

Another problem with unique references is that even a simple method call on a unique may break uniqueness (Wrigstad 2004). When a method is called the receiver is automatically aliased because a reference to `self`[1] is provided as a hidden argument to the method and in order to maintain a strong notion of uniqueness the receiver would have to be consumed, at least for the duration of the call. A borrowing statement has been proposed as a remedy to this issue. The borrowing statement allows the programmer to treat a unique variable as a non unique within the bounds of a lexical scope. However when such a construct is introduced other problems arise, one of which is how to prevent the borrowed variable from being retained in some persistent field, since that would break uniqueness when the unique variable, that was initially the source of the borrow, is reinstated. Existing proposals deal with this by introducing a new kind of reference. This reference

[1] An implicit method argument referring to the receiver. Other languages use `this`, e.g. Java and C++, or `current`, Eiffel.

```
class Example extends Object
{
   private Object obj = new Object();
   public Object getObject()
   {
      return obj; // Returns obj even though it's private
   }
}
```

**Fig. 2.4.** Encapsulation being broken

cannot be assigned to a field or variable (heap based), and thus the problem is solved. This, however, makes the type system much more complex. As will be discussed shortly External Uniqueness deals with this matter quite elegantly using the properties of Ownership Types.

As pointed out by Clarke and Wrigstad (2003) uniqueness suffers from an abstraction problem. The cause of this abstraction problem is that uniqueness annotations, declaring how `self` is treated internally, tend to leak out into the public interface of an object. Purely internal changes of implementation may therefore force the interface to change, thus making the change propagate, possibly, throughout the whole program.

## 2.3 Encapsulation

Encapsulation is the protection of an object's representation. A common encapsulation method is name–based encapsulation, used in for instance `Java` (Gosling, Joy, Steele, and Bracha 2000) and `C++` (Ellis and Stroustrup 1992). In name–based encapsulation fields are annotated with visibility modifiers which protect the fields from being directly accessed from outside the object. This works fine with value semantics, but not with references. An object's state is determined not only by the state of the object's fields but also by the recursive state of the objects referred to by those fields (its representation.) The effect of this fact is that the state of the object cannot be controlled by the object itself unless all references to the state constituents are from within the object. Consider the code in Figure 2.4. Annotating the field as `private` will prevent anyone from accessing the field directly, but as soon as the public method is called `obj` may be aliased thus there is no longer any way to guarantee the integrity of the inner state of the object (Grothoff, Vitek, and Palsberg 2001).

```
class Example extends Object
{
    this:Object obj = new this:Object(); // owner this
    public this:Object getObject()
    {
        return obj;
    }
}
```

**Fig. 2.5.** Ownership Types

## 2.4 Ownership Types

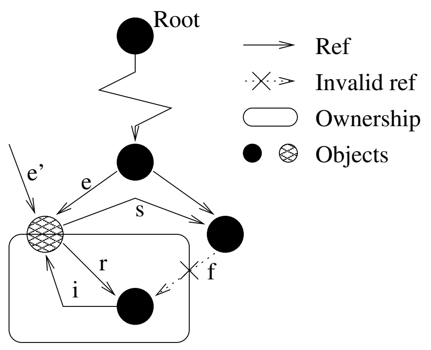
Ownership Types is an extended type system derived from Flexible Alias Protection (Noble, Vitek, and Potter 1998), fully described in Clarke's dissertation (2001). In Ownership Types objects have owners and can be owners of other objects. Owners have nesting relations (inside, outside) and form a tree structure rooted at the omnipresent owner *world* (Clarke, Noble, and Potter 1998a), which is outside all owners. An object is said to be inside another object (or part of its representation) if it is owned by that object (Clarke, Noble, and Potter 1998a; Clarke, Noble, and Potter 1998b). The type of an object is determined both by its class and its owner parameters. An important effect of this is that two variables with types with different owners can never be aliases (Clarke and Drossopoulou 2002).

It is allowed for an object to hold references to its representation or to objects that are outside of itself. To be able to form the appropriate types, classes are annotated with owner parameters that give local names for the outside owners and also capture the their nesting relations. An owner parameter is an object. It can be viewed as a permission, given at the time of instantiation, to reference the representation of that object.

In a linked list, for instance, owner parameters may be used to express the types of the data objects kept in the list. Take a look at the code in Figure 2.7. Without the class parameters the data objects would either have to be owned by *world* (or *owner*) or be a part of the list's representation, which would in the first case be impractical and in the latter render the list unusable, since adding or removing objects to it would not be allowed.

Whenever an object is created, one must state who owns that object. The owner of the object cannot subsequently change after being set.

The nesting relations of owners makes it possible to distinguish between the inside and outside of an object (Clarke, Noble, and Potter 1998a) (just look at the owner; if it is outside *this* it is an outside object and conversely.) This is a crucial property in External Uniqueness, which will be discussed



Reference kinds: e, e' – external to grey object
f – breaks deep ownership. s – sibling
i – internal. r – representation

**Fig. 2.6.** Object graph for deep ownership

shortly. Consider the code in Figure 2.5. In this case, compared to the one in the previous section (Figure 2.4), `getObject` cannot be called from outside the representation, since the necessary type for the returned object cannot be formed. The object itself and any internal objects that have been explicitly given the permission to reference the object can, however, form that type and may hence call the method.

Take a quick peek at the object graph in Figure 2.6. If the fields of an object are owned by *this*, which is the encapsulating object itself, no one outside that object may reference the objects referenced by those fields. In the example above calling the public method `getObject` from anywhere except from inside the object itself would be meaningless, or not permitted even, since the returned object cannot be referenced outside the owning object.

Ownership Types facilitates a stronger notion of encapsulation, but it is in some cases too restrictive. There are common constructs, e.g. iterators, which simply are impossible to implement in a language with deep ownership typing (Clarke, Noble, and Potter 1998b; Clarke, Noble, and Potter 1999; Noble 2000; Liskov, Boyapati, and Shrira 2003).

Wrigstad (2004) distinguishes between deep and shallow ownership. Shallow ownership lacks the nesting owner relations of deep ownership and is therefore more flexible but does not offer a strong enough property to build external uniqueness on top of. Also shallow ownership is easily broken, intentionally or otherwise, which cannot be afforded in case of External Uniqueness. Shallow ownership is used in e.g., ArchJava (Aldrich, Kostadinov, and Chambers 2002) and in Boyland et al.'s Capabilities for Sharing (2001). Shallow ownership is not implemented in Joline and we refrain from discussing it further here.

```
class LinkedList<data outside owner> extends Object
{

    this:ListNode<data> head = null;

    public void prepend(data:Object obj)
    {
        this:ListNode<data> node = new this:ListNode<data>();
        node.setData(obj);
        node.setNext(head);
        head = node;
    }

}

class ListNode<data outside owner> extends Object
{

    data:Object obj;
    owner:ListNode<data> next;

    public void setData(data:Object obj)
    {
        this.obj = obj;
    }

    public void setNext(owner:ListNode<data> next)
    {
        this.next = next;
    }
    ...

}

class Storage extends Object
{

    this:LinkedList<owner> li = new this:LinkedList<owner>();

    public void addToStorage(owner:Object obj)
    {
        li.prepend(obj);
    }

}
```

**Fig. 2.7.** Owner parameters

## 2.5 External Uniqueness

External Uniqueness was proposed as a solution to the abstraction problem (see Section 2.2) inherent in existing uniqueness proposals. External Uniqueness merges Ownership Types and Uniqueness. The use of deep ownership typing gives a strong notion of encapsulation while Uniqueness introduces much more flexibility. The use of unique references along with Ownership Types overcomes some of the restrictiveness inherent in deep ownership typing and allows for moving objects and changing their owners, or moving their bounds really. As in most other uniqueness proposals (discussed in Section 2.2) fields are annotated with the *unique* keyword. The unique keyword is not an owner. Instead it signals that the owner of the object referenced by the field or variable of the unique type is actually the field or variable itself. Any other unique field or variable is not type compatible and hence cannot alias the referred object.

Uniques have a movement bound, an owner, within the representation of which they may be moved[2]. This movement bound is crucial to maintain ownership soundness. Uniques may only be moved inwards in the ownership structure. Without this movement bound it is possible to break the external uniqueness invariant by using inheritance and subsumption (Wrigstad 2004). The moving of uniques means that one may create an object and pass it as message argument to another object which can then retain the externally created object and make it part of its representation without the risk of any residual aliasing compromising integrity of state. In a common language with name–based encapsulation, e.g. `Java` or `C++`, this would mean that the integrity of the object's state could not be guaranteed. With deep ownership where permissions should be set so that representation constituents are inside the encapsulating object this is not even possible, unless the permissions are widened, but that would break encapsulation anyway, and give a situation similar to common name–based encapsulation.

Consider the code in Figure 2.8. This example, though artificial, should not be too uncommon of a situation in a general program. The input stream object is constructed outside of the reader object and thereafter passed as a message argument to the reader, which retains the reference, making the input stream part of its representation. Directly after the passing of the input stream the stream is closed. This behavior probably is not expected by the reader which will try to read from the closed stream, causing it to throw an exception. In External Uniqueness the same code would be something like in Figure 2.9. In this case the Reader class forces the inserted object to be unique

[2] Movement bounds are omitted in the code examples for sake of brevity.

```
class Reader extends Object
{
    private InputStream is;
    public void setInputStream(InputStream is)
    {
        this.is = is;
    }
    ...
}
class Example extends Object
{
    private InputStream is;
    private Reader r = new Reader();
    public static void main(String[] args)
    {
        is = new InputStream("myfile");
        r.setInputStream(is);
        is.close(); // Puts is in a closed state
    }
}
```

**Fig. 2.8.** Passing without uniqueness

and thus the only reference to the inserted object will be that of the reader. Thus violations of state as in the previous example (Figure 2.8) are no longer possible; the reader object is in total control of its own state.

External Uniqueness has several other features, among which a stronger notion of aggregation, perhaps, is the most obvious. Because of the owner structure from Ownership Types and the uniqueness one may create object aggregates that are truly inaccessible by any other means than via the public interface of the aggregating object. Since deep ownership has the ability to distinguish between the inside and outside of an object multiple references to a unique object may be allowed as long as all but one are internal to the object, i.e. come from within the aggregate (Clarke and Wrigstad 2003). All references within a unique aggregate are inactive and therefore also innocuous, since any activation of such a reference has to be initiated from outside the aggregate via the unique reference. Internal references are effectively unique, and thus unique enough. In existing proposals it is not possible to allow internal references to a unique object (this is what's causing the abstraction problem) since no distinction can be made between an external reference and an internal innocuous ditto.

```
class Reader extends Object
{
    private unique InputStream is;
    public void setInputStream(unique:InputStream is)
    {
        this.is = is--; // *
    }
}
class Example extends Object
{
    private unique InputStream is;
    private Reader r = new Reader();
    public static void main(String[] args)
    {
        is = new InputStream("myfile");
        r.setInputStream(is--); // *
        is.close(); // throws NullPointerException
    }
}
```

**Fig. 2.9.** Passing with uniqueness in a Java-like language. The two lines with the asterisk comments use the destructive read expression, which will be discussed in more detail later.

# 3

# Our Case: The Library System

In order to really test External Uniqueness we thought that a system with a lot of object movement, or data sharing, would prove exposing. We decided a library system with libraries, borrowers and books might be a good example, as we had previous experience with a similar system.

The system has libraries, which in turn contain books and registered borrowers, see Figure 3.1 for a simple class diagram. A borrower may borrow a book from the library and return it when done. This should be easy to grasp. The important thing here, though, is that the ownership structure and uniqueness should be as conservative as possible in order to really test the model.

Potanin and Noble (2002) suggest that up to 85 percent of all objects in a system are uniquely referenced. When designing the system, in order to test these findings, we started out with all objects being unique, and only removed uniqueness on well considered grounds. As it turns out External Uniqueness, apart from managing aliasing, also offers quite a reality–like system model. Books and borrowers are unique, as are they in reality. Hence when a book is borrowed it is no longer in the library, also quite like in reality. This, of course, certainly is possible to do with a contemporary language, such as `Java` or `C++`, but the difference being that Joline, by supporting uniqueness, encourages it. The fact that the book is no longer in the shelf when borrowed removes the possibility of such errors as duplicate borrowings, which would in a contemporary language have to be checked explicitly. In order to keep track of borrowed books and the borrowers, we also have a borrowing card class, which stores the ids of the borrower and book each time a book is borrowed. Since both borrowers and books are unique the borrowing card cannot retain any references. We have therefore given each book and each borrower a unique id. In the case of books the `ISBN` and a copy number could be used as a unique id, but with other objects, say trees, there might not be such an obvi-

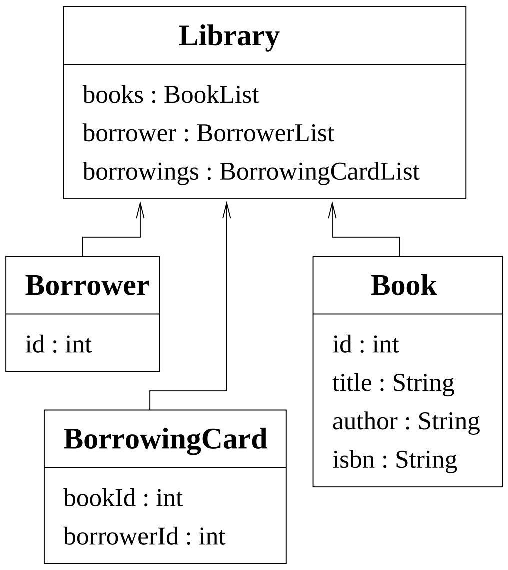


**Fig. 3.1.** Simplified class structure for Library system

ous unique value to make use of. We use an integer value that is automatically set upon instance creation. Whenever a book is borrowed the id of the borrower and book are stored in a borrowing card, and thus the borrowing is registered. Primitive types, such as integers use value semantics and can thus not be aliased, and are hence not affected by ownership or uniqueness. This is what allows us to “store” books and borrowers in a borrowing card. As will be discussed later we find comparing integers like this to be tedious and error prone, and we propose a new kind of reference which may be used only for id comparisons; calling methods, or accessing fields via such a reference would not be allowed. Hence aliasing of such references is innocuous and may be allowed even on unique objects.

# 4 Joline

This chapter discusses the `Joline` programming language and the features and constructs supported.

## 4.1 The Language

The `Joline` programming language implements External Uniqueness. `Joline` is built on the foundation of $\text{Joe}_1$ (Clarke and Drossopoulou 2002), which implements Ownership Types with effects (for more details on effects see for instance Nielson, Nielson and Hankin (1999).) `Joline` is a Java–like class–based object–oriented programming language. In `Joline` the effect annotations have been removed and other features have been added. In $\text{Joe}_1$ all owners (or permissions) are ordered outside *owner*, but aside from that owners have no nesting relations. In `Joline` the owners' relations are stated as class or method parameter annotations, which brings us to another feature absent in $\text{Joe}_1$, namely owner polymorphic methods which will be discussed shortly. We have implemented a `Joline` compiler using `Polyglot` (Nystrom, Clarkson, and Myers 2003), an extensible compiler framework written in `Java`.

Our `Joline` compiler is the only compiler of its kind to implement deep ownership.

## 4.2 Features

This section discusses some of the features in `Joline`. For a more extensive reference of the `Joline` programming language and External Uniqueness the reader is referred to Wrigstad's licentiate thesis (2004).

```
class Example extends Object
{
    public void method<a outside owner>(a:Object obj)
    {
        a:Object obj2 = obj;
        ...
    }
}

...

public void example()
{
    this:Example ex = new this:Example();
    owner:Object o = new owner:Object();
    ex.method<:owner:>(o);
}
```

**Fig. 4.1.** Owner polymorphic method

### 4.2.1 Owner Polymorphic Methods

Owner polymorphic methods are methods which in addition to regular arguments also take owner parameters. This is very similar to the owner parameterization of classes. Owner polymorphic methods are used to temporarily allow access to groups of objects which would otherwise be inaccessible. Whenever an owner polymorphic method is called it is required that the expected owners be given by the caller just as when creating a new object. See Figure 4.1. Since the owner parameters in the owner polymorphic method are visible only within the method body, objects passed as arguments can only be stored in stack based variables, i.e. local to the method. In order to retain a passed object in a persistent field the field has to have a compatible type, and such a type cannot be formed outside of the method body, since the owner is not known.

### 4.2.2 Destructive Read

In order to accomplish the transfer of a unique object from one variable to another without losing uniqueness, there is a destructive read expression. This is a unary expression that can be used on any unique lvalue[1]. Consider the code in Figure 4.2. The destructive read will return the value of the read variable and directly thereafter the read variable will be set to null. This way

[1] An expression that can appear on the left hand side of an assignment operator, i.e. in most cases/languages a variable or field.

```
class Example extends Object
{
    unique:Object obj;
    public void example()
    {
        unique:Object o = new owner:Object();
        obj = o--;    // Destructive read
        o.hashCode(); // Throws NullPointerException
    }
}
```

**Fig. 4.2.** Destructive read

the unique object has been safely transferred from one variable to another without compromising the uniqueness of the object.

### 4.2.3 Borrowing Blocks

When unique references exist in a system, in order for them to remain truly unique, they are somewhat unusable. As soon as a message is passed to such a unique object, the object is no longer unique. There will exist a reference to `self` during the method call. This makes calling methods on uniques impossible. For this reason there exists a construct called a Borrowing Block that lets the programmer borrow a unique variable and treat it as were it non unique within a limited context. During the borrow the value of the unique variable is put into a local non unique variable and the unique variable is destructively read. The borrowing variable has to be local to the borrowing block. Otherwise the variable would linger after the scope of the borrow has exited and thus possibly violate uniqueness, as would concurrent access which would be possible otherwise. The programmer may now use the new local variable to accomplish the task at hand. Right before the borrowing block exits the unique variable is reinstated with the value of the local variable. The local borrowing variable has to have a “fresh” owner that is automatically ordered inside the bound of the borrowed unique variable. This way the borrowed value cannot be retained in any persistent variable, since such a type cannot be formed outside the borrowing block. This is also true for return statements within borrowing blocks. The borrowed variable cannot be returned by the method, since such a return type cannot be expressed when writing the method header.

### 4.2.4 Scoped Regions

Scoped regions are lexical scopes used to reduce aliasing (Wrigstad 2004). Without the scoped region construct *this* is the innermost owner of a method.

However, when creating a scoped region a “fresh” owner, which is automatically ordered inside all known owners, must be named. This effectively makes the block into an owner. An object owned by the block can only exist during the lifetime of the block. This makes scoped heaps possible. An externally unique object has only one reference to it and if the object is owned by the block, destruction of the block, or stack frame, will automatically destroy the heap as well. Hence the term scoped heaps. This introduces a kind of generational ownership. One may view an externally unique aggregate as a generation which lives for a certain amount of time. Nested scoped regions are ordered inside outer scopes and thus outer scopes outlive inner scopes. Such an aggregate may thus be seen as a generation.

# 5 Jolinec – The Joline Compiler

Together with Tobias Wrigstad the author has implemented a Joline compiler using `Polyglot`, an extensible compiler framework written in `Java`. Our Joline compiler is the only of its kind to implement deep ownership. This chapter gives a short introduction to the compiler along with some experiences and thoughts on the implementation.

## 5.1 The Compiler

When deciding to implement a Joline compiler we had very little experience in compiler implementation. The choice to go with `Polyglot` (Nystrom, Clarkson, and Myers 2003) was not a difficult one to make. `Polyglot` is a framework, similar to most, e.g. the standard Java API, which enables implementation of languages similar to `Java`. The `Joline` language (and our implementation) should be viewed as a research language. The langauge does not support files or I/O and other common features of programming languages. The purpose of this implementation is to enable programming and evaluation of Ownership Types and External Uniqueness.

Cele and Stureborg (2004) performed a study on the implications of Ownership Types on the development process. In their study Cele and Stureborg implemented 5,6 KLOC with deep ownership. At the time there was no compiler for a deep ownership language, and hence all ownership related code had to be checked manually. It cannot be eliminated that errors may have been made in this manual checking, considering the amount of code. Therefore we find the use of a compiler a great asset not only because checking large amounts of code manually is tedious, but also because it gives our work greater realiability. Also, the implementation of the compiler gives a formal specification of constructs and features which have not previously been formalized for `Joline`, such as string literals, primitive types etc.

## 5.2 Implementation

Polyglot enables implementation of languages similar to Java. Like most object–oriented frameworks Polyglot lets the programmer create subclasses of framework classes and alter their behavior to suit one's needs. We have implemented the constructs described in the previous chapter and of course also ownership types and uniqueness. When compiling Joline code with jolinec, after handling all Joline specific syntax, such as owners and Joline specific constructs, the code is translated into ordinary Java, and may then be compiled with javac. This section discusses the Java implementation of the different Joline constructs.

### 5.2.1 Ownership Types

The owners in jolinec are simple names which have relations. When compiling Joline code the owner soundness is checked by comparing owners and their relations with the known owners in the scope. When the owner soundness is established all owner related code is omitted and regular Java code is produced. This has has the consequence that separate compilation is not supported. It is possible, though not clear, that this could be overcome by storing owner information in the class files for later retrieval, thus allowing for separate compilation.

### 5.2.2 Owner Polymorphic Methods

Owner polymorphic methods are very similar to regular methods in Java, except for the owner parameters. After having checked that the owners given in a call to an owner polymorphic method are sound all owner specific annotation is removed and what remains is a regular Java method.

### 5.2.3 Destructive Read

The implementation of the destructive read expression in jolinec is an adaptation of a solution presented by Boyland (2001). This solution uses a method call to make the destructive read an atomic enough expression. The implementation is depicted in Figure 5.1. To every class in the system we add a static method which just returns the first argument. The second argument will always be null since that is what effectively performs the destruction of the variable. We use a static method for faster dispatch and we use one method per class to aviod time–consuming casts. In an industrial context the compiler should be able to inline this method call.

```
// Joline code

class Test extends Object
{}

class Example extends Object
{
    public void method()
    {
        unique:Test o1 = new owner:Test();
        unique:Test o2;
        o2 = o1--;
    }
}


// Java code

class Test extends Object
{
    public static Test __jolineDreadMethod__(Test t1,
                                              Test t2)
    {
        return t1;
    }
}

class Example extends Object
{
    public void method()
    {
        Test o1 = new Test();
        Test o2;
        o2 = Test.__jolineDreadMethod__(o1, o1 = null);
    }
}
```

**Fig. 5.1.** Implementation of Destructive Read

This solution causes the uniqueness invariant not to be true at every point in the program since the arguments are aliased, but only momentary and, according to Boyland, assuming we use synchronization, this aliasing is harmless even in a multi–threaded environment (2001).

### 5.2.4 Borrowing Blocks

The `Java` implementation of borrowing blocks in `jolinec` relies solely on scopes and a destructive read. We check that the borrowed variable really

```
// Joline code
class Example extends Object
{
    public void method(unique:Object obj)
    {
        borrow obj as temp:o in
        {
            o.hashCode();
        }
    }
}


// Java code
class Example extends Object
{
    public void method(Object obj)
    {
        // borrow obj as temp:o in
        {
            Object o = Object.__jolineDreadMethod__(obj,
                                                    obj = null);
            try
                {
                    o.hashCode();
                }
            finally
                {
                    obj = o;
                }
        }
    }
}
```

**Fig. 5.2.** Implementation of Borrowing Blocks. The destructive read implementation is omitted for sake of brevity. See Figure 5.1.

is unique and that the proposed owner name is a “fresh” one. Then we create a local variable with the specified name within the block. This new non unique local variable is given the value of the borrowed variable, which is destructively read, before type checking all statements of the block. Right before the block exits the borrowed variable is reinstated with the value of the created local variable. Any reference held by that value is unique after the block exits, since all aliases are destroyed, thus the uniqueness invariant is not broken. Figure 5.2 depicts the implementation. The try–finally block ensures that whatever happens inside the borrowing block the borrowed variable is reinstated. Without the try–finally block a `return` statement within the

```
class Example extends Object
{
    unique:Object obj1;
    unique:Object obj2;
    public void example()
    {
        borrow obj1 as temp1:o1 and obj2 as temp2:o2 in
        {
            o1.hashCode();
            o2.hashCode();
        }
    }
}
```

**Fig. 5.3.** Extended borrowing block

borrowing block would cause the borrowed variable not to be reinstated and cause unexpected behavior and possible data races.

When writing code in Joline it soon became apparent that the chore with creating borrowing blocks as soon as a unique variable was to be used needed to be dealt with.

As soon as a unique variable is to be used it has to be borrowed, for reasons discussed earlier. Soon we discovered that much of the code written was in fact borrowing blocks; for some methods over 30 percent of the lines where borrowing statements. One solution to this problem was to implement an extended borrowing block syntax which allows the programmer to borrow several unique variables in one statement. This construct is shown in Figure 5.3. The implementation of this does not in any way interfere with the Joline definition of borrowing or borrowing blocks, since the resulting code is really just nested borrowing blocks, and the resulting code is the exact same as it would be had the programmer manually nested the blocks. The blocks are simply nested as statements inside another borrowing block. All owners are checked for uniqueness just as before, and the statements in the original borrowing block are simply put inside the innermost block (see Figure 5.4).

### 5.2.5 Scoped Regions

Scoped regions are implemented as regular blocks. The only action taken, before type checking the block, is to order the owner inside all known owners.

```
class Example extends Object
{
    unique:Object obj1;
    unique:Object obj2;
    public void example()
    {
        borrow obj1 as temp1:o1
        {
            borrow obj2 as temp2:o2 in
            {
                o1.hashCode();
                o2.hashCode();
            }
        }
    }
}
```

**Fig. 5.4.** Generated extended borrowing block

### 5.2.6 Primitive types

Primitive types are not part of External Uniqueness, and also not part of Ownership Types. Since primitive types use value semantics, and thus cannot be aliased, they are of little interest in the formal description of External Uniqueness. In `Joline`, being an actual language, they are however frequently needed and therefore they are supported and may be used just as in `Java`.

### 5.2.7 Strings

The `String` class in `Java` is somewhat special, since it does not contain any destructive methods. Any change to a string does not change the actual string but returns a copy altered in the desired way. This renders aliasing of strings innocuous, since they are "read–only". One might view strings as primitives with copy–semantics, even though the actual copy is deferred until actually needed. For convenience we have therefore exempted strings from the requirement that a unique must be destructively read (moved) when assigned to another variable, passed as argument or returned from a method.

### 5.2.8 String literals

String literals in Joline (and `Java`), i.e. an array of characters surrounded by quotation marks, return a `String` object initialized with the given characters. We have decided the returned string should be unique with movement bound *world*. This way the string may be retained in a `String` variable of any type. The other solution would be annotating the literal expression with an owner,

```
public void method()
{
    unique:Object o = new owner:Object();
    nocheck
        {
            System.err.println(o);
        }
}
```

**Fig. 5.5.** No Check Block

just as with normal objects, but since strings never have any additional permission parameters, that would depend on the owner, there is no need to do so.

### 5.2.9 No Check Blocks

We have implemented a construct that we call a `nocheck`–block (see Figure 5.5.) This is a block which is totally ignored by the `jolinec` compiler and simply passed on to `javac`, except for removing the actual `nocheck`–block header. The `nocheck`–block is not part of the `Joline` language and we only implemented this construct to be able to use `Java` features, such as printing error–messages, in a seamless manner. Beware, though, of using the `nocheck`–block construct to run code that the application depends on, since violation of the ownership or uniqueness invariants is allowed unconditionally.

## 5.3 Compiling Code

Separate compilation is not supported in our `Joline` compiler. This is because all owner information has to be available when compiling. After successful compilation all owner specific code is removed and regular `Java` code is produced. This code is then compiled with `javac` and regular class files are produced. Hence all `Joline` code must be present when compiling with `jolinec`. This is not an inherent restriction in Owership Types or External Uniqueness, but merely one of our implementation.

## 5.4 Unsupported Java Features

This section discusses features in `Java` that have been omitted in Joline for different reasons.

```
class Example extends Object
{
    public static void main()
    {
        ...
    }
}

// ... translates to ...

class Example extends Object
{
    public static void main(String[] args)
    {
        ...
    }
}
```

**Fig. 5.6.** Java main method

### 5.4.1 Main

Arrays are not considered in External Uniqueness and thus not present in Joline. Arrays were not necessary for our purposes and were therefore not implemented in the compiler. Adding array support to the Joline compiler, however, would be a trivial task. One huge consequence, though, comes as a result of this, namely that one cannot write a runnable program in Joline. All `Java` programs must have a `main` method as their entry point, which takes an array of strings as argument (the command line arguments passed to `java`) and therefore there was at first no way of running a pure Joline program. We have, however, implemented another `main` method which takes no arguments and translates to a regular `Java` `main` method (see Figure 5.6.) This of course removes the possibility of passing command line arguments to the program, but this can be easily circumvented in several ways, for instance by setting environment variables or piping data to the standard input.

### 5.4.2 Java API support

Our implementation of `Joline` does not support the use of precompiled `Java` frameworks. Precompiled `Java` frameworks, like the standard `Java` API, do not support owners and thus do not work with `Joline`.

For convenience we have implemented support for some frequently used classes from the `java.lang`–package. These classes may be used “out of the box” just as in normal `Java`. These classes are: `Object`, `String`, `Class`, `Exception` and `Throwable`.

❖ 6

# Analysis and Discussion

## 6.1 Issues and Remedies

In this chapter we discuss issues discovered while programming `Joline`.

### 6.1.1 Shared Data

Dealing with shared data in Joline is not quite as straightforward as in a contemporary object–oriented programming language such as `Java` or `C++`. This is a natural effect of the stronger encapsulation offered by deep ownership.

A special case of shared data is global data. In a contemporary language, such as `Java`, a common way to share global data is by storing the data in a class variable. And as class objects are global so is the data. In Joline this is problematic since the only owner allowed in a static context is *world*. The reason for this is that class objects are not instantiated in quite the same fashion as normal objects, i.e. the ones created by using the `new` keyword in the code. Normal objects are given an owner upon instance creation, and also may be given other permissions via owner parameters, but since class objects are created by a class loader the matter becomes a bit more intricate. As of now we have decided that the reasonable owner of class objects be *world*, because class objects should be globally visible. However, this of course has the consequence that class objects cannot know of any other owner than *world* (and *this* which is only accessible from within the class object, so it does not help us), except for in owner polymorphic methods, but they do not really offer a solution as the polymorphic owner is not known when forming the types of static fields. There might be a solution to this matter, however, as will be discussed shortly. The fact that class objects all have the owner *world* may render maintaining the owner structure impossible. This means that global data cannot be global in the sense that it be accessible via some global reference. However, by always passing a reference as argument to all objects that need the reference, upon creation, some degree of globality may be attained. In our library system the borrowers, to be able to return borrowed books, need to be aware of the libraries. We have a list containing references to all libraries, and since keeping this list global would destroy the owner structure, we pass a reference to the list to all borrowers.

This, of course, does not work very well if the global data is to be swapped for another object, since every object has its own reference to the global object and all those references will have to be updated. An obvious solution to that, however, is to wrap the actual object inside some proxy object, which is always the same, and swapping the wrapped object will then work (Gamma, Helm, Johnson, and Vlissides 1994). This would be analogous, in some sense, to pointers to pointers in the C programming language (Kernighan and Ritchie 1988).

A possible solution to the problem with global data might be to use several class instances with different owners. In `Java` no distinction is made between classes and types (this is not entirely true for the specification of `Java 5.0`, where genericity is added.) In Joline, however, since the type of an object is determined both by the class type and the owner, such a distinction is made. Because of this one could argue that there could be several class instances, one for each class–owner pair. These class instances would be given limited visibility in the system, since they may only be referred to by objects with the proper permission, perhaps giving a stronger notion of `package` similar to, though quite different from, confined types (as described by Vitek and Bokowski (2001) and Zhao, Palsberg, and Vitek (2005).) Using several class instances for different owners is quite similar to `Java`, which allows several instances of a class by using several class loaders. Having an owner polymorphic `Class.forName(...)` method could perhaps do this. It would enable class objects to be aware of arbitrary owners in addition to *world*, thus allowing the use of these owners in a static context. We have not implemented this feature and there may well be consequences not known to us. Nonetheless we think this is an interesting idea that should be further inquired into.

### 6.1.2 Exceptions and Events

In ownership research exceptions are seldom discussed. What permissions should be granted to an exception is unclear and not agreed upon. On the one hand it may seem reasonable that exceptions be owned by *world*; thus there would be no restrictions as to how far they may be thrown. On the other hand exceptions may carry a “source” reference to the object causing the exception to be thrown. In this case it would seem unreasonable that an

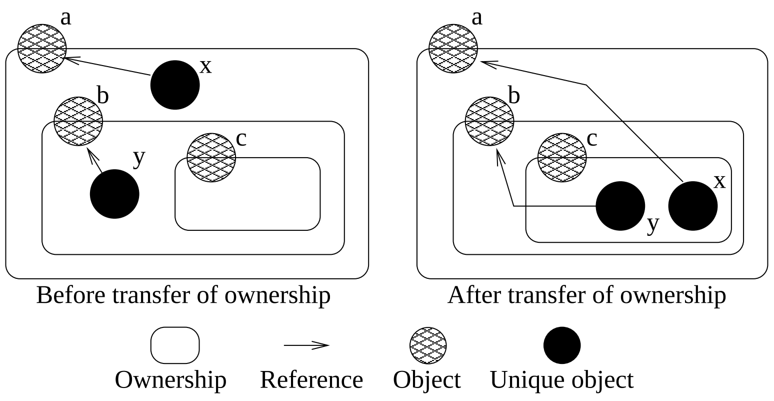


**Fig. 6.1.** The Unique Listener Pattern

object should reveal its inner workings just because it has been broken in some sense. Permissions would limit how far in the ownership graph an exception may be thrown.

In an event driven system, like most GUI-systems, where the system is idle waiting for the user to act, typically clicking a button or typing a key, the same discussion applies. These events are sent to listeners which react on the user's actions. What permissions should be assigned to these event objects (which often contain a "source" reference) is not a straightforward choice to make.

Cele and Stureborg identified a pattern, The Unique Listener Pattern, which is described in their masters thesis (2004). The pattern is depicted in Figure 6.1 (stolen from (Cele and Stureborg 2004).) The objects `a` and `b` want to register listeners `x` and `y`, respectively, to `c`. Since External Uniqueness allows for uniques to be moved inwards, `a` and `b` can create their listeners, equipping them with back–pointers, and pass them to `c`. When passed to `c`, `c` assumes ownership over the listeners, making them part of its representation. This means that `c` may pass representation objects to the listeners, which still have their back–pointers and can alert `a` and `b` when ever needed. This enables the use of proxy objects that are inside the object for which they act as proxy.

### 6.1.3 ID–references

In most applications identity checks, i.e. testing whether an object is the same (in whatever sense) as another, are common. When dealing with unique references this, of course, is not possible, since there can never be more than one reference to a unique object at any time. The need to be able to identify an object, however, remains. In our library system books and borrowers are unique. Whenever a book is borrowed it is moved from the library to the borrower, just as in real life. However, the library has to keep track of all books, even if borrowed, in which case the borrower has to be remembered as well. We use a borrowing card class to store information about the book and borrower, but since they are both unique we cannot retain references to any of them. The solution we have used is a unique id of some primitive type (such as `int`) that is stored and later may be used for comparing with the id of an object. Cele and Stureborg use the same technique in their masters thesis (2004). Primitive types are not affected by Ownership Types, since they cannot be aliased, and are also not part of External Uniqueness. The same argument may apply to objects of the String class in `Java`, since strings are immutable, and any change to such an object results in a copy being returned, hence aliasing of strings is innocuous. Storing and comparing ids this way we find to be a bit more tedious and error prone than comparing references, but it works. Apparently aliasing of objects only for the sake of identity checks is a reoccurring and often found pattern. If these aliases really are used only for identity checks, then this aliasing might be considered innocuous. This raises the question whether this calls for a weakening of ownership types (and uniqueness) to allow for a new type of reference which may only be used for identity checks?

Exactly what properties should be given such an id–reference is not known as of yet. Several proposals on read–only references have been presented over the years; MacLennan (1982), Hogg's Islands (1991), Kent and Howse's Value Types in Eiffel (1996), Noble et al.'s Flexible Alias Protection (1998), Müller and Poetzsch-Heffter's Universes (2000), Boyland et al.'s Capabilities for Sharing (2001), Skoglund and Wrigstad's mode system for read–only references in Java (2001), Kniesel and Theisen's JAC (2001) and Birka and Ernst's Javari (2004). The idea of read–only references is that they may only be used to perform read operations on objects, while precluding any change (write) of the object. Existing read–only proposals, however, suffer from representational or observational exposure (or both) which may break encapsulation, as pointed out by Boyland (2005b). Boyland, while criticizing existing proposals, also proposes fractional permissions as a possible solution to this problem (2003, 2005a). Time will show if this research will bear fruit and if so it might prove beneficial for our purposes. Until then, however, our idea of an id–reference is somewhat different. This type of reference would only be valid for id comparisons, and calling methods or accessing fields via such a reference would not be allowed. Whether these id–references should be affected by ownership bounds is not known as of yet. One might argue that such id–references, since they cannot affect the objects they refer to, do not cause aliasing, quite contradictory to the definition of aliasing (Hogg, Lea, Wills, deChampeaux, and Holt 1992), but then again, these references are nothing like the ones considered therein. On the other hand one might want to limit how far out in the owner structure an id–reference may be moved, just like the bounds of uniques in External Uniqueness. A bounded id–reference could still be given the bound *world*, and would then be movable without limitations.

Another unresolved matter concerning these id–references is whether they should be weak; that is, if such a reference should not prohibit the garbage collector from disposing of the referred object when the actual reference is voided. Our guess, albeit not very substantiated, is that the id–references should be weak. Although this means that invalid references may linger in system, it might be beneficial, for some reason, to conclude that two such references point (or did point) to the same object, and since no action may be taken on such a reference it really makes no difference whether the object exists or not. Furthermore downcasts of id–references, i.e. turning id–references back to regular references, have to be considered. We believe downcasts should be allowed as long as they are valid owner wise. As long as one can express the type of the cast reference there should be no reason not to allow downcasts. Downcasts, however, bring with them controversy as to whether these id–references should be week. If one has a weak id–reference, there is no way of knowing whether the referred to object really exists, and casting such a reference and reading it may well throw an exception if the object has been disposed of. On the other hand downcasts of id–references may overcome other problems such as interface changes due to the number of owner parameters needed in owner polymorphic methods, discussed in Section 6.1.5.

### 6.1.4 Equality of Uniques

As discussed in Section 6.1.3 comparing uniques is a problem, but unfortunately it does not stop with pointer equality. When comparing objects one is often interested in whether two objects' contents are equal (structural equality) and not whether they actually be the same (pointer equality). With strings, for instance, comparing equality generally means comparing the characters in the string regardless of whether the strings are aliases. As we shall see this poses a problem with External Uniqueness.

Consider the code in Figure 6.2. In this example both strings are borrowed (and thus given temporary owners). Both objects have to be borrowed, since otherwise the object sent as argument would have to be consumed by the method call. This example, however, is erroneous. The `equals` method of `s1` cannot express the type of `s2` since it is not known when writing the method header. Hence the `equals` method has to be owner polymorphic. We depict this in Figure 6.3. This is where we discover the problem. In the string case

```
class Example extends Object
{
    public void method()
    {
        unique:String str1 = "String 1";
        unique:String str2 = "String 2";
        borrow str1 as o1:s1 and str2 as o2:s2 in
        {
            if (s1.equals(s2))
                ...
        }
    }
}
```

**Fig. 6.2.** Erroneous compare of uniques with equals, borrowing both

```
class Example extends Object
{
    public void method()
    {
        unique:String str1 = "String 1";
        unique:String str2 = "String 2";
        borrow str1 as o1:s1 and str2 as o2:s2 in
        {
            if (s2.equals<o1>(s1))
                ...
        }
    }
}

class String
{
    public boolean equals<o outside owner>(o:String other)
    {
        if (this.length() != other.length())
            return false;
        for (int i = 0; i < this.length(); ++i)
            if (this.charAt(i) != other.charAt(i))
                return false;
        return true;
    }
}
```

**Fig. 6.3.** Comparing uniques with owner polymorphic equals

this actually works. However, the reason for that is that the characters of the string are primitive, and hence are not affected by ownership (don't have owners). If the characters were instead objects we would have to have another owner polymorphic `equals` method for the character objects, and if their representation were not primitive this would repeat itself. So for the string case this works, but in any case where equality is dependent on anything not primitive we need another owner polymorphic method. If we had a pure object oriented language, such as `Smalltalk` (Goldberg and Robson 1983), where everything is objects there still might be a solution, though. Take, for instance, characters. If the character objects are immutable singleton objects, i.e. there is only one instance of each character (this may lead to hefty aliasing, but since they are immutable aliasing is innocuous, however it also has the effect that they can never be made part of an object's representation) and they also be ordered so that there is a way of telling which object comes before and after a certain object, it should be possible to construct an owner polymorphic equals method that works even with object primitives.

### 6.1.5 Collections

Collections, such as lists, stacks and binary trees, are commonly used in many applications. Collections often treat their data objects as objects of a common super type. That way collections may be written only once and then reused when needed. In Joline, however, since owner parameters are checked statically, an object may be cast to a super type with fewer owner parameters, but when cast back the owners cannot be checked. Consider the example in Figure 6.4 and Figure 6.5 for a collection example (we cannot know whether *owner* is the correct permission.)

There are two solutions to this problem. Firstly, one could store the owners in each object and at run–time dynamically check the cast operation. When statically checking the owner, as is presently done, the owners can be forgotten about in run–time, since a successful compilation guarantees that the program will run without errors, at least as far as owners are concerned. The second solution is generics , as in `Java` (Bracha, Odersky, Stoutamire, and Wadler 1998) and `C++` (Ellis and Stroustrup 1992). With generics, even though the collection is general and reusable, the programmer may specify which type the collection should contain. This would enable us to specify which owner parameters the objects in the collection should have and thus the cast will be avoided. Potanin et al. recently proposed Featherweight Generic Ownership for genericity and deep ownership (2005). This, of course, does not help the general problem, i.e. that owner parameters are lost when a reference is cast.

```
class A extends Object
{}

class B<a outside owner> extends A
{}

class Example extends Object
{
    public void example()
    {
        this:B<owner> b1 = new this:B<owner>();
        this:A a1 = b1;
        this:B<this> b2 = (this:B) a1; // How to check the
    }                               // permissions of a1?
}
```

**Fig. 6.4.** Casting reference with permissions

```
class List<data outside owner> extends Object
{
    this:ListNode<data> first;
    public void add(data:Object o)
    {
        ...
    }
    public data:Object get(int index)
    {
        ...
        return obj;
    }
}

class Thing<o outside owner> extends Object { ... }

class Example extends Object
{
    this:List<this> list;
    public void method()
    {
        this:Thing<owner> obj = new this:Thing<owner>();
        list.add(obj);
        ...
        obj = (this:Thing<owner>) list.get(0); // *
    }
}
```

**Fig. 6.5.** Collection example

Generics also don’t help the problem with the `equals` method discussed in the previous section. Even if we use generics we still may have to change the interface of the collection class in order to meet different requirements on the amount of owner parameters needed to compare the data objects. To actually solve the problem with polymorphism, dynamic checking of owners is probably the only way to go. The drawback with dynamic checks, of course, is that the compiler cannot in any way assure that the program will run without errors due to invalid cast operations. The positive side with storing owners dynamically is that it is much more flexible and enables such features as reflection.

## 6.2 Observations

This section discusses observations made when programming in `Joline`. These observations are not of errors or flaws, but merely of situations in which problems may arise, if one is not wary.

### 6.2.1 Borrowing Blocks

When borrowing a unique field in `Joline` the field is destructively read to maintain uniqueness. This is a very straightforward way of ensuring that aliasing cannot occur. However, as it turns out it has some effects that might be considered unintuitive, at least to the unwary programmer. Take a look at the code in Figure 6.6. This, albeit somewhat artificial, is not a very uncommon situation in a general program (in our system this happened a few times.) In `method2` an exception is thrown because a method is called on null. This is exactly what should happen in this case and this behavior might be the desired behavior. However, there are other ways of dealing with the use of borrowed fields. One way could be to use some kind of lock, like with concurrency, which prevents the use of a borrowed field until it is reinstated. There is, however, as with concurrency, no way of ensuring that the first borrow will ever exit, and thus also no way of ensuring that the lock will ever be released. A different approach might be to allow reading of an object during a borrow, while disallowing any writing operations (see Section 6.1.3 for a brief discussion on read–only references.)

### 6.2.2 Initialization of State

Initializing a newly created object is commonly done by passing arguments to the constructor. However, it is not uncommon that the state be updated using some setter method at a later time. When this is done on a unique object

```
class Example extends Object
{
    unique:Object obj = new unique:Object();
    public void method1()
    {
        borrow obj as o1:o in
        {
            method2(o.hashCode());
        }
    }
    public void method2(int hash)
    {
        borrow obj as o2:o in
        {
            if (hash == o.hashCode()) // Throws
                {                     // NullPointerException
                    ...
                }
        }
    }
}
```

**Fig. 6.6.** Use of borrowed field

complications may arise. Since uniques require a borrow before a message is sent, one must be wary of one's owners. Say, for instance, that a borrower in a library system is to be aware of all libraries in the system (much like a phone book.) Then, because of the previously discussed issue with global data, borrower objects are equipped with a method that sets the library list reference. If the library list is owned by the same owner as the borrower the, perhaps, most intuitive thing to do is to simply express the library list as a *owner*:`LibraryList`. This is where the problem arises. Since the borrower reference is borrowed, during the call to the setter method, the owner changes. Thus the library list and the borrower no longer have the same owner, and thus the compiler, quite correctly, will report an error. The solution to this is to use an additional class permission to describe the owner of the list. The owner of the borrower object still will change, but since the owner of the library list now is not called *owner* it does not change during the borrowing, thus the problem is solved. The code in Figure 6.7 and 6.8 depicts the problem and solution respectively.

It may well be that using additional permissions for objects received from methods is a working pattern, that should be adhered to whenever designing a class. However, this is not clear, and more research is needed.

```
class Example extends Object
{
    this:LibraryList libs = new this:LibraryList();
    public void method()
    {
        unique:Borrower b1 = new Borrower();
        borrow b1 as o1:b in
        {
            b.setLibraries(libs);
        }
    }
}
class Borrower extends Object
{
    owner:LibraryList libs = null;
    public void setLibraries(owner:LibraryList libs)
    {
        this.libs = libs;
    }
}
```

**Fig. 6.7.** Erroneous library example

```
class Example extends Object
{
    this:LibraryList libs = new this:LibraryList();
    public void method()
    {
        unique:Borrower<this> b1 = new Borrower<this>();
        borrow b1 as o1:b in
        {
            b.setLibraries(libs);
        }
    }
}
class Borrower<libOwner outside owner> extends Object
{
    libOwner:LibraryList libs = null;
    public void setLibraries(libOwner:LibraryList libs)
    {
        this.libs = libs;
    }
}
```

**Fig. 6.8.** Library example with data owner

## 6.3 Discussion

Most of the issues described in the previous section are due to deep ownership. Some of these issues are not helped by the added flexibility of External Uniqueness, such as the problem with shared (or global) data. Other issues are due mostly to the addition of uniqueness, such as comparing of unique objects. It seems that the addition of uniqueness to deep ownership gives more flexibility in some cases, while introducing new problems in others. However, the net outcome of adding uniqueness to deep ownership seems to be positive.

The results yielded by our research seem to be mostly related to deep ownership. This is hardly unexpected. The current External Uniqueness proposal is built on top of deep ownership for a number of reasons, where the most crucial is the ability of deep ownership to distinguish between the inside and outside of an object (its representation.) It is conceivable that there be other ways to achieve this, than by the use of deep ownership. Recently Liu and Smith (2005) presented `Classages`, a novel interaction–centric object–oriented language, which they claim has the ability to distinguish the inside and outside of an object. It should thus be possible to implement External Uniqueness on top of Classages. Deep ownership, however, has other benefits in the use of owners, and using Classages as a base for External Uniqueness would require redesigning the existing solutions for borrowing and scoped regions. Without ownership, borrowing requires type system additions to preserve the uniqueness invariant, as discussed in Section 2.2, most likely losing the orthogonality of the current proposal. Scoped regions would probably also be hard to implement, although some region–based memory management technique (Talpin and Jouvelot 1992; Talpin and Tofte 1997; Grossman, Morrisett, Cheney, Hicks, Jim, and Wang 2002) might be possible to adopt to fit this purpose.

### 6.3.1 Implications of External Uniqueness on Design

Our experience from designing and implementing non–trivial programs in External Uniqueness is that along with managing aliasing External Uniqueness helps conceptually by offering a very real–world–like system model. Ownership helps to clearly define who owns what, and uniqueness mimics the real world closely by ensuring that objects that are unique, such as books in our system, cannot be aliased but have to be moved from one variable to another.

Cele and Stureborg (2004) pointed out that the design phase becomes much more important when using ownership, and that late found design flaws may be very difficult, or expensive, to rectify. This is very much our experience as well.

### 6.3.2 External Uniqueness, Useful or Useless?

Ownership types, as pointed out by a number of researchers (Clarke, Noble, and Potter 1998b; Clarke, Noble, and Potter 1999; Noble 2000; Liskov, Boyapati, and Shrira 2003), is often too restrictive and certain common constructs are impossible to implement. The addition of uniqueness to ownership types, which constitutes External Uniqueness, however, loosens some of this restrictiveness, and enables more flexibility. Objects may be moved between owners (as long as they are unique), which makes external creation and passing of representation objects possible, without breaking encapsulation, by ensuring there be no residual aliasing. We believe this makes External Uniqueness possible to use in real life applications, as opposed to most other alias management proposals, which often are too restrictive for real life use, or too permissive to actually manage aliasing. That being said, we still find External Uniqueness to be a bit too immature for implementation in an industrial context, and apart from what we present here, there may well be other issues passed undetected by this thesis. We believe more research is needed to fully develop External Uniqueness and `Joline`.

# 7 Conclusions

## 7.1 Critique

Naturally, we cannot claim to have found and presented all aspects of realizing and using External Uniqueness from a single case study. However as there has been very little research in this area we believe that these results are interesting, even though they should only be generalized carefully, if at all. However, the system we have implemented covers most of the aspects from the three case studies by Cele and Stureborg (2004), with similar results. Also, from their findings, program size does not seem to matter.

Based on this, we do not believe that a larger number of systems would change our findings dramatically.

Our implementation of the `Joline` compiler is based on the formalization in Wrigstad's licentiate thesis (2004). We cannot eliminate the possibility that there be errors in our implementation, which may pose a threat to the reliability of our work. However, we have continuously and thoroughly tested the compiler and we are confident that it implements the formal specification correctly.

We believe our additions to the `Joline` specification, such as strings and string literals are well understood and appear to be working fine. However, default owners on e.g. string literals are more or less ad–hoc and not very well researched. We cannot claim that these additions will not have unforeseen implications elsewhere, however up til now there has been no such evidence.

## 7.2 Programming With Ownership Types

Although Ownership Types has been around for some years now there is very little experience in the practical use of it, as with External Uniqueness. To the best of our knowledge the only real practical programming of non–trivial programs using Ownership Types is that of Cele and Stureborg (2004) and our work presented in this thesis. Cele and Stureborg performed a study on the implications of Ownership Types on the development process.

Our impression of Ownership Types after having used it in practice is not very different from the results in Cele and Stureborg's masters thesis. We think the stronger encapsulation and the explicit ownership is beneficial conceptually by offering a system model which maps well to reality. However, as pointed out by Cele and Stureborg the importance of the design phase increases, and it is possible that iterative design processes, like Extreme Programming (as explained by Beck (1999)), may be much harder to perform.

External Uniqueness, with its uniqueness properties, we think maps even closer to reality, since objects that are unique must really be moved and not just aliased, which probably is very common today, in a contemporary object–oriented language, such as `Java` or `C++`.

## 7.3 Future work

Design patterns (Gamma, Helm, Johnson, and Vlissides 1994) are considered good practices in object–oriented design, and how these are affected by External Uniqueness is not known. There might well be certain patterns that are no longer possible to implement due to the restrictions of Ownership Types or, rather, that require an owner structure that loses the advantages of the stronger encapsulation offered by Ownership Types. This matter should be further inquired into.

Several proposals on read–only references have been presented over the years, but as discussed earlier, these suffer from observational and representational exposure (Boyland 2005b). Boyland proposes fractional permissions as a possible solution (2003, 2005a). This work is far from finished and it is uncertain whether it will bear fruit. The concept of read–only references, however, we think is very interesting, both as an alias management technique of its own, and as a possible implementation of id–references and borrowed fields in `Joline`; thus we believe read–only references merit further exploration.

The work on our `Joline` compiler should be continued. As described in this thesis there are features we have found to be needed, albeit not present in the formal specification of `Joline`. These features include for instance arrays. How features such as arrays would affect the `Joline` language is not known and should be further looked into. The same holds true for id–references, as described in Section 6.1.3 and multiple class–instances belonging to different representations. Their implementation in `Joline` should be trivial, but more work is required to fully understand their implication.

# References

[2002] Aldrich, J., V. Kostadinov, and C. Chambers (2002, August 02). Alias annotations for program understanding.

[1997] Almeida, P. S. (1997, June). Balloon types: Controlling sharing of state in data types. In M. Akşit and S. Matsuoka (Eds.), *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, Volume 1241 of *Lecture Notes in Computer Science*, pp. 32–59. New York, NY: Springer-Verlag.

[1995] Baker, H. G. (1995, January). 'use-once' variables and linear objects-storage management, reflection and multi-threading. *SIGPLAN Notices 30*(1), 45–52.

[1999] Beck, K. (1999). *Extreme Programming Explained: Embracing Change*. Addison-Wesley.

[2004] Birka, A. and M. D. Ernst (2004, June 28). A practical type system and language for reference immutability.

[2001] Boyapati, C. and M. Rinard (2001, November). A parameterized type system for race-free Java programs. *ACM SIGPLAN Notices 36*(11), 56–69. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[2001] Boyland, J. (2001, May). Alias burying: Unique variables without destructive reads. *Software—Practice and Experience 31*(6), 533–553.

[2003] Boyland, J. (2003). Checking interference with fractional permissions. In R. Cousot (Ed.), *Static Analysis: 10th International Symposium*, Volume 2694 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, New York, pp. 55–72. Springer.

[2005a] Boyland, J. (2005a, June). Checking data and code movement using permissions. Presentation slides from Types For Tools, Schloss Dagstuhl.

[2005b] Boyland, J. (2005b, June). Why we should not add `readonly` to Java (yet). 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'2005).

[2001] Boyland, J., J. Noble, and W. Retert (2001, June). Capabilities for sharing. In J. L. Knudsen (Ed.), *ECOOP 2001 — Object-Oriented Programming: 15th European Conference, Budapest, Hungary*, Volume 2072 of *Lecture Notes in Computer Science*, Berlin, pp. 1–27. Springer-Verlag.

[1998] Bracha, G., M. Odersky, D. Stoutamire, and P. Wadler (1998, October). Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. OPPSLA'98*, pp. 183–200.

[1971] Brooks, P. (1971). *The pursuit of wilderness*. Boston, Houghton Mifflin. ISBN: 0395120934.

[2004] Cele, G. and S. Stureborg (2004). Ownership types in practice. Master's thesis, Department of Computer and Systems Scieces at Stockholm University.

[2001] Clarke, D. (2001, July 12). *Object Ownership and Containment*. Ph. D. thesis, School of Computer Science and Engineering University of New South Wales, Australia.

[2002] Clarke, D. and S. Drossopoulou (2002, November). Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, Volume 37(11) of *ACM SIGPLAN Notices*, pp. 292–310. ACM.

[1998a] Clarke, D., J. Noble, and J. Potter (1998a, September 10). The ins and outs of objects.

[2003] Clarke, D. and T. Wrigstad (2003, July). External uniqueness is unique enough. In L. Cardelli (Ed.), *ECOOP 2003 — Object-Oriented Programming: 17th European Conference, Darmstadt, Germany*, Volume 2743 of *Lecture Notes in Computer Science*, Berlin, pp. 176–200. Springer-Verlag.

[1998b] Clarke, D. G., J. Noble, and J. M. Potter (1998b, July 20). Ownership types for flexible alias protection.

[1999] Clarke, D. G., J. Noble, and J. M. Potter (1999, July 20). Who's afraid of ownership types?

[1992] Ellis, M. A. and B. Stroustrup (1992). *The Annotated C++ Reference Manual*. Reading: Addison-Wesley.

[1994] Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1994). *Design Patterns*. Addison-Wesley.

[1983] Goldberg, A. and D. Robson (1983). *Smaltalk-80: The Language and Its Implementation*. Reading, Massachusetts: Addison-Wesley.

[2000] Gosling, J., B. Joy, G. Steele, and G. Bracha (2000). *The Java Language Specification Second Edition*. The Java Series. Boston, Mass.: Addison-Wesley.

[2002] Grossman, D., G. Morrisett, J. Cheney, M. Hicks, T. Jim, and Y. Wang (2002, April 21). Region-based memory management in cyclone.

[2001] Grothoff, C., J. Vitek, and J. Palsberg (2001, July 20). Encapsulating objects with confined types.

[1991] Harms, D. E. and B. W. Weide (1991, May). Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering 17*(5), 424–435.

[1991] Hogg, J. (1991, November). Islands: aliasing protection in object-oriented languages. *ACM SIGPLAN Notices 26*(11), 271–285.

[1992] Hogg, J., D. Lea, A. Wills, D. deChampeaux, and R. Holt (1992, June 01). The geneva convention on the treatment of object aliasing.

[1996] Kent, S. and J. Howse (1996, January). Value Types in Eiffel. In *Proceedings of TOOLS Europe'96 (TOOLS 19)*, pp. 145–160. Prentice Hall.

[1988] Kernighan, B. W. and D. M. Ritchie (1988). *The C Programming Language, ANSI C* (2. ed.). Englewood Cliffs, New Jersey, USA: Prentice Hall.

[2001] Kniesel, G. and D. Theisen (2001, May). JAC — access right based encapsulation for Java. *Software - Practice and Experience 31*(6), 555–576.

[2003] Liskov, B., C. Boyapati, and L. Shrira (2003, April 20). Ownership types for object encapsulation.

[2005] Liu, Y. D. and S. Smith (2005, Oct). Interaction-based programming with classages. In *Proceedings of the 20th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*.

[1982] MacLennan, B. J. (1982, December). Values and objects in programming languages. *ACM SIGPLAN Notices 17*(12), 70–79.

[2000] Müller, P. and A. Poetzsch-Heffter (2000). A type system for controlling representation exposure in Java. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter (Eds.), *Formal Techniques for Java Programs*. Technical Report 269, Fernuniversität Hagen. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[1999] Nielson, F., H. R. Nielson, and C. Hankin (1999). *Principles of Program Analysis*. Springer-Verlag.

[2000] Noble, J. (2000, August 25). Iterators and encapsulation.

[1998] Noble, J., J. Vitek, and J. Potter (1998). Flexible alias protection. *Lecture Notes in Computer Science 1445*, 158–??

[2003] Nystrom, N., M. R. Clarkson, and A. C. Myers (2003). Polyglot: An extensible compiler framework for Java. *Lecture Notes in Computer Science 2622*, 138–152.

[2002] Potanin, A. and J. Noble (2002, October 20). Checking ownership and confinement properties.

[2005] Potanin, A., J. Noble, D. Clarke, and R. Biddle (2005, June). Featherweight generic ownership. 7th Workshop on Formal Techniques for Java-like Programs - FTfJP'2005.

[2001] Skoglund, M. and T. Wrigstad (2001). A mode system for readonly references. In kos Frohner (Ed.), *Formal Techniques for Java Programs*, Number 2323 in Object-Oriented Technology, ECOOP 2001 Workshop Reader, Berlin, Heidelberg, New York, pp. 30–. Springer-Verlag.

[1992] Talpin, J.-P. and P. Jouvelot (1992, January 31). Polymorphic type, region and effect inference.

[1997] Talpin, J.-P. and M. Tofte (1997, January 31). Region-based memory management.

[2001] Vitek, J. and B. Bokowski (2001). Confined types in Java. *Software — Practice and Experience 31*(6), 507–532.

[1990] Wadler, P. (1990, May 20). Linear types can change the world!

[2004] Wrigstad, T. (2004, aug). *External Uniqueness – A Theory Of Aggregate Uniqueness For Object-Oriented Programming*. Ph. D. thesis, Department of Computer and Systems Siences, Stockholm University.

[2005] Zhao, T., J. Palsberg, and J. Vitek (2005). Type–Based Confinement. *Journal of Functional Programming 15*(6), 1–46. Accepted for publication.