External Uniqueness is Unique Enough

Dave Clarke

Tobias Wrigstad

institute of information and computing sciences, utrecht university

technical report UU-CS-2002-048

www.cs.uu.nl

External Uniqueness is Unique Enough

Dave Clarke[1] and Tobias Wrigstad[2]

[1] Institute of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands. dave@cs.uu.nl
[2] Deptartment of Computer and Systems Sciences
Stockholm University/KTH, Stockholm, Sweden. tobias@dsv.su.se

Abstract. *External uniqueness* is a surprising new way to add unique references to an OOPL. The idea is that an externally unique reference is the only reference into an aggregate from outside the aggregate. Internal references which do not escape the boundary of the aggregate are innocuous and therefore permitted. Based on ownership types, our proposal not only overcomes an abstraction problem from which existing uniqueness proposals suffer, it also enables many examples which are inherently not unique, such as a unique reference to a set of links in a doubly-linked list, without losing the benefits of uniqueness.

1 Introduction

Two essentially different approaches to managing aliasing in object-oriented programming exist. On one hand sits unique or alias-free references. These are based on a very simple idea: any variable or field annotated with the keyword `unique` contains the only reference to the object it holds, otherwise it contains `null` [26, 31, 9, 11, 8, 1]. Apart from helping reason about programs, unique references safely enable, for example, idioms essential in concurrent programming such as the transfer of ownership pattern [28], and help enforce software protocols [16]. In all cases, the same notion of uniqueness applies.

Unfortunately, all extant uniqueness proposals suffer from an *abstraction problem* which we identify in this paper: *as software evolves, programs which use uniqueness are forced to change their interfaces when purely internal implementation changes are made.* The interface changes propagate through a program.

The second approach to managing aliasing employs alias encapsulation, as exemplified by ownership types [15]. Simply put, these impose a form of object-level privacy by preventing objects (rather than just fields) from being accessed outside of their enclosing encapsulation boundaries [15, 32, 13, 8, 6, 14, 1]. Ownership types have been employed for reasoning about programs [14, 32], for alias management [15, 33], in program understanding [1], to eliminate data-races [8] and deadlocks [6] from concurrent programs, and to enable safe lazy updates in object-oriented databases [7].

Existing attempts to unify uniqueness and ownership typing [8, 1] unfortunately offer little additional benefit from their combination, while perpetuating the abstraction problem.

We believe that the key to the abstraction problem is that the distinction cannot be made between external references and innocuous, internal references which do not escape encapsulation boundaries. There are essentially two kinds of ownership types, based on the degree of protection provided. The stronger form, namely those with deep ownership, [15, 14] provide the machinery necessary to make this distinction. These form the basis for our proposal.

In this paper, we introduce a different kind of uniqueness called *external uniqueness*. External uniqueness loosens the conventional uniqueness constraint, requiring that there be

only one reference to an aggregate from outside of the aggregate, without limiting the number of references to the aggregate from its inside. Interestingly, it turns out that the externally unique reference is the only active reference to an object, so it is effectively unique.

Our proposal not only overcomes the abstraction problem, it also enables a number of interesting examples which result from the synergy between uniqueness and deep ownership: we enable, for example, the transfer of entire aggregates between objects, or even the combination of the encapsulated representation of data structures such as doubly-linked lists, even in the presence of aliasing within such aggregates, without leaving any unwanted aliases which would break encapsulation. Above all, we do so for a class-based object-oriented programming language with subtyping.

Outline Our paper is organised as follows. In Section 2 we describe the abstraction problem with uniqueness, and point to a way around it. As ownership types are the key to a solution we review them in Section 3. We then describe external uniqueness in Section 4, before illustrating its power through example in Section 5. Section 6 covers the essence of our formalisation. We discuss our proposal in Section 7 and related work in Section 8, before concluding in Section 9.

2 Challenging Uniqueness

Existing approaches to uniqueness in object-oriented programming are broken. We now outline how the two different approaches both suffer from the same abstraction problem, before indicating a way out of the mess.

2.1 An Abstraction Problem

To add unique references to an OOPL, one must consider how a class treats its instances internally via `this` (or self). Approaches in the literature reflect the treatment of `this` in a class' interface in one of two ways:

via class annotation classes are divided into two kinds, those which may assign `this` internally, and those which do not. Only instances of the latter may be referenced uniquely [31].

via method annotation methods are annotated to indicate that they may consume `this` [26, 9]. Calling such a method requires (at least conceptually) that its target be destructively read.

Proposals combining ownership types and uniqueness follow suit: Boyapati and Rinard [8] adopt the first approach, whereas Aldrich, Kostadinov and Chambers [1] adopt the latter.

In both cases, a problem surfaces when the implementation of a class changes the way it uses `this`. For concreteness, assume that we have the following class:

```
class BlackBox {
   void xyzzy() { .. }
}
```

and a variable (or field):

```
unique BlackBox bb;
```

When we change the implementation of the `BlackBox` class so that the `xyzzy` method assigns `this` internally, we are *forced* to change `BlackBox`'s interface.

Using class annotations we would have to modify `BlackBox` to indicate that its instances cannot be referred to uniquely: `class neverunique BlackBox`. As a result, variable declarations such as the one above would no longer be valid, and would have their uniqueness stripped. The consequence is that all destructive reads of `BlackBox` objects throughout the entire program would have to be changed to ordinary reads, perhaps with destructive read implemented manually.

When using method annotations, we would have to modify the `xyzzy` method to indicate that it consumes `this`, such as `void xyzzy() consumes { .. }`. The call `bb.xyzzy()` may create an internal reference to its target, requiring that the target `bb` be consumed to preserve uniqueness. The consequence here is even more drastic, as the semantics of method call changes: calls to this method suddenly consume their target, whereas in the original program they did not.

In both case, a purely internal change to the implementation of `BlackBox` forces changes to its interface, which propagate through the program — either statically or dynamically. Not only does this introduce the opportunity for errors, since the behaviour of a program changes, it means that objects cannot be treated like black boxes. Thus extant uniqueness proposals break *abstraction*.

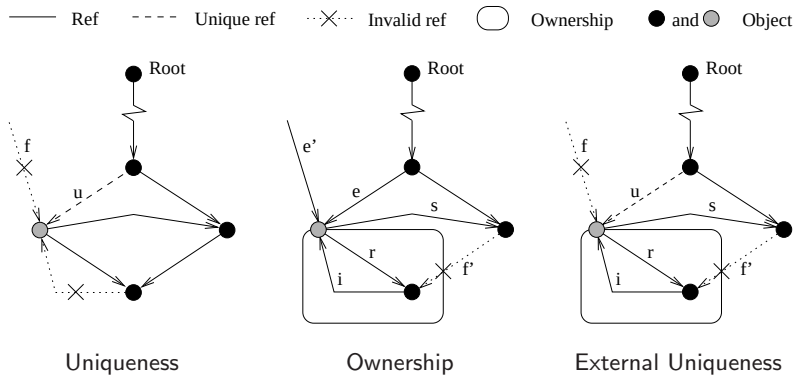
2.2 Distinguishing internal and external references

The abstraction problem occurs, we believe, because the distinction cannot readily be made between internal and external references. For example, traditional object-oriented programming languages cannot distinguish the references between the links of a linked list, which are internal to the data structure, from references that go *into* a data structure from *outside* of it, such as the reference from the handle object to the first link.

A purely internal reference to an object which has only one external reference cannot be used by objects other than the holder of the external reference. This means that no changes to the object can be made via the internal reference violating the "uniqueness" of the external reference, since the internal reference is only accessible once already inside the object. Thus, purely internal references are innocuous. Their existence should not affect how an aggregate is viewed externally; they ought to be preserved to maintain the internal consistency of an aggregate. When this is not the case, knowledge of internal reference behaviour exposes an object's implementation details and thus violates abstraction, as we have shown.

Fortunately, the desired distinction can be made in a programming language with ownership types, such as was originally proposed by Clarke, Potter and Noble [15]. This form of ownership types provide strong protection against external aliasing of an object's internals, enabling a strong notion of aggregate object. Technically, each object has an owner through which all external access paths into the object must pass, meaning that owners are dominators in the object graph (take a peek at Figure 1). The resulting anti-aliasing guarantees are compatible with the containment implicit in object-oriented programming.

Based on the machinery of ownership types, we propose a new take on uniqueness called *external uniqueness*. External uniqueness restricts the external references to an object to be at most one, while permitting arbitrary internal aliasing. Consequently, external uniqueness refines the object graph property underlying ownership types from dominating nodes to dominating edges (see again Figure 1). Using external uniqueness, we treat `this` non-uniquely and allow it to arbitrarily assigned internally. Methods cannot steal `this`, and hence need not be annotated, keeping the syntactic overhead down and avoiding clutter in interfaces.



Reference kinds: u – (externally) unique. f, f' – invalid (f breaks uniqueness, f' breaks deep ownership). s – sibling. e, e' – external to grey object. i – internal. r – representation.

Fig. 1. Comparing uniqueness, deep ownership and external uniqueness

Furthermore, under very mild conditions, we need not annotate classes,[1] since instances from every class can be referred to uniquely. Thus external uniqueness can solve the abstraction problem.

Since ownership types are crucial to our proposal, we shall now review what they do and how they do it.

3 How Ownership Types Work

Ownership types package together into a class-based type system a number of conditions which together act locally to restrict the global structure of object graphs. The underlying idea is very simple.

Objects have owners and can be the owners of other objects. Ownership forms a tree, where an object is *inside* ($\prec^*$) the object which owns it. An additional owner called `world` forms the root of the tree, hence all objects are inside `world`. Finally, there is a condition which governs whether one object can refer to another:

$$\alpha \rightarrow \beta \Rightarrow \alpha \prec^* \mathsf{owner}(\beta)$$

This says that object α can only access an object β whose owner is outside of itself, or alternatively, an object cannot be accessed from outside of its owner. The owner can be seen as the permission required to access an object, and an object's position in the inside relation determines whether the object has enough permission. A nice little theorem [34, 13] states that if we have all of these conditions, then an object's owner will be on all paths from the root of the graph to that object, which is to say that an object's owner is its *dominator*. We call this property *owners-as-dominators*, and type systems which enjoy it are said to offer *deep ownership*.

[1] Assigning `this` to a static variable prohibits unique references to the objects of its class. This behaviour is rare, so we will not even consider it here.

Deep ownership is illustrated in the centre picture in Figure 1. If one considers the objects an object owns to be nested inside it, as depicted by the rounded box, then deep ownership can be stated as: *no reference can pass through an object's boundary from outside to the inside.*

The tricky part is realising these ideas in a class-based OOPL. Firstly, owners become a syntactic category and are indicated in an object's type. Types take the form: $p_0\ c\langle p_{i\in 1..n}\rangle$, where p_0 is the owner of objects of that type, and each $p_{i\in 1..n}$ is a binding for the parameters of the class. Constraints on owners are recorded in the type system using both inside ($\prec^*$) and its converse *outside* ($\succ^*$). These constraints are specified in class headers (and elsewhere) and must be satisfied when forming types.

Outside of a class there is only one owner, the global `world`. Within the body of a class, `this` is used to denote objects owned by the current instance. These objects are directly inside the current instance — objects inside these objects are also inside, but inaccessible. A class implicitly takes a parameter for the owner of its instances, denoted `owner` within the class body — objects with this owner are called *siblings*. In addition, the class may have parameters, such as `a` and `b` in the following:

```
class c<a, b *> a> {
   // valid owners: this, owner, world, a, b
   // relationship: this <* owner <* a <* b <* world
}
```

External objects have `owner` or some parameter as owner. In addition to these owners, additional owner variables may be introduced via owner polymorphic methods, scoped regions, or borrowing. Whether these are internal or otherwise is determined from their relationship (if any) with `this`.

That takes care of owners, now for the nesting between them. Firstly, every owner is inside `world`. Within a class body we have `this` $\prec^*$ `owner`, and `owner` $\prec^*\ \alpha$ for each of the class' parameters α. Since the ordering of owners is required when forming types, we also can specify the ordering, such as the constraint `b` $\succ^*$ `a` above, in the class header.

The final constraint required is that the owner part of a type be preserved through subtyping, as this acts as the permission governing access to the object.

More detailed descriptions of ownership types are available in the literature [13, 14]. From now on, when we refer to ownership types, we assume a deep model of ownership, and we say unique to indicate external uniqueness, adding the appropriate qualifiers either where required or for emphasis.

4 A Tour of External Uniqueness

We now present our proposal for external uniqueness. It is a minor extension to ownership types with major consequences. For concreteness, we work in the context of a core programming language called Joline, much of which ought to be familiar (see Table 1). We first describe external uniqueness, then operations required to support it, and address a few technicalities required to maintain soundness of the system, before diving into examples in the next section.

4.1 External Uniqueness in a Nutshell

A reference to an object is *externally unique* if it is the only reference from outside an object to it. Aliasing from inside the object is still permitted, because such references form a part

$c \in$ **ClassName**, $f \in$ **FieldName**, $m \in$ **MethodName**, $x, y \in$ **TermVar**, $\alpha \in$ **OwnerVar**.

$P \in$ **Program**	$::=$	$class_{i \in 1..n}\ s\ e$
$class \in$ **Class**	$::=$	$\texttt{class}\ c\langle \alpha_i\ \mathsf{R}_i\ p_{i \in 1..m}\rangle\ \texttt{extends}\ c'\langle p'_{i' \in 1..n}\rangle\ \{\ fd_{j \in 1..r}\ meth_{k \in 1..s}\ \}$
		where R is either $\prec^*$ or $\succ^*$.
$fd \in$ **Field**	$::=$	$t\ f = e;$
$meth \in$ **Method**	$::=$	$\langle \alpha_i\ \mathsf{R}_i\ p_{i \in 1..m}\rangle\ t\ m(t_i\ x_{i \in 1..n})\ \{\ s\ \texttt{return}\ e\ \}$
$lval \in$ **Lvalue**	$::=$	$x \mid e.f$
$e \in$ **Expr**	$::=$	$\texttt{this} \mid lval \mid lval\texttt{--} \mid \texttt{new}\ t \mid \texttt{null} \mid e.m\langle p_{j \in 1..m}\rangle(e_{i \in 1..n})$
$s \in$ **Stat**	$::=$	$\texttt{skip;} \mid t\ x = e; \mid e; \mid lval = e; \mid s\ s \mid \texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}$
		$\mid (\alpha)\ \{\ s\ \} \mid \{\ s\ \} \mid \texttt{borrow}\ lval\ \texttt{as}\ \langle \alpha\rangle\ x\ \{\ s\ \}$
$p, q \in$ **Owner**	$::=$	$\texttt{this} \mid \alpha \mid \texttt{world} \mid \texttt{owner} \mid \texttt{unique}$
		$\mid \texttt{unique}_p$ (*in elaborated language only*)
$t \in$ **Type**	$::=$	$p\ c\langle p_{i \in 1..n}\rangle$

Table 1. Syntax of Joline

of the aggregate objects' implementation. External uniqueness takes uniqueness, but only applies it externally, using the distinction between the inside and outside of an object offered by ownership types. Figure 1 illustrates the distinction between uniqueness, deep ownership, and external uniqueness. The graph-theoretic property that external uniqueness enjoys is a refinement of the owners-as-dominators property. An externally unique reference corresponds to a *dominating edge* to its target, which is an edge that must occur on all paths from the root of an object graph to the target. This can be seen in the third picture in Figure 1: the dotted edge u denotes an externally unique reference. Notice that internal references to the grey object are still permitted. The dominating edge property means that if the dominating edge u is removed, then all internal objects (within the rounded box) become inaccessible from the rest of the system. Contrast this with ownership typing in the centre picture, where the removal of the grey dominator object would result in its internal objects becoming inaccessible.

Externally unique references are denoted using types such as `unique` $c\langle p_{i \in 1..n}\rangle$. The unique annotation can only appear in the owner position, and thus no $p_{i \in 1..n}$ may be "`unique`." (For technical reasons, we have types like $\texttt{unique}_p\ c\langle p_{i \in 1..n}\rangle$ — see Section 4.3.) As ownership types maintain the dominators property, to obtain external uniqueness we need only add machinery to ensure the uniqueness of references of type `unique` $c\langle p_{i \in 1..n}\rangle$ whenever viewed externally.

4.2 Operations on External Uniques

Unique values, fields and variables are affected in two ways: *movement* and *borrowing*. Movement is simplest, so we discuss it first.

Movement Movement allows an externally unique reference to be moved from one field or variable into another, possibly losing its uniqueness along the way. Movement is the only operation which can be directly performed on the contents of a unique field or variable. To preserve uniqueness the original value must no longer be accessible through its source after the movement. Either the source must be nullified, the approach we take, or a technique such as alias burying, which statically ensures that all aliases to an object are dead when a unique variable or field is read [9], must be applied.

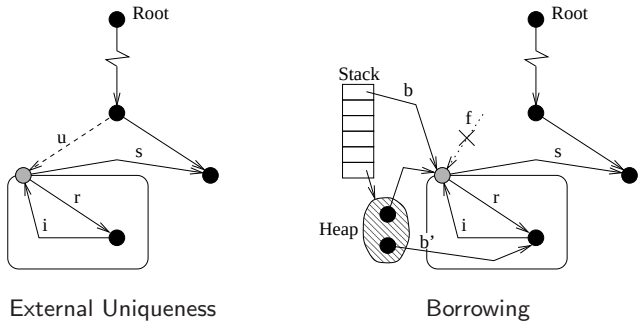


Fig. 2. Mediating between external uniqueness and borrowing — b is the original borrowed reference. b, b' are only valid during the borrowing. (Stack grows downwards)

Moving is really a compound operation which combines a means for obtaining a unique value with a means for consuming it. A unique value, as opposed to a unique field or variable, can be obtained through object creation, the destructive read of a unique field or variable (*lval*`--`), or as the result of a method call. Destructive reads are made explicit to simplify the formal account of the language. Thus a programmer writes `x = y--` or `return x--`, for example, instead of `x = y` or `return x`, respectively. Ultimately, a unique value is consumed by assigning it to a field or variable, or by passing it as an argument to a method.

The presence of internal references causes no problems when moving, because the dominating edge property guarantees that these are all (effectively) moved. When consuming a unique value, however, we must ensure that deep ownership be maintained. The appropriate constraints on movement are incorporated into the subtype relation.

Borrowing Performing an operation other than the moving of a unique reference, such as accessing a field or calling a method, requires that it first be borrowed. The borrowing construct creates a non-unique reference, x, to the borrowed entity, *lval*, for a limited lexical scope, s:

`borrow (`*lval*`) as` $\langle\alpha\rangle$ x `{` s `}`

For example, we could call the `add` method of the object in variable `unique B bus` as follows: `borrow bus as <bo> b { b.add(...); }`.

To make this a little more concrete, consider the phases of borrowing depicted in Figure 2. The left-hand picture indicates the state of play before the borrowing occurs. Initially, all access paths from the root to the grey object and its inside contain the unique reference u, which is inactive, and thus the objects are also inactive. During the borrowing, right-hand picture, the original reference is placed in variable x which can be treated non-uniquely. The type of x is not unique, having owner α, which must always be a "fresh owner", for the duration of the borrowing, and we can access its fields and methods, pass it to methods, or even store it in subsequent stack frames or on locally created heaps. Since α is only available in the scope of the borrowing, no references with types containing α are active at then end of the borrowing. Only internal aliases remain, and the situation returns to the original inactive state.

To some degree, borrowing exists primarily to facilitate type checking, by providing a construct to mediate between uniqueness and non-uniqueness. Ownership typing ensures that

no reference escapes from the borrowing construct via some back door. We must now consider how to maintain external uniqueness, since one could simply walk in the front door and move the reference contained in *lval* (indicated as u in Figure 2). There are essentially three approaches:

do nothing rather than invalidate the original *lval*, we could simply weaken the definition of uniqueness, permitting both the reference in the *lval* and the borrowed references, and even allowing movement of the *lval* underfoot. We dismiss this case for now, but discuss its merits in Section 7.

destructive we could nullify the *lval* during borrowing, and then:
- simply restore the original contents of *lval* when the borrowing ceases;
- restore the final contents of the borrowing variable x at the end of the borrowing. Restoring the initial value is consistent with conventional uniqueness, whereas enabling a different reference into the same aggregate to be reinstated is consistent with external uniqueness; or
- rather than simply nullify *lval*, we could record the state of its contents. The three possible states are: *available*, *null*, and *borrowed*, indicating that *lval* contains something, nothing, or is disabled due to some borrowing. In the presence of multiple threads, more states could be added to indicate whether a different thread is borrowing the reference.

alias burying the last possibility is to employ alias burying, as mentioned above. This would ensure that when *lval* is read that all aliases are unusable [9]. Alias burying eliminates the need for destructive reads, but unfortunately comes at cost. As it is based on program analysis, its strength is sensitive to the underlying analysis. To achieve modular checking, classes must be further annotated to indicate which unique fields are read [10]. This may well reintroduce the abstraction problem. Furthermore, we do not know whether aliasing burying works in a multi-threaded setting.

In the last two cases, there is no active reference to the target object extracted from *lval*. Thus, the combination of a borrowing mechanism and ownership types ensures that an externally unique reference is the only active reference to its target, hence the title of this paper. We proceed with the second variation of the destructive approach, preventing a reference such as f in Figure 2. This keeps our formal system simple, while retaining a strong definition of uniqueness.

A drawback of destructive reads is that it precludes simultaneous non-conflicting operations on unique references, *e.g.,* allowing read-only methods on unique references even during a borrowing (not necessarily a good thing).

4.3 Virtual owners

Moving an object to a new location could result in residual aliasing of the internals of its original location, thus violating the invariants of deep ownership and external uniqueness. To avoid this problem, all occurrences of `unique` have an associated *virtual owner* and appear in the formal system as $\texttt{unique}_p$. The virtual owner acts as a bound on the outwards movement of unique references – a unique reference may only be moved to variables inside its virtual owner. Unlike an ordinary owner, the virtual owner may change when a reference moves, *i.e.,* transfer of ownership.

Appendix B defines a function which elaborates programs with virtual owners. Other choices are possible. In our compiler the virtual owner is under programmer control: if the default is not appropriate, a programmer may write: `unique[q] C` to indicate that the virtual owner is `q`. Choosing `q` requires a trade-off: an outer `q` enables more movement, but limits what the object can access (*i.e.,* what ownership parameters can appear in its type); whereas an inner `q` would enable less movement, but permits more access.

4.4 Do constructors return externally unique references?

Clearly, the act of creation results in a unique object, so we must consider what could go wrong in a method call, which is effectively what a constructor call is. Here external uniqueness would be violated if the method assigns `this` to a preexisting external object. This object would have to have `owner` as its owner and be accessible from an object passed to the method. But, if the `owner` was merely acting as the virtual owner, then there would be no problem, because such an argument to the constructor would be unique and could be consumed by the constructor into the new object. Thus the fix is simple: *the parameters of a constructor cannot have* `owner` *in their type, except as a virtual owner.* This is a minor restriction – constructors may still make static aliases to its arguments and create and receive any objects (modulo type soundness and the restriction that `owner` may not appear in their type).

For simplicity, we omit constructors from our language description.

5 External Uniqueness at Work and Play

We now illustrate external uniqueness with a number of examples, highlighting aspects which would be impossible under existing uniqueness proposals.

```
class Server {
  unique Bus bus;
  this ServerSocket ss;

  void accept() {
    while (true) {
      unique Socket s = ss.accept();
      borrow bus as <bo> b {
        b.add(new unique Client(s--));
      }
    }
  }
}
```

```
class Client {
  owner Bus bus;
  unique Socket s;
  Client(unique Socket s) {
    this.s = s--;
  }
}
class Bus {
  this ClientList clients;
  void add(unique Client c) {
    this Client current = c--;
    clients.add(current);
    current.bus = this;       (*)
  }
}
```

Fig. 3. External uniqueness

External uniqueness In Figure 3 a server is listening for network connections. As soon as a client connects, a server side representation of the client is created and moved to a message bus. The reference to the message bus is unique. The point of interest is the line marked marked `(*)`: a reference to the server's bus is stored in the client, thus creating an internal alias. In ordinary models of uniqueness, this reference would either make the original unique

reference to the bus impossible, or consume the server's bus object the first time it was passed to a client. With external uniqueness, this reference is permitted.

The initialisation problem ([18]) To keep objects such as the following lexer flexible, we must be able to initialise its internal stream with an externally created object, without leaving any external aliasing to the stream object. Proposed solutions suffer from the weaknesses of shallow ownership [1, 8] (see Section 8).

```
class Lexer {
  this InputStream stream;     // internal
  Lexer(unique InputStream s) { stream = s--; }
}
void lexerClient() {
  unique InputStream stream = new FileInputStream(file);
  unique Lexer l = new Lexer(stream--);
}
```

Simulating borrowing Previous approaches to uniqueness use *lent* parameters to avoid the capture of unique references passed to methods. We can simulate this using borrowing and a call to an owner polymorphic method. Apart from making them applicable to objects with different owners, owner polymorphic methods cannot capture any argument whose owner is a parameter:

```
class Printer {
   static <o <* world> void print(o Printable p) { .. }
}
unique B b;    // B implements Printable
borrow b as <a> b' { Printer.print<a>(b'); }
```

Another benefit of this scheme is that the borrowed object may be stored temporarily on the heap during the borrowing. This is discussed shortly.

Transfer of ownership Transfer of ownership is an important design pattern in concurrent object-oriented programming [28]. In the following token ring, the token object is passed from one thread to the next by calling the `give` method. For example, Figure 4 illustrates the move of `Token` B from `TokenRing` element A to C. This idiom relies on the virtual owner of B being the owner of the surrounding object A, that is D, to ensure that B has no references to the objects inside A. That is, the token ring elements must be siblings.

```
class TokenRing {
   owner TokenRing next;  // sibling
   unique Token token;
   void give() { next.receive(token--); }
   void receive(unique Token tkn)
       { token = tkn--; }
}
```

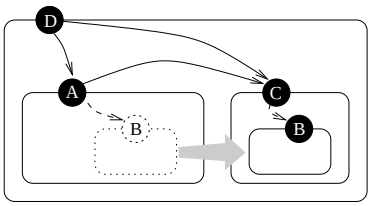


Fig. 4. A Token Ring. Transfer token B from A to its sibling C

External uniqueness allows the token object to be a fully-fledged aggregate which may be the resource shared between the elements in the token ring. Of course, there is no reason why this couldn't be a movement from one machine to another.

```
class Link<data> {
  data Object data;
  owner Link<data> next, prev;
}
class List<data> {
  unique Link<data> head;
  void append(owner List<data> other) {
    borrow head as <ho> bh {                    (1)
      ho Link<data> ohead = other.head--;       (2)
      if (bh == null) { bh = ohead; }           (*)
      else if (ohead != null) {
        ho Link<data> h = bh;
        while (h.next != null) { h = h.next; }
        h.next = ohead; h.next.prev = h;
      }
    }
  }
}
```

Fig. 5. Merging two doubly-linked lists

Merging representations External uniqueness enables the protected, self-referential internals of two data structures to be merged, without copying, into a new data structure, which offers the same degree of protection, *without* any residual aliasing from the original data structures. For example, Figure 5 shows the merging of one doubly-linked list into another. The phases of operation are: `(1)` borrow the local interior, `head`, using temporary owner `ho`; `(2)` move the other interior, `other.head`, so that it has owner `ho`; perform the merge (an append in this case); and then reinstate with the resulting value of `bh`, which may have been set in line `(*)`. Note that `other.head` is consumed in this operation.

David Holmes posed this example as a challenge when he saw the original ownership types proposal [15]. No existing combination of uniqueness and ownership types can handle it, but we finally do so in an elegant manner.

Moving parts In the previous example, there is a single unique reference into the internal links of the doubly-linked list from the `List` object. It is simple to modify this example to account for multiple references into a data structure. For concreteness, we consider a doubly-linked list with a tail pointer. The key is to introduce a proxy objects to hold both references into the list, and have a unique reference to this object.

```
class HeadAndTail<data> {          class List<data> {
   owner Link<data> head, tail;       unique HeadAndTail<data> handle;
}                                  }
```

Local objects and orthogonality Joline also has a construct which enables the creation of temporary heap objects which may refer to existing objects, including borrowed unique references. The statement `(`α`) { `s` }` creates a new owner which is implicitly nested inside some existing collection of owners, giving access to existing objects. Through the invariants of ownership types, the lifetime of objects of types containing α is limited to the scope of s. Objects created within the block can be used normally, even passed to owner polymorphic methods, but are discarded at the end of the block.

For example, the `print` method in an earlier example could have the body which follows. Here the `LayoutManager` object, which has owner `a`, can access the external `Printable` object, but cannot persist when the scope exits.

```
static <o <* world> void print(o Printable p) {
  (a) {                 // temporary owner
     a LayoutManager<o> lm = new a LayoutManager<o>(p);
     lm.addBorder();
     lm.print("/dev/printer");
  }
}
```

This feature is impossible without deep ownership, unless one uses an effects system. Scoped regions resemble an abstract, statically safe, type-level version of ScopedMemories from Real-time Java [5], which requires no dynamic checks.

6 A Touch of Formality

In this section, we present the static semantics for Joline, focusing on the most important rules. The remainder are relatively standard and straightforward and can be found in Appendix A. Type checking follows an elaboration phase in which each occurrence of `unique` in the user's program is annotated with a virtual owner. Program elaboration is defined in Appendix B: essentially it sets the virtual owner to be either `owner` or the innermost owner in scope. As a notational shortcut, we present rules with multiple conclusions, denoting one copy of the rule for each conclusion. The type environment E records the nesting relation on owner parameters, and the types of free term variables:

$$E ::= \emptyset \mid E, x : t \mid E, \alpha \prec^* \bigsqcup\{p_{i\in 1..n}\} \mid E, \alpha \succ^* p$$

The test $\mathsf{unique}(t)$ is true if and only if the type t is unique, that is, that its owner is $\texttt{unique}_p$. $\mathcal{F}_c$ and $\mathcal{M}_c$ are maps for each class c from the field names or method names in that class (or superclass) to their declared types, modulo the substitution of superclass parameters. The function $\mathsf{owners}()$ when applied to an environment gives all the owner parameters defined in that environment, including the global `world`, and when applied to a type gives all the owner arguments in the type. Substitutions of owners for owner parameters are denoted σ. The act of substituting is denoted $\sigma(\cdots)$. The notation σ^p represents the substitution $\sigma \cup \{\texttt{owner} \mapsto p\}$. A type may be written either as $p\ c\langle p_{i\in 1..n}\rangle$ or $p\ c\langle\sigma\rangle$, where $\sigma = \{\alpha_i \mapsto p_{i\in 1..n}\}$ for appropriate $\alpha_{i\in 1..n}$.

Types

$$\frac{\begin{array}{c}\text{(TYPE)}\\ \texttt{class}\ c\langle\alpha_i\,\mathsf{R}_i\,p_{i\in 1..n}\rangle \cdots \in P \\ \sigma = \{\texttt{owner} \mapsto q, \alpha_i \mapsto q_{i\in 1..n}\} \quad E \vdash \sigma(\alpha_i\,\mathsf{R}_i\,p_i)_{i\in 1..n}\end{array}}{\begin{array}{c}E \vdash q\ c\langle q_{i\in 1..n}\rangle \\ E \vdash \texttt{unique}_q\ c\langle q_{i\in 1..n}\rangle\end{array}}$$

A type is well-formed whenever the substituted owner arguments satisfy the ordering on parameters prescribed in the class. The type can be either unique or not. In the case of unique types, the virtual owner must satisfy the same constraint which it would satisfy as the actual owner.

Subtyping

(SUB-CLASS)

$$\frac{E \vdash p\ c\langle\sigma^p\rangle \quad \texttt{class}\ c\langle\ldots\rangle\ \texttt{extends}\ c'\langle p'_{i\in 1..n}\rangle \cdots \in P}{\begin{array}{c} E \vdash p\ c\langle\sigma\rangle \leq p\ c'\langle\sigma^p(p'_{i\in 1..n})\rangle \\ E \vdash \texttt{unique}_p\ c\langle\sigma\rangle \leq p\ c'\langle\sigma^p(p'_{i\in 1..n})\rangle \\ E \vdash \texttt{unique}_p\ c\langle\sigma\rangle \leq \texttt{unique}_p\ c'\langle\sigma^p(p'_{i\in 1..n})\rangle \end{array}}$$

(SUB-MOVE)

$$\frac{E \vdash p' \prec^* p \quad E \vdash p\ c\langle p_{i\in 1..n}\rangle}{E \vdash \texttt{unique}_p\ c\langle p_{i\in 1..n}\rangle \leq \texttt{unique}_{p'}\ c\langle p_{i\in 1..n}\rangle}$$

Subtyping is derived from subclassing, modulo the binding of superclass parameters. As this corresponds to the composition of two order-preserving functions, it is order-preserving, required to preserve deep ownership [13]. In particular, subtyping preserves the owner. Letting the owner vary, as in Cyclone [24], would be unsound in our system [14]. Finally, subtyping may preserve uniqueness or forget it, in which case the virtual owner is converted to a regular owner.

The second subtyping rule covers when it is permitted to move an object of one type to another. It is only possible to move the virtual owner inwards. This is no great limitation. One may explicitly specify a unique reference with virtual owner `world` which can be moved anywhere in the system.

Statements

(STAT-UPDATE)

$$\frac{E \vdash \mathit{lval} : t\ \ \textbf{ref} \quad E \vdash e : t}{E \vdash \mathit{lval} = e\texttt{;}\ ; E}$$

(STAT-SCOPED-REGION)

$$\frac{E, \alpha \prec^* \bigsqcup P \vdash s\ ; E' \quad P \subseteq \mathsf{owners}(E)}{E \vdash (\alpha)\ \texttt{\{}\ s\ \texttt{\}}\ ; E}$$

(STAT-BORROW)

$$\frac{E \vdash \mathit{lval} : \texttt{unique}_p\ c\langle p_{i\in 1..n}\rangle\ \ \textbf{ref} \quad E, \alpha \prec^* p, x : \alpha\ c\langle p_{i\in 1..n}\rangle \vdash s\ ; E'}{E \vdash \texttt{borrow}\ \mathit{lval}\ \texttt{as}\ \langle\alpha\rangle\ x\ \texttt{\{}\ s\ \texttt{\}}\ ; E}$$

The rule (STAT-UPDATE) simply enforces that updates can be performed to l-values only the types match, modulo subtyping. The rule (STAT-SCOPED-REGION) introduces a new owner variable for the duration of the given block. The bounds P, though unspecified in code, determine which objects may be accessed by objects created in this scope. From the rule (STAT-BORROW) any uniquely typed l-value may be borrowed. This is achieved by introducing a new owner variable which is restricted in scope to act as the owner of the temporary non-unique reference to the borrowed value. To ensure that this reference, or other references into the borrowed value do not escape this scope, we require that this owner is inside the virtual owner of the unique type. The remainder of the type *must* correspond exactly to the type of the l-value, so that it is reinstated with a correctly type value when the borrowing ends.

l-values

(LVAL-VAR)

$$\frac{x : t \in E \quad x \neq \texttt{this}}{E \vdash x : t\ \ \textbf{ref}}$$

(LVAL-FIELD)

$$\frac{\begin{array}{c} E \vdash e : p\ c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \\ \texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this} \end{array}}{E \vdash e.f : \sigma^p(t)\ \ \textbf{ref}}$$

These rules give the types of l-values, which are variables (other than `this`) and fields. Their type is given exactly as declared, modulo substitution of parameters. l-values may be

updated or, if unique, destructively read. The condition $\texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this}$, which was called the *static visibility* in the original ownership types system [15], ensures that types contain `this` in them can only be accessed internally to the object. It amounts to saying that fields (and methods) which return or require the object in internals are private. This is not essential; we could have used *dynamic aliasing* [14], but the type system would have been a little too complex to present our ideas. Subsumption *does not* apply to l-value types to ensure the validity of reinstatement at the end of borrowing.

Expressions

$$\frac{E \vdash \mathit{lval} : t \;\; \textbf{ref} \;\; \neg\mathsf{unique}(t)}{E \vdash \mathit{lval} : t}\;(\textsc{expr-lval}) \qquad \frac{E \vdash \mathit{lval} : t \;\; \textbf{ref} \;\; \mathsf{unique}(t)}{E \vdash \mathit{lval}\texttt{--} : t}\;(\textsc{expr-dread})$$

$$\frac{\begin{array}{c} E \vdash e : p\; c\langle\sigma\rangle \quad \mathcal{M}_c(m) = \forall(\alpha_i\, \mathsf{R}_i\, p_{i\in 1..n})t_{j\in 1..m} \rightarrow t_0 \\ \texttt{this} \in \mathsf{owners}(\forall(\alpha_i\, \mathsf{R}_i\, p_{i\in 1..n})t_{j\in 1..m} \rightarrow t_0) \Rightarrow e \equiv \texttt{this} \\ \sigma' = \{\alpha_i \mapsto q_{i\in 1..n}\} \quad E \vdash \sigma'(\alpha_i\, \mathsf{R}_i\, p_{i\in 1..n}) \quad E \vdash e_j : \sigma'(\sigma^p(t_j)) \quad \text{for all } j \in 1..m \end{array}}{E \vdash e.m\langle q_{i\in 1..n}\rangle(e_{j\in 1..m}) : \sigma'(\sigma^p(t_0))}\;(\textsc{expr-call})$$

Not all l-values can be directly treated as a value. The rules (EXPR-LVAL) and (EXPR-DREAD) correspond to extracting the value within the l-value. If the type is non-unique, then the (contents of) l-value can automatically be used as a value. If the type is unique, then a destructive read must be used to convert its contents into an expression.

The rule for method call is a behemoth. Firstly, it applies only to non-unique types (as does the rule for field access). The second line is the static visibility test which restricts expressions containing `this` in their type (as declared in the class) to being used only internally, that is on `this`. The owner arguments of the target type and the owner arguments supplied to the method form two substitutions to transform the method's argument and return types into types terms of the owners in scope. The value supplied to each argument of the method must have the type expected by the method. Finally, the method may return a uniquely typed value.

Note that there are no rules for accessing the fields and methods of unique references. To do so a borrowing must first be issued.

As usual, subsumption *does* apply to expression typing.

7 Discussion

We believe that our model of uniqueness is more appropriate and more stable than existing approaches, in that it it overcomes their limitations.

The abstraction problem Ownership types distinguish internal and external references. With them, we can permit arbitrary internal aliasing within an uniquely referenced aggregate, without loss of effective uniqueness. From a software engineering perspective, our proposal is better suited to software evolution than traditional uniqueness, since it does not require interfaces to change when the internal implementation does, as illustrated by the Client-Server example in Figure 3. In addition, we need not clutter interfaces with annotations on methods or constructors indicating calls that destructively consume their targets, as in other proposals, because methods cannot be called on a unique reference and their targets cannot be consumed. The price is the loss of uniqueness on `this`, precluding moves initiated within an object. The gains are much greater.

Orthogonality Borrowing was introduced in existing uniqueness proposals so that unique references can be operated upon without consuming their target. Existing uniqueness proposals impose the restriction that a borrowed reference cannot be stored in the field of an object, making borrowed references a kind of second class citizen. These proposals lack mechanisms in their type systems to treat borrowed references as usual non-unique references and maintain the invariant that after the borrowing has ceased, the reference again is unique.

Rather than introducing borrowed references as another kind of reference, our borrowing is a construct which mediates between unique and non-unique perspectives. Any non-unique reference, including those introduced in a borrowing and those internal to an object, can be passed to owner polymorphic methods, or stored on the heap in objects created within scoped regions. Ownership types guarantees that any aliases created are temporary or appropriately contained. The result is a cleaner language which employs orthogonal constructs in a flexible manner.

A unique is a unique is a unique A potential problem when dealing with unique references in the presence of borrowing is that reading a field or variable containing a uniquely typed reference while aliases exist violates the uniqueness expectation of those accessing the field. As we mentioned in Section 4, there are a number of ways for dealing with this. Recapping, we could weaken the definition of unique and not nullify a variable when borrowing, or we could nullify when borrowing thus introducing null pointers and potential race conditions, or we could employ alias burying and accept a sophisticated program analysis which does not work in a multi-threaded setting.

With either of the last two solutions, external uniqueness becomes effectively unique. That is, while not strictly unique, the dominating edge property means that there is only ever one active reference to an externally unique object, since internal references only become active during a borrowing when the original reference is unavailable. Thus there is either a single active unique reference to an object, or many references with limited scope, but never both. We can therefore conclude that external uniqueness is unique enough.

Reentrancy and internal threads An interesting consequence of our proposal regarding the reentrancy of objects is that, once a reference has been borrowed, an object can only be reentered from references inside the original object. It is not possible to call a method from an external object which then results in a call to the original object. We expect that this *localised reentrancy* will be handy when reasoning about objects. An important conclusion for concurrency is that an object referred to uniquely can only be entered by one thread.

However, if multiple threads are created within an object[2] during a borrowing, then their mere existence threatens the possibility of retaining a strong notion of uniqueness. As we see it, one of a number of restrictions is required, or we could adopt a more lax model.

Firstly, we could make the borrowing thread outlive all subthreads. This implies that if any internal threads are created, the borrowing thread must block at the end of the borrowing until these internal threads have finished.

The second approach is to permit the borrowing thread to proceed once it has finished the borrowing. To avoid weakening the system, we could also delay the possibility of re-borrowing the original object until the original collection of threads has ceased.

The third approach is to accept a weakened version of uniqueness. In this case a borrowing does not nullify its target and movement can occur underfoot. However, we can take the

[2] A thread is internal to an object if its owner is inside the object.

view that borrowing and internal threads are attached to the aggregate to which the unique reference refers. Thus the computation associated with the borrowing and internal threads can be considered to be inside the aggregate, and are thus moved with the moving aggregate.

More research is required to find the best approach.

Uniqueness, non-null and finality The stronger form of uniqueness interacts badly if we wish to have final fields containing unique values or non-null unique types [19]. The problems encountered are similar, though independent. When a final field contains a unique reference, a destructive borrowing violates the finality of the field and similarly, the non-nullity of a non-null type. We suspect that alias burying again will come to the rescue in the sequential setting, but beyond that we cannot yet have non-null or final unique references. Yet another approach is to have weaken external uniqueness for final fields.

Uniqueness and shallow ownership Dominating edges are possible only when dominators can be enforced. But this is impossible in systems lacking deep ownership, such as AliasJava [1] and some in the first author's thesis [13], and hence they cannot support external uniqueness. Adding ordinary uniqueness in such cases is easy, since the invariants underlying deep ownership need not be considered. (This is what originally caused us to baulk when adding uniqueness.)

Weakness and limitations Apart from the difficulty maintaining the uniqueness invariant in a sensible and cheap manner, external uniqueness also inherits the weaknesses of deep ownership. In brief, one must program with the single entry point to objects requirement. Consequently a choice must be made between the amount encapsulation desired and the flexibility. The difficulties arise when trying to implement patterns such as iterators or command objects which require multiple, sometimes temporary, access paths into the internals of an object, or observers which require multiple permanent references [20]. Shallow ownership overcomes these weaknesses [1], though some attempts have been made to overcome some them while maintaining deep ownership [14].

8 Related Work

Table 2 presents a comparison between different proposals in the literature with regard to the kinds of uniqueness, alias encapsulation, and borrowing they provide. For reasons of space, we restrict our discussion mainly to object-oriented programming languages. For a few pointers into less closely related literature, see *e.g.,* [9, 11], or for a recent discussion on aliasing in general, see [14].

We consider three kinds of uniqueness:

free values can be unique (*e.g.,* from object construction).

conventional uniqueness fields and variables may contain unique references to an object. Such a reference is the only one stored in the heap, and possibly the stack, modulo any borrowing.

external uniqueness fields and variables may contain externally unique reference *into* an aggregate object. Internal references to unique object are permitted.

Both forms of uniqueness subsume free. Freedom without uniqueness means that the freeness is lost as soon as the value is stored in a field or variable. Commonly used synonyms for uniqueness are linear and unshareable.

	Uniqueness	Encapsulation	Borrowing
This paper	External	Deep	Orthogonal
PRFJ [8] [a]	Conventional	Deep	Parameter
Flexible Alias Protection [33]	Free	Deep	–
Vault [16, 17] [b]	∼Conventional	Shallow	Orthogonal
AliasJava [1]	Conventional	Shallow	Parameter
Pivot Uniqueness [29] [c]	<Conventional	Shallow	Parameter
Capabilities for sharing [11] [d]	∼Conventional	∼Shallow	∼Parameter
Islands [26] [e]	Conventional	Full	∼Parameter
Balloons [2]	Conventional	Full	Parameter
OOFX/Alias Burying [9, 23]	Conventional	None	Parameter
Eiffel* [31]	Conventional	None	Parameter
Virginity [30]	Free	None	Parameter

Table 2. Comparison. Legend: – not applicable; *a*-*e* see text

We consider three kinds of ownership/encapsulation:

shallow direct access to certain objects is limited.

deep the only access (transitively) to the internal state of an object is through a single-entry point. References to external state is possible.

full same as deep ownership, except that no references to objects outside the encapsulating object from within the encapsulating boundary are permitted.

While shallow ownership prevents direct access to the encapsulated objects, proxy objects may be created (internally or externally) which access the encapsulated objects and may escape the encapsulation boundary. level of encapsulation can be obtained using existential types. Deep ownership goes further by lifting the nesting of objects into the type system and ensuring that no references to deeply nested objects pass through their enclosing boundary. This is also called flexible alias encapsulation [33]. Full alias encapsulation [33] is like deep ownership except that references to external objects are not permitted from within the encapsulation boundaries. In graph theoretic terms, deep ownership imposes that owners are dominators which break path connectivity when removed, whereas full alias encapsulation imposes that "owners" are cut points which break graph connectivity when removed.

Other forms of encapsulation, such as the package level restrictions in Confined Types [25], are too coarse-grained to enable external uniqueness.

Kim, Bertino and Garza [**?**] define semantics for references capable of expressing a shallow form of ownership and traditional uniqueness for *composite references*. This system is however not statically checked nor does it provide deep ownership or external uniqueness, though the machinery seems to be in place to implement the latter via dynamic checks. A more detailed comparision with this and other models from the OODB community will be instructive.

We consider two kinds of borrowing of unique references:

borrowed parameters method parameters, `this`, and/or local variables may borrow a unique reference. Borrowed references may not be assigned to fields.

orthogonal borrowing references are either unique or non-unique. Scope restrictions apply to a borrowed unique reference to ensure that the uniqueness invariant can be regained.

In Eiffel* [31], AliasJava [1], Balloons [2], Pivot Uniqueness [29] and Capabilities for sharing [11], borrowing weakens uniqueness since the unique reference is still visible and usable despite

the existence of the temporary borrowed aliases. This is avoided in PRFJ [8], Vault [16] and our proposal by using nullification or scope restrictions, and in Alias Burying [9] by invalidating all borrowings if a unique is read while borrowed. Checking the constraints underlying alias burying modularly leads to an interdependence between uniqueness and read effects [10]. Guava [3] also uses lent parameters to avoid capturing of objects.

Some noteworthy remarks regarding Table 2 follow. (a) PRFJ [8] permits object graphs which violates deep ownership, but it uses an effects system to prevent access through the offending references. The result is *effectively deep ownership*. In addition, to increase flexibility, PRFJ allows `unique` to be used even as a non-owner parameter. For example, to allow a list class where the data in the links is unshared, see Figure 5, `unique` is used in the `data` owner position. This requires that the list class be written with this in mind. Unfortunately, no formalisation exists, and so we have remained conservative and permit `unique` to appear only as the owner. (b) DeLine and Fähndrich [17] give a practical linear type system for a non object-oriented, imperative language. Our borrowing resembles their adopt operation, where a linear reference temporarily becomes non-linear. They also have a focus operation which enables non-linear references to linear entities to be treated linearly, avoiding destructive read. The inflexible nature of classes makes this difficult to achieve in our setting. (c) Pivot uniqueness [29] only enables unique fields to be assigned with newly created objects or null. (d) Capabilities for sharing [11] offers primitive and dynamic constructs that combine to give various kinds uniqueness, read-only references *etc.*, though no one specific policy is enforced. Furthermore, no static type system exists. Finally, (e) Islands [26] only allows borrowing through read-only references.

Uniqueness and Linearity Girard's linear logic [22] created the opportunity for stronger control of resources in programming languages. However, a number of researchers have realised that programming with uniqueness or linearity in its strictest form is painful [37, 4]. Wadler's `let!` construct, quasi-linear types [27], and Vault's adoption and focus [17], for example, introduce means for alleviating this pain. Our notion of external aliasing and to a lesser extent our borrowing construct were designed for a similar goal, but in the object-oriented setting.

Region-based memory management Our scoped region construct is similar to lexically scoped `letregion` construct used in region-based memory management [35, 36]. There are a number of differences. Firstly, our construct is under programmer control, as in Cyclone [24], whereas the regions calculus is the basis for a compiler's intermediate language. Secondly, the principal aim of region-based memory management differs from ours, which is to limit the aliasing between objects. The final difference is the technical machinery used to achieve safety: our approach is structural, maintaining a specific nesting relationship between objects to ensure that no references into a deleted region remain (see also [13]), whereas the regions calculus uses effects to determine that references into a deleted region are never dereferenced. Both Cyclone [24] and Gay and Aiken's RC [21] manage a nesting relationship which captures when one object *outlives* another. While some attempts to explicitly add region-based memory management to Java exist [38, 12], they require interfaces to be extended with effects annotations to ensure modular checking, whereas our structural approach does not. Although the structural approach lacks the delicacy of the regions calculus, we believe that it is closer to the spirit of object-oriented programming. Indeed, real-time Java [5] includes ScopedMemory objects which behave similarly to our scoped regions, without guarantees of static safety. All regions systems lack the deep ownership and unique references.

Deep ownership enables an object to be seen as having a region (referred to using `this`) containing the objects it owns, revealing an interesting duality: *in the region calculus, the lifetime of objects depends upon the lifetime of regions; in deep ownership types, the lifetime of regions depends upon the lifetime of objects.* We plan to investigate this further.

9 Conclusions and Future Work

In this paper we introduced a new take on uniqueness called external uniqueness. This was a natural extension of ownership types; while not free, it is, in a sense, already paid for once you have ownership types.[3] On the surface our proposal seems to violate the very essence of uniqueness, permitting arbitrary internal references to an object which is referred to by an externally unique pointer, but, as we have demonstrated, the external references is the only active reference to the object, and is thus effectively unique. Furthermore, our definition solves an abstraction problem from which existing approaches to uniqueness suffer.

This work forms part of a general program to localise references, behaviour, control, *etc.* to simplify reasoning about programs. We wish to further our work in this direction. In addition, we are adding external uniqueness to our Joe compiler [14] so that we can then explore more programming patterns and apply our ideas to concurrency, mobility, memory management and so forth.

Note An earlier version of this work will be presented at FOOL 10 in January 2003. A complete formal account of external uniqueness will appear as Tobias' licentiate thesis in March or April 2003.

References

1. Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA*, November 2002.
2. Paulo Sérgio Almeida. Balloon Types: Controlling sharing of state in data types. In *ECOOP Proceedings*, June 1997.
3. David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA Proceedings*, pages 382–400, 2000.
4. Henry G. Baker. 'Use-once' variables and linear objects – storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1), January 1995.
5. Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
6. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, November 2002.
7. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types and safe lazy upgrades in object-oriented databases. Technical Report MIT-LCS-TR-858, Laboratory for Computer Science, MIT, July 2002.
8. Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA Proceedings*, 2001.
9. John Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 31(6):533–553, May 2001.
10. John Boyland. The interdependence of effects and uniqueness. In *3rd Workshop on Formal Techniques for Java Programs*, June 2001.
11. John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *ECOOP Proceedings*, June 2001.
12. M. V. Christiansen and P. Velschrow. Region-based memory management in Java. Master's thesis, Department of Computer Science (DIKU), University of Copenhagen, May 1998.

[3] Quoting Simon Peyton Jones

13. David Clarke. *Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.
14. David Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, November 2002.
15. David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA Proceedings*, 1998.
16. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.
17. Robert DeLine and Manuel Fähndrich. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2002.
18. David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Technical Report SRC-RR-98-156, Compaq Systems Research Center, July 1998.
19. Manuel Fähndrich and K. Rustan M. Leino. Non-null types in an object-oriented language. In Erik Poll, editor, *Formal Techniques for Java-like Programs*, Málaga, Spain, June 2002. Appears in Technical report NIII-R0204, Computing Science Department, University of Nijmegen, 2002.
20. Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
21. David Gay and Alex Aiken. Language support for regions. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, June 2001.
22. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
23. Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99*, 1999.
24. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2002.
25. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *OOPSLA*, 2001.
26. John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA Proceedings*, November 1991.
27. Naoki Kobayashi. Quasi-linear types. In *26th ACM Symposium on Principles of Programming Languages*, January 1999.
28. Doug Lea. *Concurrent-Programming in Java: Design Principles and Patterns*. Java Series. Addision-Wesley, 1998.
29. K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *PLDI*, June 2002.
30. K. Rustan M. Leino and Raymie Stata. Virginity: A contribution to the specification of object-oriented software. *Information Processing Letters*, 70(2):99–105, April 1999.
31. Naftaly Minsky. Towards alias-free pointers. In *ECOOP Proceedings*, July 1996.
32. P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.
33. James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98—Object-Oriented Programming*, volume 1445 of *Lecture Notes In Computer Science*, pages 158–185, Berlin, Heidelberg, New York, July 1988. Springer-Verlag.
34. John Potter, James Noble, and David Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, Adelaide, Australia, November 1998. IEEE Press.
35. J.-P Talpin and P. Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.
36. Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.
37. Phil Wadler. Linear type can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, April 1990. North-Holland.
38. Bennett Norton Yates. A type-and-effect system for encapsulating memory in Java. Master's thesis, Department of Computer and Information Science and the Graduate School of the University of Oregon, August 1999.

A Type Rules

The complete collection of type rules are given here.

$E \vdash \diamond$ — Good environment

$$\frac{}{\emptyset \vdash \diamond}\ (\text{ENV-}\emptyset) \qquad \frac{E \vdash t \quad x \notin \mathrm{dom}(E)}{E, x : t \vdash \diamond}\ (\text{ENV-}x) \qquad \frac{E \vdash p \quad \alpha \notin \mathrm{dom}(E)}{E, \alpha \succ^* p \vdash \diamond}\ (\text{ENV-}\alpha \succ^*) \qquad \frac{E \vdash p_{i\in 1..n} \quad \alpha \notin \mathrm{dom}(E)}{E, \alpha \prec^* \bigsqcup\{p_{i\in 1..n}\} \vdash \diamond}\ (\text{ENV-}\alpha \prec^*)$$

$E \vdash p$ — Good owner

$$\frac{\alpha\ \mathsf{R}\ _ \in E}{E \vdash \alpha}\ (\text{OWNER-VAR}) \qquad \frac{\texttt{this} : t \in E}{E \vdash \texttt{this}}\ (\text{OWNER-THIS}) \qquad \frac{E \vdash \diamond}{E \vdash \texttt{world}}\ (\text{OWNER-WORLD})$$

$E \vdash p \prec^* q$ — Owner p is inside q

$$\frac{\alpha \prec^* \bigsqcup P \in E \quad p \in P}{E \vdash \alpha \prec^* p}\ (\text{IN-ENV1}) \qquad \frac{\alpha \succ^* p \in E}{E \vdash p \prec^* \alpha}\ (\text{IN-ENV2}) \qquad \frac{E \vdash p}{E \vdash p \prec^* \texttt{world}}\ (\text{IN-WORLD})$$

$$\frac{\texttt{this} : t \in E}{E \vdash \texttt{this} \prec^* \texttt{owner}}\ (\text{IN-THIS}) \qquad \frac{E \vdash p}{E \vdash p \prec^* p}\ (\text{IN-REFL}) \qquad \frac{E \vdash p \prec^* q \quad E \vdash q \prec^* q'}{E \vdash p \prec^* q'}\ (\text{IN-TRANS})$$

$\vdash P, \vdash class$ — Good Program and Class

$$\frac{\vdash class \text{ for all } class \in P \quad \vdash s\,; E \quad E \vdash e : t}{\vdash class_{i\in 1..n}\ s\ e : t}\ (\text{PROGRAM}) \qquad \frac{}{\vdash \texttt{class Object \{ \}}}\ (\text{CLASS-OBJECT})$$

$$\frac{\begin{array}{c} E_0 = \texttt{owner} \prec^* \texttt{world}, \alpha_i\ \mathsf{R}_i\ p_{i\in 1..m} \quad E_0 \vdash \texttt{owner} \prec^* \alpha_{i\in 1..m} \\ E_0 \vdash \texttt{owner}\ c'\langle\sigma\rangle \quad \texttt{owner} \notin \mathrm{rng}(\sigma) \quad E = E_0, \texttt{this} : \texttt{owner}\ c\langle\alpha_{i\in 1..m}\rangle \\ \{f_{i\in 1..n}\} \cap \mathrm{dom}(\mathcal{F}_{c'}) = \emptyset \quad E \vdash e_i : t_{i\in 1..r} \quad E \vdash meth_{j\in 1..s} \\ \mathcal{M}_c(m) \equiv \sigma(\mathcal{M}_{c'}(m)) \quad \forall m \in \mathsf{names}(meth_{j\in 1..s}) \cap \mathrm{dom}(\mathcal{M}_{c'}) \end{array}}{\vdash \texttt{class}\ c\langle\alpha_i\ \mathsf{R}_i\ p_{i\in 1..m}\rangle\ \texttt{extends}\ c'\langle\sigma\rangle\ \{t_i\ f_i = e_{i\in 1..r}\ meth_{j\in 1..s}\}}\ (\text{CLASS})$$

$E \vdash meth$ — Good Method

$$\frac{E'' = E, \alpha_i\ \mathsf{R}_i\ p_{i\in 1..n}, x_j : t_{j\in 1..m} \quad E'' \vdash s;\ E' \quad E' \vdash e : t_0}{E \vdash \langle\alpha_i\ \mathsf{R}_i\ p_{i\in 1..n}\rangle\ t_0\ m(t_i\ x_{i\in 1..m})\{\ s\ \texttt{return}\ e\ \}}\ (\text{METHOD})$$

$E \vdash t$ — Good Type

$$\frac{\begin{array}{c} \texttt{class}\ c\langle\alpha_i\ \mathsf{R}_i\ p_{i\in 1..n}\rangle \cdots \in P \\ \sigma = \{\texttt{owner} \mapsto q, \alpha_i \mapsto q_{i\in 1..n}\} \quad E \vdash \sigma(\alpha_i\ \mathsf{R}_i\ p_i)_{i\in 1..n} \end{array}}{\begin{array}{c} E \vdash q\ c\langle q_{i\in 1..n}\rangle \\ E \vdash \texttt{unique}_q\ c\langle q_{i\in 1..n}\rangle \end{array}}\ (\text{TYPE})$$

$E \vdash t \leq t'$ — t is a subtype of t'

$$\frac{E \vdash t}{E \vdash t \leq t}\ (\text{SUB-REFL}) \qquad \frac{E \vdash p\ c\langle\sigma^p\rangle \quad \texttt{class}\ c\langle\ldots\rangle\ \texttt{extends}\ c'\langle p'_{i\in 1..n}\rangle \cdots \in P}{\begin{array}{c} E \vdash p\ c\langle\sigma\rangle \leq p\ c'\langle\sigma^p(p'_{i\in 1..n})\rangle \\ E \vdash \texttt{unique}_p\ c\langle\sigma\rangle \leq p\ c'\langle\sigma^p(p'_{i\in 1..n})\rangle \\ E \vdash \texttt{unique}_p\ c\langle\sigma\rangle \leq \texttt{unique}_p\ c'\langle\sigma^p(p'_{i\in 1..n})\rangle \end{array}}\ (\text{SUB-CLASS})$$

$$\frac{E \vdash t \leq t' \quad E \vdash t' \leq t''}{E \vdash t \leq t''}\ (\textsc{sub-trans}) \qquad \frac{E \vdash p' \prec^* p \quad E \vdash p\ c\langle p_{i\in 1..n}\rangle}{E \vdash \texttt{unique}_p\ c\langle p_{i\in 1..n}\rangle \leq \texttt{unique}_{p'}\ c\langle p_{i\in 1..n}\rangle}\ (\textsc{sub-move})$$

$E \vdash s\ ; E'$ — Good statement. Resulting environment E'

$$\frac{E \vdash \diamond}{E \vdash \texttt{skip;}\ ; E}\ (\textsc{stat-skip}) \quad \frac{x \notin \mathrm{dom}(E) \quad E \vdash e : t}{E \vdash t\ x\ = e;\ ; E, x : t}\ (\textsc{stat-local}) \quad \frac{E \vdash e : t}{E \vdash e;\ ; E}\ (\textsc{stat-expr}) \quad \frac{E \vdash \mathit{lval} : t\ \mathbf{ref} \quad E \vdash e : t}{E \vdash \mathit{lval} = e;\ ; E}\ (\textsc{stat-update})$$

$$\frac{E \vdash s_1\ ; E'' \quad E'' \vdash s_2\ ; E'}{E \vdash s_1\ s_2\ ; E'}\ (\textsc{stat-sequence}) \quad \frac{E, \alpha \prec^* \bigsqcup P \vdash s\ ; E' \quad P \subseteq \mathsf{owners}(E)}{E \vdash (\alpha)\ \texttt{\{}\ s\ \texttt{\}}\ ; E}\ (\textsc{stat-scoped-region}) \quad \frac{E \vdash s\ ; E'}{E \vdash \{s\}\ ; E}\ (\textsc{stat-block})$$

$$\frac{E \vdash e : \texttt{bool} \quad E \vdash s_1\ ; E' \quad E \vdash s_2\ ; E''}{E \vdash \texttt{if}\ (e)\ \texttt{\{}\ s_1\ \texttt{\}}\ \texttt{else}\ \texttt{\{}\ s_2\ \texttt{\}}\ ; E}\ (\textsc{stat-if-then-else}) \quad \frac{\begin{array}{c} E \vdash \mathit{lval} : \texttt{unique}_p\ c\langle p_{i\in 1..n}\rangle\ \mathbf{ref} \\ E, \alpha \prec^* p, x : \alpha\ c\langle p_{i\in 1..n}\rangle \vdash s\ ; E' \end{array}}{E \vdash \texttt{borrow}\ \mathit{lval}\ \texttt{as}\ \langle\alpha\rangle\ x\ \texttt{\{}\ s\ \texttt{\}}\ ; E}\ (\textsc{stat-borrow})$$

$E \vdash \mathit{lval} : t\ \mathbf{ref}$ — l-value of type t

$$\frac{x : t \in E \quad x \neq \texttt{this}}{E \vdash x : t\ \mathbf{ref}}\ (\textsc{lval-var}) \qquad \frac{\begin{array}{c} E \vdash e : p\ c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \\ \texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this} \end{array}}{E \vdash e.f : \sigma^p(t)\ \mathbf{ref}}\ (\textsc{lval-field})$$

$E \vdash e : t$ — Expression e has type t

$$\frac{E \vdash \mathit{lval} : t\ \mathbf{ref} \quad \neg\mathsf{unique}(t)}{E \vdash \mathit{lval} : t}\ (\textsc{expr-lval}) \qquad \frac{E \vdash \mathit{lval} : t\ \mathbf{ref} \quad \mathsf{unique}(t)}{E \vdash \mathit{lval}\texttt{--} : t}\ (\textsc{expr-dread})$$

$$\frac{\texttt{this} : t \in E}{E \vdash \texttt{this} : t}\ (\textsc{expr-this}) \quad \frac{E \vdash t}{E \vdash \texttt{new}\ t : t}\ (\textsc{expr-new}) \quad \frac{E \vdash t}{E \vdash \texttt{null} : t}\ (\textsc{expr-null}) \quad \frac{E \vdash e : t \quad E \vdash t \leq t'}{E \vdash e : t'}\ (\textsc{expr-subsumption})$$

$$\frac{\begin{array}{c} E \vdash e : p\ c\langle\sigma\rangle \quad \mathcal{M}_c(m) = \forall(\alpha_i\ \mathsf{R}_i\ p_{i\in 1..n})t_{j\in 1..m} \to t_0 \\ \texttt{this} \in \mathsf{owners}(\forall(\alpha_i\ \mathsf{R}_i\ p_{i\in 1..n})t_{j\in 1..m} \to t_0) \Rightarrow e \equiv \texttt{this} \\ \sigma' = \{\alpha_i \mapsto q_{i\in 1..n}\} \quad E \vdash \sigma'(\alpha_i\ \mathsf{R}_i\ p_{i\in 1..n}) \quad E \vdash e_j : \sigma'(\sigma^p(t_j)) \quad \text{for all } j \in 1..m \end{array}}{E \vdash e.m\langle q_{i\in 1..n}\rangle(e_{j\in 1..m}) : \sigma'(\sigma^p(t_0))}\ (\textsc{expr-call})$$

B Program Elaboration

Elaboration

Classes and definitions

$\mathsf{E}(class_{i\in 1..n}\ s\ e) = \mathsf{E}(class_i)_{i\in 1..n}\ \mathsf{E}_{\texttt{world}}(s)\ \mathsf{E}_{\texttt{world}}(e)$

$\mathsf{E}(\texttt{class}\ c\langle \alpha_i\ \mathsf{R}_i\ p_{i\in 1..m}\rangle\ \texttt{extends}\ c'\langle p'_{i'\in 1..n}\rangle\ \{\ fd_{j\in 1..m}\ meth_{k\in 1..m}\ \}) =$
$\texttt{class}\ c\langle \alpha_i\ \mathsf{R}_i\ p_{i\in 1..m}\rangle\ \texttt{extends}\ c'\langle p'_{i'\in 1..n}\rangle\ \{\ \mathsf{E}_{\texttt{owner}}(fd_j)_{j\in 1..m}\ \mathsf{E}_{\texttt{owner}}(meth_k)_{k\in 1..m}\ \}$

$\mathsf{E}_p(t\ f = e;) = \mathsf{E}_p(t)\ f = \mathsf{E}_p(e);$

$\mathsf{E}_p(\langle \alpha_i\ \mathsf{R}_i\ p_{i\in 1..n}\rangle\ t\ m(t_j\ x_{j\in 1..m})\ \{\ s\ \texttt{return}\ e\ \}) =$
$\langle \alpha_i\ \mathsf{R}_i\ p_{i\in 1..n}\rangle\ \mathsf{E}_p(t)\ m(\mathsf{E}_p(t_j)\ x_{j\in 1..m})\ \{\ \mathsf{E}_p(s)\ \texttt{return}\ \mathsf{E}_p(e)\ \}$

Lvalues and Expressions

$\mathsf{E}_p(x) = x$ $\quad \mathsf{E}_p(e.f) = \mathsf{E}_p(e).f$ $\quad \mathsf{E}_p(\texttt{this}) = \texttt{this}$

$\mathsf{E}_p(lval\texttt{--}) = \mathsf{E}_p(lval)\texttt{--}$ $\quad \mathsf{E}_p(\texttt{new}\ t) = \texttt{new}\ \ \mathsf{E}_p(t)$ $\quad \mathsf{E}_p(\texttt{null}) = \texttt{null}$

$\mathsf{E}_p(e.m\langle p_{j\in 1..m}\rangle(e_{i\in 1..n})) = \mathsf{E}_p(e).m\langle \mathsf{E}_p(p_j)_{j\in 1..m}\rangle(\mathsf{E}_p(e_i)_{i\in 1..n})$

Statements

$\mathsf{E}_p(\texttt{skip;}) = \texttt{skip;}$ $\quad \mathsf{E}_p(t\ x = e;) = \mathsf{E}_p(t)\ x = \mathsf{E}_p(e);$ $\quad \mathsf{E}_p(e;) = \mathsf{E}_p(e);$

$\mathsf{E}_p(s\ s') = \mathsf{E}_p(s)\ \mathsf{E}_p(s')$ $\quad \mathsf{E}_p(lval = e;) = \mathsf{E}_p(lval) = \mathsf{E}_p(e);$

$\mathsf{E}_p(\{\ s\ \}) = \{\ \mathsf{E}_p(s)\ \}$ $\quad \mathsf{E}_p((\alpha)\ \{\ s\ \}) = (\alpha)\ \{\ \mathsf{E}_\alpha(s)\ \}$

$\mathsf{E}_p(\texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}) = \texttt{if}\ (\mathsf{E}_p(e))\ \{\ \mathsf{E}_p(s_1)\ \}\ \texttt{else}\ \{\ \mathsf{E}_p(s_2)\ \}$

$\mathsf{E}_p(\texttt{borrow}\ lval\ \texttt{as}\ \langle\alpha\rangle\ x\ \{\ s\ \}) = \texttt{borrow}\ \mathsf{E}_p(lval)\ \texttt{as}\ \langle\alpha\rangle\ x\ \{\ \mathsf{E}_\alpha(s)\ \}$

Contexts and Types

$\mathsf{E}_p(\alpha) = \alpha$ $\quad \mathsf{E}_p(\texttt{owner}) = \texttt{owner}$ $\quad \mathsf{E}_p(\texttt{world}) = \texttt{world}$

$\mathsf{E}_p(\texttt{unique}) = \texttt{unique}_p$ $\quad \mathsf{E}_p(p_0\ c\langle p_{i\in 1..n}\rangle) = \mathsf{E}_p(p_0)\ c\langle \mathsf{E}_p(p_i)_{i\in 1..n}\rangle$