External Uniqueness

Dave Clarke
Inst. of Computer and Information Sciences
Utrecht University
Utrecht, The Netherlands
dave@cs.uu.nl

Tobias Wrigstad
Dept. of Computer and Systems Sciences
Stockholm University/KTH
Stockholm, Sweden
tobias@dsv.su.se

ABSTRACT

Existing approaches to adding unique references to object-oriented programming languages suffer from two problems, which we identify in this paper: the *abstraction problem* and the *orthogonality problem*. To counter these problems, we propose an alternative: *external uniqueness*. The idea is that an externally unique reference is the only reference *into* an aggregate object from outside the aggregate. An externally unique reference is the only active reference to an object, and is thus effectively unique. We present a formal account of external uniqueness which relies essentially on deep ownership types, and, by demonstrating that it does not suffer from the two problems, argue that it is better suited to object-oriented programming than conventional uniqueness. We also discuss a third problem, the *definitional problem*, which we cannot yet address satisfactorily.

1. INTRODUCTION

Unique or alias-free pointers have been proposed to reduce aliasing in object-oriented programming languages in order to, for example, simplify reasoning about programs [21, 27] and to safely enable the transfer of ownership, an essential idiom in concurrent programming [24]. The idea is simple: any variable or field annotated with the keyword `unique` contains the only reference to the object it holds, otherwise it contains `null`. To preserve uniqueness fields and variables are destructively read, meaning that they are nullified once their contents are read. Usually a *borrowing* mechanism is available to increase the usability of uniqueness by allowing the unique reference to be temporarily passed to methods without being consumed [21, 27, 9, 10, 7, 1]. In all cases, the same notion of uniqueness remains.

In this paper we argue that existing uniqueness proposals are broken, because they suffer from at least one of the two problems we identify here.

The *abstraction problem* occurs when changes to the implementation of an object with regard to uniqueness force changes to its interface, thus violating abstraction.

The *orthogonality problem* is that, although we borrow unique references to enable usual operations to be performed upon them, the collection of operations is limited.

To overcome these problems 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, though not limiting the number of references to the aggregate from its inside. Interestingly, however, it turns out that the externally unique reference is the only active reference to an object, so it is effectively unique. The key is that external uniqueness is an extension of the deep ownership available in certain ownership types systems [14, 13].

Not only do we resolve the two problems, but we discover an interesting consequence of our proposal: we enable transfer of entire aggregates between objects, even in the presence of aliasing within such aggregates. This permits us to move (or even combine) the encapsulated representation of data structures such as doubly-linked lists, without leaving any unwanted aliases which would break encapsulation. Above all, we do all of this for a class-based object-oriented programming language with subtyping.

Unfortunately, our proposal does not address all issues satisfactorily. There remains a *definitional problem* which is much more difficult to deal with. We discuss this problem and current approaches to it, but leave its resolution for future work.

Outline. This paper is organised as follows: after describing the three problems in Section 2, we overview ownership types in Section 3. In Section 4 we introduce external uniqueness, before adding it to a programming language called Joline in Section 5. After showing Joline at work in Section 6, we give an overview of its static semantics in Section 7. We wrap up the paper with a discussion of the three problems in Section 8, a comparison of related work in Section 9, before concluding in Section 10.

2. CHALLENGING UNIQUENESS

The extant model of uniqueness is broken. Proposals in the literature suffer from some of the following problems:

Abstraction software evolution which changes the uniqueness aspects of an object's implementation can force changes in the object's interface;

Orthogonality a borrowed reference cannot be treated as a usual non-unique reference; and

Definitional it is possible to encounter a field or variable containing a uniquely typed reference to an object while other active references to the same object exist.

In the related work section we detail which proposals suffer from which problem.

2.1 The Abstraction Problem

There are essentially two ways for dealing with unique references in object-oriented programming, each depending upon how a class treats itself (`this`) internally. The treatment of `this` is reflected in the interface to a class. For example, Minsky [27] divides classes into two worlds, those which may assign `this` internally and those which do not. Only instances of the latter may be uniquely referenced. Hogg [21], Boyland [9], and Aldrich *et.al.,* [1] annotate a class' interface to indicate which methods consume `this`. Calling such a method requires a destructive read of the target object.

Assume now that we have a field referring to some object of class `BlackBox`, such as `unique BlackBox bb`. When we change the implementation of the `BlackBox` class so that a reference to `this` is retained internally, we are *forced* to change the interface to `BlackBox`. In the first instance, the declaration above would no longer be valid. This would mean that it would have to be changed, having its uniqueness annotation removed, and all destructive reads of `BlackBox` objects would have to be changed to ordinary reads, perhaps with destructive read implemented manually. In the second instance, the method which creates the internal link (which may be the constructor) would consume the `BlackBox` object. Not only does this require the method to be annotated as such, but it also changes the semantics for objects which call that method since their target suddenly is consumed, whereas previously it was not.

Not only does a change in the implementation of `BlackBox` force changes to its interface, violating abstraction, but the change propagates through the program, either dynamically or statically, introducing the opportunity for errors, since the behaviour of the program is also changed.

We would like to consider objects as black boxes, but are forced to change the interface as we change the implementation under existing uniqueness proposals. Thus, we conclude that the extant uniqueness proposals break *abstraction*.

The abstraction problem, we believe, is because the distinction cannot be made between internal, innocuous references which do not escape abstraction boundaries and external references.

2.2 The Orthogonality Problem

To perform certain kinds of operations on a unique reference, it must first be borrowed. A borrowing is intended to be a temporary operation which does not consume the unique reference. Often many method arguments are not consumed by their target object, and, after they are annotated to indicate this, can be used with unique references.

To avoid consuming the original unique reference, restrictions must be placed upon the borrowed references. The invariant these restrictions endeavour to maintain is that no borrowed reference is captured. Thus, *after* the borrowing has ceased, the original reference is again unique.

To have a clean programming language, we would like to use existing constructs and treat borrowed references as ordinary references, so long as the invariant is maintained. But existing approaches to borrowing impose the restriction that a borrowed reference cannot be stored in the field of an object. Thus a borrowed reference is yet another kind of reference with its own rules and regulations, rather than an entity managed using existing language constructs.

The orthogonality problem, we believe, is due to a weakness in the type system of the underlying programming language, in particular, with regard to alias encapsulation.

2.3 The Definitional Problem

The definitional problem is really a dilemma. Some existing borrowing constructs permit, through reentrancy, via multi-threading, or even as soon as a method is called, both an active unique reference to an object and active borrowed references to the same object at the same time. When a declaration says that a field or variable contains a unique reference to an object, it ought to contain the only active reference, otherwise the very essence of uniqueness is violated, and any reasoning power would be reduced.

There are essentially three approaches for dealing with this problem. The first is to accept the the weakened reasoning power which comes from a weakened definition of uniqueness. The second approach is to nullify the original unique reference during borrowing, and accept the the risk of more null pointer exceptions and race conditions between different threads trying to obtain a single unique reference. The final approach is to ensure that if the unique reference is used, that all aliases (or borrowings of it) are dead [9]. The problem here is that a complicated program analysis using effects annotations (for modularity) is required. Apart from the overhead of such annotations, this approach probably reinforces the abstraction problem and moreover may not easily extend to a multi-threaded setting.

2.4 Steps toward a solution

We solve the abstraction problem using external uniqueness, a new take on uniqueness which is based on Clarke, Noble, and Potter's *ownership types* [14]. External uniqueness permits arbitrary internal aliasing, and it is precisely because ownership types make the distinction between the inside and outside of an object that we can even consider proposing it. Indeed, external uniqueness refines the object graph property underlying ownership types from dominating nodes to dominating edges. The orthogonality problem can also be dealt with using ownership types (and some obvious extensions) in the underlying language. Finally, to keep our proposal relatively simple, we address the definitional problem essentially by nullifying the unique reference when borrowing. In actual fact, we separate the operations which can be performed on a unique reference (movement and borrowing) from those which can be applied on a regular reference (method call, field access, *etc.*) and introduce a borrowing construct for temporarily treating a unique reference to a non-unique reference. While not perfect, the alternative would require at least an effects system, which would obscure the essence of our work, and so we leave that for future work.

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

Traditional object-oriented programming languages cannot make the distinction between references internal to a data structure, *e.g.,* references between links of a linked list, and references that are external and go *into* a data structure from *outside* of it, *e.g.,* the reference from the handle object for the linked list. Making such a distinction would allow one to permit arbitrary internal references within an abstraction, while allowing a unique external reference to it. Having this distinction would be a good thing, since requiring knowledge of internal references exposes an object's implementation details and thus violates abstraction.

This distinction can be made in a programming language which has ownership types, such as was originally proposed by Clarke, Potter and Noble [14]. Ownership types provide strong protection against external aliasing of an object's representation, thus enabling a strong notion of aggregate objects where the owner of the aggregate acts as a single-entry point to the aggregate. The resulting anti-aliasing guarantees are compatible with the containment implicit in object-oriented programming.

Ownership types package together into a class-based type system a number of conditions which restrict the structure of object graphs in a certain way. Let us look at the required ingredients:

Objects have owners and can be the owners of other objects. Ownership forms a transitive relation in which each object is directly *inside* its owner ($\mathtt{this} \prec^* \mathtt{owner}$). This relation is actually a tree with special owner `world`, for objects shared across the system, as its root ($p \prec^* \mathtt{world}$). Finally, there is a condition which governs whether one object can access another:

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

This says that object α can only access objects 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 [30, 12] 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*. This is illustrated in the leftmost picture in Figure 1, except that there may be several references to object A. If one considers the objects an object owns to be nested inside that object, as denoted by the rounded box, the owners-as-dominators property can be stated as: *no reference can pass through an object boundary from outside of the boundary to inside.*

To do this for classes in a flexible manner we first parameterise a class by an owner, which in the class body is denoted `owner`. Since classes require this owner parameter, types, which are instances of classes, have the owner instantiated. In addition, `this` can be used as an owner for the objects owned by the current object. As in the discussion above, we impose an ordering on owners, and thus `this` is inside `owner`. Since the owner is used as a permission to determine whether or not another object can access an object, we must ensure that subtyping preserves the owner.

In addition to the owner, classes may have other owner parameters. These additional parameters can be seen as providing permissions for an object to access external objects. An ordering may be placed on these owner parameters because they may be used as owner within the class body, and when forming types we must ensure that this ordering is observed. Finally, we require that the owner be inside all of these parameters since it is the only part of a type which cannot be forgotten via subsumption and thus represents the minimum permission required to access an object. Without this requirement, we would lose deep ownership.

In summary, an object may only refer to objects that it owns (*i.e.,* objects of types with `this` as owner) or to objects owned by any of its parameters. More detailed descriptions of ownership types are available in the literature [12, 13].

4. EXTERNAL UNIQUENESS

We propose a new form of uniqueness which we call *external uniqueness* in which a reference is *externally unique* if it is the only external reference to an object. Internal aliasing to the object from its owner representation is still permitted, because such references form part of an aggregate object's implementation, and ought to be preserved to maintain the internal consistency of the aggregate.

It is only in combination with ownership types that we can even consider proposing this new version of uniqueness. Ownership types enable the distinction between the inside and the outside of an object to be made and enforced. In combination with our borrowing mechanism, ownership types actually ensure that an externally unique reference is the only active reference to its target.

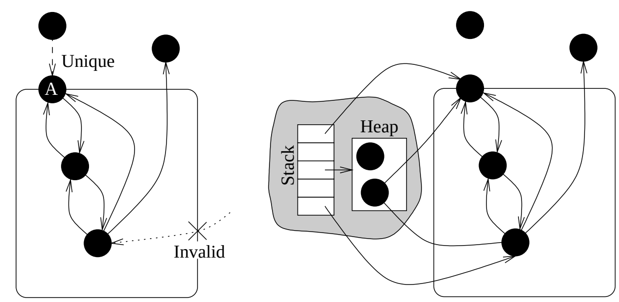

Figure 1: (a) External uniqueness. (b) Borrowing

External uniqueness enjoys a graph-theoretic description which refines that which underlies deep ownership. An externally unique reference corresponds to a *dominating edge* in the object graph. A dominating edge between the root of a graph and a node is an edge which is on all paths from the root to that node. Its removal therefore means that its target is inaccessible to the root of the graph. In Figure 1(a), the dotted edge to object A denotes an externally unique reference. Notice that internal references to A are still permitted. The dominating edge property means that if A is removed, then all objects within the rounded box will become inaccessible to the rest of the system. Since ownership types maintain the owners-as-dominators property, we need only add the constraint that externally unique references are unique to obtain the dominating edge property.

Having dominating edges means that you have dominators also, but these are only possible with deep ownership. Thus systems having only shallow ownership, such as AliasJava [1], cannot support external uniqueness.

4.1 Operations on externally uniques

Externally unique fields and variables support two operations: *movement* and *borrowing*.

Movement allows an externally unique reference to be moved (destructively) from one field or variable into another, possibly losing its uniqueness property. Movement is the only operation which can be directly performed on a unique field, variable or value. Moving an object causes no problems even though there may be internal references, because the dominating edge property guarantees that no access paths to these would be affected by the move. Care must be taken, however, to ensure that the move does not give the object a new owner which would not have permission to access the objects the moved object could access. That is, movement is limited to avoid creating residual aliasing into existing abstractions. In summary, movement is no different from simple movement of unique pointers to objects, though we do have a scope restriction. The consequences are greater: *entire aggregate objects may be moved.*

Borrowing enables a unique reference to be treated temporarily as an ordinary reference for a given lexical scope. This reference may have its fields accessed, its methods called, be passed to methods and even stored in the fields of objects on the heap. The type system ensures that, when the borrowing has finished, no additional references remain and the original reference, or some other reference into the aggregate, can be (re)stored to the place from which it was originally borrowed. The phases of a borrowing can be seen in Figure 1. Diagram (a) indicates the state of play before the borrowing occurs: there are no access paths into the objects inside object A and A is effectively inactive. During the borrowing, Diagram (b), the original reference is cut and placed on the stack in a non-unique variable. The subsequent stack frames may include references to the original object or the objects inside it. In addition, heaps may be created which have a stack-based lifetime, and the objects on these heaps may store references to the original object and the objects inside it. After the borrowing all these stack and heap references become unusable, and the situation returns to Diagram (a).

In short, moving enables unique references to be passed around and borrowing enable them to be operated upon.

4.2 External uniqueness is unique enough

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. But during the borrowing the original unique reference is invalidated, and the object may be treated non-uniquely. 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.

Syntax of Joline

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

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

Table 1: Syntax of Joline

5. A PROGRAMMING LANGUAGE WITH EXTERNAL UNIQUENESS

We present a core object-oriented programming language, called Joline, which has ownership types, externally unique references, and constructs to support operations upon them (Table 1). We now informally describe its characteristics.[1]

The owners of objects are denoted by the syntactic category called *owner*. They are ordered by the relations $\prec^*$ (inside) and $\succ^*$ (outside), as discussed in Section 3. Within a class body, `this` is used as an owner to denote the objects owned by the current instance and `owner` is an owner variable referring to the owner of the current instance — objects with the same owner are called *siblings*. In addition, classes or methods may be parameterised by owner variables, and the borrowing and scoped region construct introduce additional owner parameters which can be used in the appropriate scopes. Finally, the owner `world` is used as the owner of objects potentially shared system-wide.

To account for uniqueness there are two "pseudo-owners" which are only used in the owner position of a type: `unique` indicates an externally unique field, variable, or value. To aid type checking, programs are elaborated to replace all occurrences of `unique` by an instance of the other kind of pseudo-owner, $\texttt{unique}_p$. The p is an owner and acts as a bound on outwards movement of unique references in order to maintain deep ownership — we thus call p the *movement bound* of the unique reference. One difference between p and a regular owner is that p may change when the reference is moved whereas a regular owner is invariant.

A class takes a number of owner parameters (and implicitly `owner`). An ordering on these parameters is specified and must be satisfied when forming types.

Types take the form: $p_0\ c\langle p_{i \in 1..n}\rangle$. p_0 is the owner of objects of that type, or a pseudo-owner to indicate that the reference is unique. No other argument may be a pseudo-owner. All arguments must satisfy the ordering imposed on the parameters of class c for the type to be well-formed.

Methods may also be parameterised by owners. Apart from making them usable for objects with different owners, parameterised methods cannot capture any of its arguments whose owner is a parameter.

Unique values, fields and variables are affected in two ways: *moving* and *borrowing*.

Moving. Moving enables a unique reference to be transferred from one field or variable to another, possibly losing the uniqueness property along the way. To preserve uniqueness the source of the move must be nullified. 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. One does not directly use fields and variables containing uniques values in expressions. Rather one writes `x = y--` or `return x--`, for example, instead of `x = y` or

[1]For simplicity we will say unique for external unique, adding appropriate qualifiers when a distinction need be made.

`return x`, respectively. This distinction may seem trivial, but it simplifies the formal account of our language.

A unique value is consumed by assigning it to a field or variable, or by passing it as an argument to a method. When consuming a unique value, we ask not only for type compatibility, but also that the deep ownership be maintained. This requires a special movement relation, denoted $t \searrow t'$, to be tested in our type system whenever the value of type t is consumed by a target of type t'. The movement relation is one place where the movement bound comes into play.

Borrowing. Performing an operation other than moving a unique reference requires that it first be borrowed. A *borrowing* has the following syntax:

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

A borrowing places the contents of a unique reference (*lval*) into a new variable x. The original l-value is temporarily nullified at run-time so that it cannot be accessed during borrowing to avoid breaking uniqueness. The borrowing of a unique reference thus destroys the only external reference to the target object. α acts as the new owner of this object for the duration of statement `s`. As it is not known outside of the scope, it guarantees that references to the inside of the borrowed object cannot escape. The type of `x` for the duration of this borrowing is a non-unique version of the exact type of *lval* (required for type safety). Since the type of `x` is not unique for the duration of the borrowing, one can access its fields and methods, pass it to methods, or even store it in the heap provided that type conformance is observed. Since the owner α is only available in the scope of the borrowing, the only reference with types containing α still active after the borrowing is the reference stored in x. Internal aliases may exist, but no other aliases to the borrowed object can. The final value in x can then be safely reinstated into *lval*.

Choosing to restore the initial borrowed value would have been consistent with conventional uniqueness, but by allowing the borrowed variable to be modified and then reinstating with the final value, we enable the *lval* to be replaced by another reference into the same aggregate. This is consistent with the definition of external uniqueness.

Scoped regions. Joline also has a construct which enables the creation of objects with lexically bounded lifetimes. Such objects can refer to borrowed objects, thus supporting orthogonality in a manner not permitted in other systems with borrowing. The statement (α) `{` s `}` creates a new owner, valid only for the duration of the statement `s`. This owner is implicitly nested inside some existing owner, giving access to existing objects, but also, through the invariants of ownership types, limiting the lifetime of objects it owns to the scope of `s`. Objects created within the block can be used normally, even passed to sufficiently polymorphic methods, but become unreachable at the end of the block. This feature is not possible without deep ownership, unless one uses an effects system.

Constructors. One additional concern which we must address is whether a constructor call actually returns an externally unique reference to the object it creates. Since a constructor can be viewed as object creation followed by a method call, and the result of object creation is clearly unique, we need only analyse what could go wrong during

```
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;                (*)
  } }
```

Figure 2: External uniqueness

a method call. External uniqueness would be violated if during a method `this` is assigned to a preexisting, external object. Such an object would have to have `owner` as its owner, *i.e.,* the owner of the object being created, and be accessible from the objects passed as arguments to the method. Avoiding this requires a simple fix: *the parameters of a constructor cannot have* `owner` *in their type, except as a movement bound.* This limitation does not prevent an object allowing internal references to itself, but it does prevent references to itself being passed to objects which are externally accessible. Externally created objects may be passed in via unique references, incorporated into the object being created (with `owner` as owner), and only then given a reference to the object being created. Thus any siblings to the newly created object may be externally created and unique, internally created, or added later via borrowing.

Since we can permit constructors to return externally unique references, we omit them from our language description.

6. EXAMPLES AND IDIOMS

We illustrate external uniqueness and the other machinery of Joline with a number of examples, highlighting aspects which would be impossible under existing uniqueness proposals.

External Uniqueness. Consider the code in Figure 2. 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.

Method and Field Access. Accessing a field or method of a unique reference, such as `unique B bus` above, requires that it first be borrowed. For example, calling the `add` method requires the following code:

`borrow bus as <bo> b { b.add(...); }`.

Simulating Borrowing. Other works include a notion of borrowing which permits a unique reference to be passed to a method that guarantees not to create any heap-based aliases to the borrowed entity [9, 1, 7]. We can simulate this using our borrowing and a call to an owner polymorphic method (a method parameterised by owners), without the restriction on heap-based aliases:

```
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'); }
```

Orthogonality. The scoped region construct enables the construction of temporary heap objects which may refer to existing objects, even when those objects may be borrowed unique references. For example, the `print` method in the above example may have the following body.

```
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");
  } }
```

Deep ownership guarantees that no aliasing, such as from the layout manager to the printable, persists when the scoped region exits.

External Initialisation. We would like to initialise the internal representation of an object such as a lexer with an externally specified stream object, without there being any residual aliasing to the stream. This is known as the *initialisation problem* [17]. It can be readily solved with a combination of uniqueness and ownership [1, 7], but we do so with the additional advantages of external uniqueness.

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

Transfer of ownership. Transfer of ownership is an important design pattern in concurrent object-oriented programming [24]. In the following token ring, the token object is passed from one element to the next by calling the `give` method. For example, Figure 3 illustrates the move of `Token` B from `TokenRing` element A to C.

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

This idiom relies on the movement bound of B being the owner of the surrounding object A, that is D. This ensures that B has no references to the objects inside A, thus leaving no residual aliases when it is moved, guaranteed by deep ownership.

External aliasing allows the token object to be a fully fledged aggregate object which may be the resource shared between the elements in the token ring.

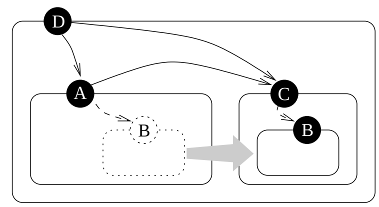

Figure 3: Transfer B from A to its sibling C

Merging Representations. David Holmes posed a challenge when he first saw the original ownership types proposal [14]. He wanted a number of data structures with their own protected, possibly self-referential, internal representation to be merged without copying, with the requirement that at the end of the merge the resulting representation should have the same (expected) encapsulation.

The code in Figure 4 shows the merging of two doubly-linked lists. The phases of operation are: borrow the local representation `(1)`, move the other representation into the temporary owner `ho` `(2)`, 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 the head of argument list is consumed in this operation.

No existing combination of uniqueness and ownership types can handle this example. It is nice to see that we can finally do so in an elegant manner.

7. STATIC SEMANTICS

In this section, we present an overview of the static semantics for Joline. Type checking follows an elaboration phase in which each occurrence of `unique` in the user's program is annotated with a movement bound. Program elaboration is defined in Appendix B: essentially it sets the movement bound to be either `owner` or the innermost owner in scope. In the programming language non-default movement bounds are under programmer control, using syntax `unique[q] Class<o,p>`. When choosing such a movement bound, there is a trade-off to make: an outer `q` enables more movement, but limits what the object can access; an inner `q` enables less movement, but permit more access.

We now focus on the most important type rules for Joline. The remainder are relatively standard and straightforward and can be found in Appendix A. 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:

$$E ::= \emptyset \mid E, x : t \mid E, \alpha \prec^* p \mid E, \alpha \succ^* p$$

The test $\mathsf{unique}(t)$ is true if and only if the type t is unique, that is, that its owner is $\mathtt{unique}_p$. $\mathcal{F}_c$ and $\mathcal{M}_c$ are maps for

```
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;
      }
    }
  }
}
```

Figure 4: Merging representations of two doubly-linked lists

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 owners() when applied to an environment gives all the owners defined in that environment, including the global `world`, and when applied to a type gives all the owner parameters in that type. Substitutions of owners for owner parameters are denoted σ. The act of substituting is denote $\sigma(\cdots)$. 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

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

A type is well-formed whenever the substituted owner parameters satisfy the ordering on parameters prescribed in the class, as we discussed in Section 3. The type can be either unique or not. In the case of unique types, the movement bound must satisfy the same constraint which it would satisfy as an actual owner.

Subtyping

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

Subtyping is derived from subclassing by simply applying the substitution binding arguments to superclass parameters to the extended subclass type. As this corresponds to the composition of two order-preserving functions, it is order preserving. In particular, subtyping preserves the owner, because, as we said, this is the permission required to access objects of this field.[2] Finally, subtyping may preserve uniqueness or forget it, in which case the movement bound becomes the owner.

Movement

$$\frac{E \vdash p' \prec^* p \quad E \vdash p\ c\langle p_{i\in 1..n}\rangle}{E \vdash \texttt{unique}_p\ c\langle p_{i\in 1..n}\rangle \searrow \texttt{unique}_{p'}\ c\langle p_{i\in 1..n}\rangle \qquad E \vdash \texttt{unique}_p\ c\langle p_{i\in 1..n}\rangle \searrow p'\ c\langle p_{i\in 1..n}\rangle}\ \text{(MOVE-UNIQUE)}$$

$$\frac{E \vdash t}{E \vdash t \searrow t}\ \text{(MOVE-NOWHERE)}$$

The movement relation $t \searrow t'$ determines when a value of type t may be assigned to a field or variable, or passed to a method as an argument of type t'. Movement really only permits the movement bound, which is an upper bound, to be replaced by a more inner owner. That is, the type system permits the movement bound to become tighter, but not looser. Movement to objects with the same movement bound is of course permitted — this is a key requirement for our transfer of ownership and merging representation examples.

The second rule simply states that the assignment is possible for non-uniques just in the case that the two types correspond. This simply allows uniform treatment of unique and non-unique types in the type system.

Subsumption is not included within the movement relation, but it can be performed on the value which one wishes to move.

Statements

$$\frac{E \vdash \mathit{lval} : t'\ \ \mathbf{ref} \quad E \vdash e : t \quad E \vdash t \searrow t'}{E \vdash \mathit{lval} = e;\ ; E}\ \text{(STAT-UPDATE)}$$

$$\frac{E, \alpha \prec^* p \vdash s\ ; E' \quad \text{some } p}{E \vdash (\alpha)\ \{\ s\ \}\ ; E}\ \text{(STAT-SCOPED-REGION)}$$

$$\frac{E \vdash \mathit{lval} : \texttt{unique}_p\ c\langle p_{i\in 1..n}\rangle\ \ \mathbf{ref} \qquad E, \alpha \prec^* p, x : \alpha\ c\langle p_{i\in 1..n}\rangle \vdash s\ ; E'}{E \vdash \texttt{borrow}\ \mathit{lval}\ \texttt{into}\ \langle\alpha\rangle\ x\ \{\ s\ \}\ ; E}\ \text{(STAT-BORROW)}$$

The judgement for statements, $E \vdash s\ ; E'$, means that statement s is checked against environment E, and, as s could include variable declarations, results in the environment E' extended with the declared variables and their types.

The rule (STAT-UPDATE) simply enforces that updates can only be performed to l-values, only when the movement from the expression's type to the l-value's type is permitted. (Note that subsumption occurs at expression level.)

The rule (STAT-SCOPED-REGION) introduces a new owner variable for the duration of the given block. The owner p, though unspecified in code, determines which objects any objects created in this scope may access.

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 movement bound of the

[2]The owner may not vary, as Cyclone permits [20], because this would be unsound in our system [13].

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 borrowed ends.

l-values

$$\frac{x : t \in E \quad x \neq \texttt{this}}{E \vdash x : t \;\; \textbf{ref}} \;(\text{LVAL-VAR}) \qquad \frac{E \vdash e : p\ c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \qquad \texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this}}{E \vdash e.f : \sigma^p(t) \;\; \textbf{ref}} \;(\text{LVAL-FIELD})$$

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 $\texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this}$ ensures that types contain `this` in them can only be accessed internally to the object. This means that fields (and methods) containing `this` in their type are private. This is not essential; we could have used *dynamic aliasing* [13], but the type system would have been a little too complex to present our ideas.

Subsumption *does not* apply to l-value types in order to ensure that reinstatement at the end of a borrowing works. Note that not all l-values can be directly treated as a value.

Expressions

$$\frac{E \vdash lval : t \;\; \textbf{ref} \quad \neg\mathsf{unique}(t)}{E \vdash lval : t} \;(\text{EXPR-LVAL}) \qquad \frac{E \vdash lval : t \;\; \textbf{ref} \quad \mathsf{unique}(t)}{E \vdash lval\texttt{--} : t} \;(\text{EXPR-DREAD})$$

$$\frac{\begin{array}{c} E \vdash e : p\ c\langle\sigma\rangle \quad \mathcal{M}_c(m) = \forall(\alpha_i\ \mathsf{R}_i\ p_{i\in 1..n})t_{j\in 1..m} \to t_0 \\ \texttt{this} \in \mathsf{owners}(\forall(\alpha_i\ \mathsf{R}_i\ p_{i\in 1..n})t_{j\in 1..m} \to t_0) \Rightarrow e \equiv \texttt{this} \\ \sigma' = \{\alpha_i \mapsto q_{i\in 1..n}\} \quad E \vdash \sigma'(\alpha_i\ \mathsf{R}_i\ p_{i\in 1..n}) \\ E \vdash e_j : t'_j \quad E \vdash t'_j \searrow \sigma'(\sigma^p(t_j)) \quad \text{for all } j \in 1..m \end{array}}{E \vdash e.m\langle q_{i\in 1..n}\rangle(e_{j\in 1..m}) : \sigma'(\sigma^p(t_0))} \;(\text{EXPR-CALL})$$

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 condition on the second line states that if `this` appears in the type, indicates a method requiring or returning internal representation, may only be used internally, upon `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 in terms of the owners in scope. The value supplied to each argument of the method must have a type which can be moved to the type expected by the method. The method may return a uniquely typed value. It is possible to infer the binding σ' of owners to the parameters of the method.

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.

8. DISCUSSION

We believe that our model of uniqueness is more appropriate and more stable than existing approaches. Furthermore our approach overcomes their limitations. External uniqueness is unique enough, since a unique field or variable contains the only active reference to an unshared object.

8.1 Abstraction Problem

Ownership types allows us to distinguish internal references from external reference. With this, we can permit arbitrary internal aliasing within an uniquely referenced aggregate, without loss of effective uniqueness. Thus, from a software engineering perspective, our proposal is better suited to software evolution than traditional uniqueness, since it allows interfaces to remain the same when the internal implementation changes. Furthermore, we need not clutter interfaces with annotations on methods or constructors indicating which calls will destructively consume their targets, as is the case with other proposals. The price is the loss of uniqueness on `this`, which precludes subjective moves. The gains are much greater.

8.2 Orthogonality Problem

Using our borrowing, owner polymorphic methods, and scoped regions, we overcome the orthogonality problem. Some form of borrowing is essential for mediating between unique and non-unique references. The other two constructs are natural extensions of those in a language with ownership types.

Borrowing permits a unique reference to be treated non-uniquely, and then be passed to owner polymorphic methods, or stored on the heap in objects created within a scoped region. We thus overcome the limitation that borrowing references can never be assigned to fields, without actually needing the concept of a borrowed reference in the language. Ownership types guarantees that any aliases created during the borrowing are not accessible at the end of the borrowing.

8.3 The Definitional Problem

By moving a unique reference into a temporary stack-based variable, thereby nullifying the original l-value, when borrowing, no object can witness a reference non-uniquely and uniquely simultaneously. The reinstatement feature of our borrowing construct makes the operation more convenient to use: programmers need not explicitly reinstate the borrowed value. In fact, it provides additional power. Since externally uniqueness is phrased in terms of references into aggregates, the reinstated value could be another reference into the same aggregate. Our reason for disabling the original unique reference during borrowing is that once borrowed, all the internal references to the object become accessible. Although the scope of these references is restricted, their existence simultaneously with an active unique reference diminishes the power to reason locally about objects. The cost of borrowing in this manner is the introduction of null pointers and potential race conditions to initially obtain the reference.

A more satisfactory manner for dealing with this problem is to use Boyland's Alias Burying [9], which would make all borrowings of the unique references (and all of its siblings) invalid when the original unique reference is accessed. We have not used this approach because it would have complicated our system by requiring at least an effects system.

8.4 Reentrancy

There are interesting consequences to our proposal regarding the reentrancy of objects. Once a reference has been borrowed, the object can only be reentered from an object 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 for reasoning about objects. An important conclusion for concurrency is that an object referred to uniquely can only be entered by one thread.

If we had an effects system, we could safely enable multiple read-only traversals of data referred to with a unique reference, adding a useful and harmless degree of flexibility to our system.

9. RELATED WORK

Table 2 presents a comparison between different proposals in the literature with regard to the kinds of uniqueness, aliasing encapsulation, and borrowing they provide. In addition, the table indicates, where relevant, which of the problems we identified in this paper that the proposals suffer from. 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, 10], or for a recent discussion on aliasing in general, see [13].

We consider two kinds of uniqueness and free:

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

conventional uniqueness fields and variables may contain unique references to objects. 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 objects 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.

We consider five kinds of alias encapsulation:

shallow direct access to certain objects is prevented.

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

effectively deep while the object graph structure does not match that of deep ownership, an effects system prevents the use of the offending references.

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

thread references to objects can be encapsulated within threads.

While shallow encapsulation 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. A similar level of encapsulation can be obtained using existential types. Deep encapsulation 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 [29]. Effectively deep encapsulation is effectively the same as deep encapsulation, except that an effects system is required on top of the type system to control reference usage. Full alias encapsulation [29] is like deep encapsulation except that references to external objects are not permitted from within the encapsulation boundaries. In graph theoretic terms, deep encapsulation imposes that owners are dominators which break path connectivity when removed, whereas full encapsulation imposes that owners are cut points which break graph connectivity when removed.

We consider two kinds of borrowing of unique references:

limited borrowing method parameters, `this`, and/or local variables may borrow a unique reference, but 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 enable the uniqueness invariant to be regained.

Regarding Table 2 we make the following observations:

a) Islands only allow read-only borrowing.

b) Boyland's Alias Burying [9] eliminates the need for destructive reads with a clever static analysis. Using this technique here could have been possible, at the cost of complicating our semantics, since an effects system is required.

c) Guava [3] has an interesting combination of features. So-called *value objects* are passed by moving or copying and thus remain uniquely referred to (but not externally so). The confinement of objects (which are disjoint from value objects) is thread-based, and while borrowing is possible, it applies only to objects not uniques.

d) Adding ordinary uniqueness to shallow ownership types system such as AliasJava's [1] is trivial, since the invariants underlying deep ownership need not be considered. (This caused us to baulk when adding uniqueness.)

e) PRFJ [7] addresses the definitional problem by only allowing unique local variables to be borrowed. The original local variable is thus out of scope during the borrowing.

f) Universes [28] is a cut-down version of ownership types with pervasive read-only references which may access but not modify the internal representation of objects.

g) Balloons seem not to suffer from the orthogonality problem. But their encapsulation model is too restrictive and their checking technique is rather fragile.

h) Pivotal uniqueness enables unique fields to be assigned with newly created objects or null only.

i) Capabilities for sharing defies easy classification.

j) DeLine and Fähndrich [16] give the programmer more control to *adopt* linear references non-linearly and *focus* a non-linear reference linearly, hiding all other references to the same entity. To some degree, this adoption and focus approach achieves what alias burying does, obviating the need to nullify when borrowing. But Vault is an imperative language, whereas Joline is an object-oriented language with classes and subtyping. The inflexibility of classes as a basis for a type system makes this approach difficult to adopt.

Uniqueness and Linearity. Girard's linear logic [18] created the opportunity for stronger control of resources in programming languages. However, a number of researchers

	Feature			Problem		
	Uniqueness	Encapsulation	Borrowing	Abstraction	Orthogonality	Definitional
This paper	External	Deep	Orthogonal	√	√	♢
Ownership Types [14, 12, 13]	None	Deep	–	–	–	–
Ownership Types [12, 11]	None	Shallow	–	–	–	–
Flexible Alias Protection [29]	Free	Deep	–	×	–	–
Islands [21] [a]	Conventional	Full	∼Limited	×	–	–
OOFX/Alias Burying [9, 19] [b]	Conventional	None	Limited	×	×	☺
Eiffel* [27]	Conventional	None	Limited	×	×	♡
Guava [3] [c]	∼Conventional	Thread	–	×	×	–
AliasJava [1] [d]	Conventional	Shallow	Limited	×	×	♡
PRFJ [7] [e]	Conventional	Effectively Deep	Limited	×	×	♢
Universes [28] [f]	None	∼Deep	–	–	–	–
Balloons [2] [g]	Conventional	Full	Limited	×	√	♡
Virginity [26]	Free	None	Limited	×	×	–
Pivot Uniqueness [25] [h]	<Conventional	Shallow	Limited	×	×	♡
Capabilities for sharing [10] [i]	∼Conventional	∼Shallow	∼Limited	–	–/×	–/♡
Vault [15, 16] [j]	∼Conventional	Shallow	Orthogonal	–	√	☺

Table 2: Comparison. Legend: – not applicable; × suffers from problem; √ provides adequate solution; ♡ weakened uniqueness; ♢ nullification or similar; ☺ avoids problem; *a-j* see text. (The perfect score is External-Deep-Orthogonal-√-√-☺)

have realised that programming with uniqueness or linearity in its strictest form is painful [34, 4]. Wadler's `let!` construct, quasi-linear types [23], and Vault's adoption and focus [16], 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.

Object models. Compared with object-oriented data models [31], our conception of ownership can be seen as a realisation of the IS-PART-OF relationship, or aggregate, except that (1) ownership is decoupled from the reference structure, and (2) all references to an owned object are confined within its owner. Adding external uniqueness, we obtain a relationship similar to the *exclusive components* of Kim, Bertino, and Garza [22] which permit arbitrary *weak references* (non-component references), whereas we allow only those which are confined within an object.

Region-based memory management. Our scoped region construct is similar to lexically scoped `letregion` construct used in region-based memory management [32, 33], except that our construct is under programmer control, as in Cyclone [20], the technical machinery is based on reference structure rather than effects, and our aim is for orthogonality rather than memory management. We note in passing that regions systems lack the deep encapsulation of objects within other objects present in ownership types [14]. An object can be seen as having a region (referred to using `this` as an owner) containing the objects it owns. From this we uncover 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 have plans to investigate this further, although recently Boyapati *et.al.*, have made steps in this direction [8].

Applications of ownership and uniqueness. Ownership typing has many applications: reasoning about programs [13, 19, 28], alias management [14, 29], eliminating data races [7] and deadlock [5], program understanding [1], and for safe lazy upgrades in object-oriented databases [6]. Other applications which could be explored for ownership typing, especially now that we have uniqueness, include enforcing protocols [15], memory management [20], and mobility.

10. CONCLUSIONS AND FUTURE WORK

In this paper we identified a number of problems with existing approaches to unique references in object-oriented programming languages. We counter two of these by devising an alternative model of uniqueness based on ownership types. While on the surface our proposal seems to violate the very essence of uniqueness, permitting internal references to an object referred to uniquely, it turns out that an externally unique pointer is effectively unique because it is always the only active pointer. This was all done for an object-oriented programming language with subtyping.

One major problem remains for future work: addressing the definitional problem may simply involve adopting Alias Burying [9] or ideas from Vault [16], at the cost of a more complicated formalism.

This work forms part of a general program to localise references, behaviour, control, *etc.* in order to simplify reasoning. We wish to further our work in this direction. In addition, we are in the process of adding external uniqueness to our compiler for `Joe` [13]. We can then explore more programming patterns and apply our ideas to applications in concurrency, mobility, memory management, *etc.*

The full technical development of external uniqueness including dynamic semantics, and proofs of soundness and other properties will appear as the second author's licentiate thesis in March or April 2003.

Acknowledgements. Thanks to James Noble for a very comprehensive reading of this paper and comments to match, to Chandra Boyapati for clarifications regarding his work, to the anonymous referees for many useful pointers, and especially to John Tang Boyland for rectifying our misconceptions regarding the definitional problem and alias burying.

11. 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] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, November 2002.

[6] 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.

[7] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA Proceedings*, 2001.

[8] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. Available from `http://www.pmg.lcs.mit.edu`, November 2002.

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

[10] John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *ECOOP Proceedings*, June 2001.

[11] David Clarke. An object calculus with ownership and containment. In *Foundations of Object-oriented Programming (FOOL8)*, London, January 2001.

[12] David Clarke. *Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.

[13] David Clarke and Sophia Drossopolou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, November 2002.

[14] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA Proceedings*, 1998.

[15] 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.

[16] 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.

[17] 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.

[18] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[19] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99*, 1999.

[20] 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.

[21] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA Proceedings*, November 1991.

[22] Won Kim, Elisa Bertino, and Jorge F Garza. Composite objects revisited. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 337–347, Portland, Oregon, 1989.

[23] Naoki Kobayashi. Quasi-linear types. In *26th ACM Symposium on Principles of Programming Languages*, January 1999.

[24] Doug Lea. *Concurrent-Programming in Java: Design Principles and Patterns*. Java Series. Addision-Wesley, 1998.

[25] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *PLDI*, June 2002.

[26] 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.

[27] Naftaly Minsky. Towards alias-free pointers. In *ECOOP Proceedings*, July 1996.

[28] 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.

[29] 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.

[30] John Potter, James Noble, and David Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, Adelaide, Australia, November 1998. IEEE Press.

[31] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[32] J.-P Talpin and P. Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[33] Mads Tofte and Jean-Pierre Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1997.

[34] Phil Wadler. Linear types 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.

APPENDIX

A. TYPE RULES

The complete collection of type rules are given here.

$E \vdash \diamond$ — Good environment

$$\frac{}{\emptyset \vdash \diamond}\ (\text{ENV-}\emptyset) \qquad \frac{E \vdash t \quad x \notin \mathrm{dom}(E)}{E, x : t \vdash \diamond}\ (\text{ENV-}x) \qquad \frac{E \vdash p \quad \alpha \notin \mathrm{dom}(E)}{E, \alpha\, \mathsf{R}\, p \vdash \diamond}\ (\text{ENV-}\alpha)$$

$E \vdash p$ — Good owner

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

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

$$\frac{\alpha \prec^* p \in E}{E \vdash \alpha \prec^* p}\ (\text{IN-ENV1}) \qquad \frac{\alpha \succ^* p \in E}{E \vdash p \prec^* \alpha}\ (\text{IN-ENV2})$$

$$\frac{E \vdash p}{E \vdash p \prec^* \mathtt{world}}\ (\text{IN-WORLD}) \qquad \frac{\mathtt{this} : t \in E}{E \vdash \mathtt{this} \prec^* \mathtt{owner}}\ (\text{IN-THIS})$$

$$\frac{E \vdash p}{E \vdash p \prec^* p}\ (\text{IN-REFL}) \qquad \frac{E \vdash p \prec^* q \quad E \vdash q \prec^* q'}{E \vdash p \prec^* q'}\ (\text{IN-TRANS})$$

$\vdash P$ — Good Program

$$\frac{\vdash \mathit{class} \text{ for all } \mathit{class} \in P \quad \vdash s\ ; E \quad E \vdash e : t}{\vdash \mathit{class}_{i \in 1..n}\ s\ e : t}\ (\text{PROGRAM})$$

$\vdash \mathit{class}$ — Good Class

$$\frac{}{\vdash \mathtt{class\ Object\ \{\ \ \}}}\ (\text{CLASS-OBJECT})$$

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

$E \vdash \mathit{meth}$ — Good Method

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

$E \vdash t$ — Good Type

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

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

$$\frac{E \vdash t}{E \vdash t \leq t}\ (\text{SUB-REFL}) \qquad \frac{E \vdash t \leq t' \quad E \vdash t' \leq t''}{E \vdash t \leq t''}\ (\text{SUB-TRANS})$$

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

$E \vdash t \searrow t'$ — t may move to t'

$$\frac{E \vdash p' \prec^* p \quad E \vdash p\ c\langle p_{i \in 1..n}\rangle}{\begin{array}{c} E \vdash \mathtt{unique}_p\ c\langle p_{i \in 1..n}\rangle \searrow \mathtt{unique}_{p'}\ c\langle p_{i \in 1..n}\rangle \\ E \vdash \mathtt{unique}_p\ c\langle p_{i \in 1..n}\rangle \searrow p'\ c\langle p_{i \in 1..n}\rangle \end{array}}\ (\text{MOVE-UNIQUE})$$

$$\frac{E \vdash t}{E \vdash t \searrow t}\ (\text{MOVE-NOWHERE})$$

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

$$\frac{E \vdash \diamond}{E \vdash \mathtt{skip;}\ ; E}\ (\text{STAT-SKIP}) \qquad \frac{x \notin \mathrm{dom}(E) \quad E \vdash e : t}{E \vdash t\ x\ = e;\ ; E, x : t}\ (\text{STAT-LOCAL})$$

$$\frac{E \vdash e : t}{E \vdash e;\ ; E}\ (\text{STAT-EXPR}) \qquad \frac{E \vdash \mathit{lval} : t'\ \mathbf{ref} \quad E \vdash e : t \quad E \vdash t \searrow t'}{E \vdash \mathit{lval} = e;\ ; E}\ (\text{STAT-UPDATE})$$

$$\frac{E \vdash s_1\ ; E'' \quad E'' \vdash s_2\ ; E'}{E \vdash s_1\ s_2\ ; E'}\ (\text{STAT-SEQUENCE}) \qquad \frac{E, \alpha \prec^* p \vdash s\ ; E' \quad \text{some } p}{E \vdash (\alpha)\ \{\ s\ \}\ ; E}\ (\text{STAT-SCOPED-REGION})$$

$$\frac{E \vdash s\ ; E'}{E \vdash \{s\}\ ; E}\ (\text{STAT-BLOCK}) \qquad \frac{E \vdash e : \mathtt{bool} \quad E \vdash s_1\ ; E' \quad E \vdash s_2\ ; E''}{E \vdash \mathtt{if}\ (e)\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \}\ ; E}\ (\text{STAT-IF-THEN-ELSE})$$

$$\frac{\begin{array}{c} E \vdash \mathit{lval} : \mathtt{unique}_p\ c\langle p_{i \in 1..n}\rangle\ \mathbf{ref} \\ E, \alpha \prec^* p, x : \alpha\ c\langle p_{i \in 1..n}\rangle \vdash s\ ; E' \end{array}}{E \vdash \mathtt{borrow}\ \mathit{lval}\ \mathtt{as}\ \langle\alpha\rangle\ x\ \{\ s\ \}\ ; E}\ (\text{STAT-BORROW})$$

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

$$\frac{x : t \in E \quad x \neq \mathtt{this}}{E \vdash x : t\ \mathbf{ref}}\ (\text{LVAL-VAR}) \qquad \frac{E \vdash e : p\ c\langle\sigma\rangle \quad \mathcal{F}_c(f) = t \quad \mathtt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \mathtt{this}}{E \vdash e.f : \sigma^p(t)\ \mathbf{ref}}\ (\text{LVAL-FIELD})$$

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

$$\frac{E \vdash \mathit{lval} : t\ \mathbf{ref} \quad \neg\mathsf{unique}(t)}{E \vdash \mathit{lval} : t}\ (\text{EXPR-LVAL}) \qquad \frac{E \vdash \mathit{lval} : t\ \mathbf{ref} \quad \mathsf{unique}(t)}{E \vdash \mathit{lval}\mathtt{--} : t}\ (\text{EXPR-DREAD})$$

$$\frac{\mathtt{this} : t \in E}{E \vdash \mathtt{this} : t}\ (\text{EXPR-THIS}) \qquad \frac{E \vdash t}{E \vdash \mathtt{new}\ t : t}\ (\text{EXPR-NEW}) \qquad \frac{E \vdash t}{E \vdash \mathtt{null} : t}\ (\text{EXPR-NULL})$$

$$\frac{\begin{array}{c} E \vdash e : p\ c\langle\sigma\rangle \quad \mathcal{M}_c(m) = \forall(\alpha_i\, \mathsf{R}_i\, p_{i \in 1..n}) t_{j \in 1..m} \to t_0 \\ \mathtt{this} \in \mathsf{owners}(\forall(\alpha_i\, \mathsf{R}_i\, p_{i \in 1..n}) t_{j \in 1..m} \to t_0) \Rightarrow e \equiv \mathtt{this} \\ \sigma' = \{\alpha_i \mapsto q_{i \in 1..n}\} \quad E \vdash \sigma'(\alpha_i\, \mathsf{R}_i\, p_{i \in 1..n}) \\ E \vdash e_j : t'_j \quad E \vdash t'_j \searrow \sigma'(\sigma^p(t_j)) \quad \text{for all } j \in 1..m \end{array}}{E \vdash e.m\langle q_{i \in 1..n}\rangle(e_{j \in 1..m}) : \sigma'(\sigma^p(t_0))}\ (\text{EXPR-CALL})$$

$$\frac{E \vdash e : t \quad E \vdash t \leq t'}{E \vdash e : t'}\ (\text{EXPR-SUBSUMPTION})$$

B. PROGRAM ELABORATION

Elaboration

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

$$\mathsf{E}\left(\begin{array}{l}\texttt{class}\ c\langle \alpha_i\ \mathsf{R}_i\ p_{i\in 1..m}\rangle\ \texttt{extends}\ c'\langle p'_{i'\in 1..n}\rangle \\ \quad \texttt{\{}\ fd_{j\in 1..m}\ meth_{k\in 1..m}\ \texttt{\}}\end{array}\right) =$$

$$\begin{array}{l}\texttt{class}\ c\langle \alpha_i\ \mathsf{R}_i\ p_{i\in 1..m}\rangle\ \texttt{extends}\ c'\langle p'_{i'\in 1..n}\rangle \\ \quad \texttt{\{}\ \mathsf{E}_{\texttt{owner}}(fd_j)_{j\in 1..m}\ \mathsf{E}_{\texttt{owner}}(meth_k)_{k\in 1..m}\ \texttt{\}}\end{array} \quad *$$

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

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

$$\begin{array}{rcl}
\mathsf{E}_p(x) &=& x \\
\mathsf{E}_p(e.f) &=& \mathsf{E}_p(e).f
\end{array}$$

$$\begin{array}{rcl}
\mathsf{E}_p(\texttt{this}) &=& \texttt{this} \\
\mathsf{E}_p(lval\texttt{--}) &=& \mathsf{E}_p(lval)\texttt{--} \\
\mathsf{E}_p(\texttt{new}\ t) &=& \texttt{new}\ \mathsf{E}_p(t) \\
\mathsf{E}_p(\texttt{null}) &=& \texttt{null}
\end{array}$$

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

$$\begin{array}{rcl}
\mathsf{E}_p(\texttt{skip;}) &=& \texttt{skip;} \\
\mathsf{E}_p(t\ x\ = e\texttt{;}) &=& \mathsf{E}_p(t)\ x\ = \mathsf{E}_p(e)\texttt{;} \\
\mathsf{E}_p(e\texttt{;}) &=& \mathsf{E}_p(e)\texttt{;} \\
\mathsf{E}_p(lval = e\texttt{;}) &=& \mathsf{E}_p(lval) = \mathsf{E}_p(e)\texttt{;} \\
\mathsf{E}_p(s\ s') &=& \mathsf{E}_p(s)\ \mathsf{E}_p(s')
\end{array}$$

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

$$\begin{array}{rcll}
\mathsf{E}_p((\alpha)\ \texttt{\{}\ s\ \texttt{\}}) &=& (\alpha)\ \texttt{\{}\ \mathsf{E}_\alpha(s)\ \texttt{\}} & * \\
\mathsf{E}_p(\texttt{\{}\ s\ \texttt{\}}) &=& \texttt{\{}\ \mathsf{E}_p(s)\ \texttt{\}} &
\end{array}$$

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

$$\begin{array}{rcll}
\mathsf{E}_p(\alpha) &=& \alpha & \\
\mathsf{E}_p(\texttt{owner}) &=& \texttt{owner} & \\
\mathsf{E}_p(\texttt{world}) &=& \texttt{world} & \\
\mathsf{E}_p(\texttt{unique}) &=& \texttt{unique}_p & * \\
\mathsf{E}_p(p_0\ c\langle p_{i\in 1..n}\rangle) &=& \mathsf{E}_p(p_0)\ c\langle \mathsf{E}_p(p_i)_{i\in 1..n}\rangle &
\end{array}$$

Rules marked with a $*$ are the ones which are interesting.