External Uniqueness

Tobias Wrigstad [a,*] Dave Clarke [b]

[a] *DSV Stockholm University/Royal Institute of Technology, Sweden*
[b] *CWI, The Netherlands*

Abstract

External uniqueness is a novel form of unique pointers that fit more natural with object-oriented programming. External uniqueness enforces the structural property that a unique pointer is a *dominating node* to the aggregate it references—there may be no other pointer from outside the aggregate to objects in it in the rest of the system. This paper completes previous work by the authors (Clarke and Wrigstad, ECOOP 2003) with the full semantics of the Joline language and proofs of the structural invariants.

Key words: Alias Management, Ownership, Uniqueness, Object-Oriented Programming

1 Introduction

The two traditional approaches to adding unique references to object-oriented programs [1,2] are flawed. Not only do they provide only *shallow* uniqueness, in the sense that a unique object's representation can be arbitrarily shared, but the mechanisms used to enable uniqueness in a modular fashion—class-level annotations and method-level annotations—break abstraction: *as software evolves, programs which use uniqueness are forced to change their interfaces when purely internal implementation changes are made.*

Combining uniqueness with a system that provides strong encapsulation, such as Ownership Types [3,4,5] can overcome the first problem as such systems

* Corresponding author. Address: DSV Stockholm University/Royal Institute of Technology, Forum 100, 164 60 Kista, SWEDEN
Email addresses: tobias@dsv.su.se, dave.clarke@cwi.nl (Dave Clarke).

provide a stronger notion of aggregate. Two examples of systems using ownership and uniqueness are AliasJava [6] and SafeJava [7]. Sadly, these languages both suffer from the abstraction problem inherent in traditional uniqueness.

In previous work [8], we introduced the notion of *external uniqueness*. An externally unique pointer is the only pointer to an aggregate visible outside the aggregate itself. In the system we proposed, an externally unique pointer cannot be used to call methods or access fields. The only way to dereference a unique pointer and access the aggregate's internals is to temporarily convert it into a regular non-unique pointer for a specific scope. Consequently, while the externally unique pointer is in place, any internal alias of it will be invisible outside the aggregate and therefore, we conclude that external uniqueness is not weaker that regular uniqueness. External uniqueness does not suffer from problems with breaking abstraction and is to our mind more suitable for object-oriented systems than traditional uniqueness.

In this paper, we extend our previous work on external uniqueness with the full semantics of our Joline language that sports externally unique pointers, as well as proof that externally unique pointers are dominating edges, the graph-theoretic property of and structural invariant of our proposal.

Outline In Section 2, we discuss the problems of adding uniqueness to object-oriented programming languages. In Section 3 we introduce ownership-types to set the scene for Section 4 which introduces external uniqueness. Section 5 presents the Joline language and Section 6 discusses its soundness. Section 7 discusses related work and Section 8 concludes.

2 Adding Uniqueness To OOPLs

Uniqueness is a simple but powerful concept. Basically, a reference annotated with a *unique* keyword pointing to some object o is the only reference in the entire system pointing to o. Uniqueness gives powerful reasoning guarantees—for example, the full effects of updating and object is visible, and the (local) state of the object pointed to by a unique variable v will remain unchanged when evaluating a statement that does not explicitly mention v. This is useful in for example specifications when constructing type systems. The systems using uniqueness are many and its benefits have been described elsewhere [9,1,2,10,11,6,12,8,13,14].

A study by Noble and Potanin [15] suggests that uniqueness as a concept fits well with the current ways of constructing object-oriented software: inspection of heap dumps of running programs from the Purdue Benchmark Suite has

shown that as much as 85% of all objects are uniquely referenced in a program. This study is optimistic, as uniqueness violations could occur in-between the heap dumps and this go undetected by the analysis. However, more fine-grained, less optimistic studies of smaller programs have shown similar results, suggesting that Noble and Potanin's optimistic results are correct. All in all, it would seem that we stand to gain from adding support for unique pointers in object-oriented programming languages. However, this is not entirely straight-forward.

In short, there are three (closely related) problems with adding uniqueness to object-oriented languages:

(1) the implicit destruction of unique receivers,
(2) how to restrict a method's use of its receiver to allow it to be automatically reinstated after a call, and
(3) how to make sure that a constructor returns a unique reference.

As is visible from the above list, all problems are related to dealing with `this`. Different approaches in the literature reflect the treatment of `this` by annotations on a class' interface in two ways. The annotations are necessary to achieve modular checking of the uniqueness invariant and are either at class-level or method level:

Via class annotation Classes are divided into two kinds, those whose instances may assign `this` internally, and those whose instances may not. Only instances of the latter may be referenced uniquely [1].

Via method annotation Methods are annotated to indicate that they may consume `this` [9,2]. Calling such a method requires that its target be destructively read (or equivalent, in the presence of an effective uniqueness scheme).

We now detail the description of these approaches to show how they both create a problem with abstraction in the presence of evolving code.

2.1 Class-level Annotations

Class-level uniqueness annotations, proposed by Minsky [1] in Eiffel*, decorates class declarations and controls whether instances of a particular class can or cannot be uniquely referenced. In the example in Figure 1, a class `Server` is annotated with the uniqueness keyword allowing its instances to be uniquely referenced. The unique annotation requires that all methods in the server class are anonymous (do not capture `this`), that is they don't assign `this`, or pass `this` as an argument to a method. As all methods in a unique class are anonymous, `this` can be safely used as a receiver in all methods as

```
// The class declaration is annotated with a unique keyword
unique class Server extends Object {
  Int noConnections = 0;

  void connect(Client client) { // Invalid method
    client.setManager(this ); // -- won't compile (see Figure text)
  }

  Int getConnections() { // Good method
    return this.noConnections;
  }
}
```

Figure 1. Using class-level annotations to control how `this` is treated internally. The `connect()` method will not compile as it is creates an alias to `this` and passes it as an argument to the `setManager()` method in `client`.

no method will create an alias to it.

The class-level annotation approach has several problems: it violates abstraction by making the uniqueness keyword reflect aspects of the class' implementation; it is inflexible; and it places large constraints on the evolution of a program.

2.1.1 Violating the Principle of Abstraction

Using class annotations, whether or not an object can be uniquely referenced becomes a *property of the class*, or more specifically, of how the class' methods can treat the `this` variable. Thus, internal implementation details are visible in the interface, which is a violation of the principle of abstraction as this annotation controls how the object can be used externally. The negative effects of this will be addressed again shortly and compared to a similar problem for method level annotations.

2.1.2 Inflexibility

Classes whose instances should be possible to reference uniquely may only contain anonymous methods. A single method that needs to create an alias to `this` in a class will thus preclude uniquely referenced instances of the class, which is clearly very inflexible. If a class' instances should be both uniquely and non-uniquely referenced, methods invoked on non-unique references still would not be allowed to alias `this`.

Last, instances of classes not annotated with the `unique` keyword cannot be uniquely referenced, even if the class' implementation would allow it as only in-

```
neverunique class A extends Object {
  void aliasingMethod() {
    B temp = this; // Creates an alias to this
  }
}

// changing uniqueness declaration for the subclass
unique class B extends A {  }

unique B b = new B();
m.aliasingMethod(); // invalidates uniqueness
```

Figure 2. Subclassing with class-level annotations. Instances of class `A` are never unique. However, if we are allowed to subclass `A` with a unique class `B`, uniqueness of unique references to `B` objects can be invalidated if a method call binds to a method defined in `A`.

```
class Server extends Object {
  Int noConnections = 0;

  void connect(Client client) consumes {
    client.setManager(this );
  }

  Int getConnections() anonymous {
    return this.noConnections;
  }
}
```

Figure 3. Using method-level annotation to control subjective treatment of `this`.

stances of classes *annotated* with unique are allowed to be referenced uniquely.

2.1.3 Constraining Evolution

As is illustrated in Figure 2, the (non)uniqueness annotation must be preserved through subclassing as uniqueness could otherwise be invalidated by overriding methods that created aliases to `this`. This makes extension via subclassing harder or less powerful since the annotation of the superclass must be respected by all subclasses.

It might be possible to allow unique classes to have non-unique subclasses as the implementation of the unique superclasses work even if `this` is not unique, but as Figure 2 clearly shows, not the other way around.

2.2 Method-level Annotations

With method-level annotations, used by for example Hogg [9] and Boyland [2], each method is annotated to reflect its treatment of `this` and allow only methods that do not capture this to be invoked on unique receivers. Thus, method-level annotations allows an object to be uniquely referenced regardless of its class' implementation. This is more fine-grained and thus more flexible than class-level annotation. The price is increased syntactic overhead. Figure 3 shows the `Server` class from Figure 1 using `consumes` and `anonymous` annotations, similar to Boyland's proposal.

In Figure 3, `getConnection()` is now annotated with the `anonymous` keyword, meaning that it does not alias `this`. The `connect()` method is annotated with `consumes` meaning that it may create an alias to `this`. Validity of these annotations can be controlled by a simple compile-time check

While avoiding many of the problems caused by the coarseness of class-level annotations, method-level annotations are not problem-free. For example, overriding methods suffer similar constraints as subclasses in the class-level example with respect to preserving annotations.

Most importantly, however, the abstraction problem persists as the annotation of a method reflects its implementation. Thus, internal implementation details are again visible in the interface leading to problems when implementation details change over time.

2.3 Uniqueness and Abstraction

In this section, we detail the discussion about the abstraction problem with both styles of annotation above. In both cases, and for methods and constructors alike, a problem surfaces when the implementation of a class changes the way it uses `this` which leads to a violation of the principle of abstraction. We will now examine this problem in more detail, recapping the original arguments of Clarke and Wrigstad [16].

For concreteness, assume that we have the following class with a single method:

```
class BlackBox {
  void xyzzy() {
    ...  // unknown implementation
  }
}
```

and at some other place in the program, a unique variable or field

```
unique BlackBox bb;
```

As the enclosing software system evolves, a later version of `BlackBox` requires an addition to `xyzzy()` that includes the line:

```
OtherBlackBox obb = new OtherBlackBox(this);
```

Thus, the new implementation of `xyzzy()` now creates an alias to the receiver. Under the existing proposals, this forces a change of `BlackBox`'s interface. The consequences of this vary depending on whether we are using class-level annotations or method-level annotations, as we will see in the following sections.

2.3.1 With Class Annotations

As it is visible in a class' interface how it treats its `this` variable, changes to how `this` is treated might lead to problems with changes in the interface.

Using class annotations `BlackBox` would have been annotated **unique** to show that its instances can be uniquely referenced. As a consequence of the addition to `xyzzy()`, instances of `BlackBox` can no longer be uniquely referenced which is reflected in change of the class header from **unique class** `BlackBox` to **neverunique class** `BlackBox`.

Consequently, all variable declarations of type **unique** `BlackBox`, such as **unique** `BlackBox bb;` above would no longer be valid in the program, and must have their uniqueness stripped to compile. Depending on how uniqueness is realised, it may also be the case that all destructive reads of `BlackBox` objects throughout the entire program would have to be changed to ordinary reads, perhaps with destructive reads performed manually. Obviously, these changes could propagate through the entire program.

2.3.2 With Method Annotations

As it is visible in a method header how the method treats the `this` variable, changes to how `this` is treated might lead to problems with changes in the method header.

Using method annotations, the `xyzzy()` method would have been annotated **anonymous**. However, the addition to the method forces it to be changed to **consumes**. Potentially, this forces much fewer changes to the program compared to class-level annotations, as instances of `BlackBox` may still be

uniquely referenced. However, the call `bb.xyzzy()` will now create an alias to its receiver requiring that the variable `bb` is nullified to preserve uniqueness. Depending on the realisation of uniqueness this change from **anonymous** to **consumes** might propagate as an addition of a destructive read operation to the calls. If this is not the case, the result is even more drastic, as the behaviour of the method call has changed silently from the original program to consume its target. This is both awkward and counter-intuitive.

2.4 Conclusion

Uniqueness is simple and powerful concept. However, in both traditional ways of adding uniqueness to object-oriented system, a purely internal change to the implementation of the `BlackBox` class can force 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, also it means that objects cannot be treated like black boxes, because:

> *Software evolution which changes the uniqueness aspects of an object's implementation can force changes in the object's interface, which then propagates changes throughout the program.*

Thus extant uniqueness proposals break abstraction.

In conclusion, it seems that current approaches to uniqueness are ill-fitted to the object-oriented setting.

As our proposed solution to this problem is based on ownership types, we now briefly describe ownership types to set the state for external uniqueness.

3 Ownership Types

Deep ownership types [4] enforces the conceptual structural property that an object's representation (the subobjects conceptually belonging to it) is *inside* its *enclosing* object and cannot be exported outside it. This is called the *owners-as-dominators property* and gives strong encapsulation.

Ownership types introduces the notion of objects as *owners* and representation objects are *owned* by their enclosing objects. Classes are parameterised by ownership information and types are formed by instantiating these parameters with actual owners.

Deep ownership enable constraining of the object graph by capturing the nesting of objects in the types in a simple and elegant manner. Representation objects are ordered *inside* their enclosing objects, and references to representation are not allowed to flow to the outside world. As the nesting is captured in the class declarations, the nesting information is propagated through the program, giving control over the global structure of the object graph. By prohibiting references owned by some owner x to flow to objects outside x, a strong, but flexible, containment invariant is achieved that cannot be circumvented [1].

An owner can be seen as the permission to reference a group of objects. Types are formed from classes and *owner parameters*, which serve as placeholders to give permissions to reference external objects. Thus, a type does not denote a set of possible instances of a class, but a set of possible instances of a class *with a particular set of permissions to reference other objects.* Types with different owner parameters are not compatible and references of types with different owners cannot be aliases as shown by Clarke and Drossopoulou [17].

3.1 Annotating Classes with Ownership Information

To be able to statically control ownership nesting, class declarations are extended with annotations which describe the relations between owner parameters to thread nesting information through the program. As an example, the class `StringList` below takes two owner parameters where the first parameter is nested inside the second and both are outside the owner `owner`.

```
class StringList< owner1 outside owner, owner2 outside owner1 >
```

Ownership parameters of a class must always be outside the implicit owner `owner`, the owner of the instance. This is key to avoiding the problem of indirect representation exposure in shallow ownership [14].

The omnipresent owner `world` is *outside* all owners, is visible in all scopes and denotes global objects, accessible everywhere in the object graph. In addition to `owner` and `world`, a class body has access to the owners declared in its class header, and the owner `this`, which denotes the current object and is *inside* `owner`.

For subclassing, the extends clause is extended with a mapping relation from the owners of the subclass to those of the superclass. The number of owner

[1] Circumventing the containment is however possible in *shallow* ownership, causing indirect representation exposure [5]. Here we only consider non-shallow ownership.

parameters in a subclass may grow or shrink depending on the relations between the owners in the super class.

```
class List<some outside owner> extends Object { ... }
class StringList<owner1 outside owner, owner2 outside owner1>
                              extends List<owner2> { ... }
```

The owner must be preserved through subtyping as it acts as the permission governing access to the object. Preserving it by subsumption is a key to achieving a sound system as shown by Clarke [5]. In the example above, `StringList`'s second owner parameter will be mapped to `some` when viewed as its super class. This is valid if `owner2` is outside `owner`, a requirement derived from `List`'s class header. That the requirement is fulfilled can be derived from the class header of `StringList` as nesting is a transitive relation (`owner2` is outside `owner1` and `owner1` is outside `owner` implies `owner2` is outside `owner`).

3.2 Forming Types

Types have following the syntax:

$$\texttt{owner : ClassName<owner}_1\texttt{, ..., owner}_n\texttt{>}$$

where `owner` is the owner of the type, `ClassName` is a class name and $\texttt{owner}_{1..n}$ are visible permissions (in the current context) to reference external objects.

When forming types from a class, the nesting requirements of the owners in the class' header must be satisfied by the owners in scope to which they are bound. The object graph is well-constructed with respect to the nesting requirements specified in the classes.

Below are a few examples of Joline types with ownership using the recent class declaration examples.

```
class StringList<owner1 outside owner, owner2 outside owner1>
                                extends List<owner2> {
  this : StringList<owner, owner1>     representation;
  owner : StringList<owner1, owner2>   outgoing;
  // owner1 : StringList<this, owner2> illegal;
  // world : StringList<this, owner2>  alsoIllegal;

  owner : List<owner2>  super = outgoing;
}
```

In the code example above, the third and fourth variable declarations are illegal as the owners in scope do not satisfy the requirements of the class header of `StringList` as `this` is inside both `world` and `owner1`.

The variable `representation` holds a representation object with permission to reference back to the object itself as it is parameterised with `owner`.

The variable `outgoing` has the same type as the current instance. As the type of `outgoing` does not have `this` as its owner, it cannot point to a representation object as all representation objects are owned by `this` and types with different owners are not assignment compatible. Furthermore, the type is not given explicit permission to reference `this` (`this` is not an owner in the type). This means that references to representation cannot be stored in an object referenced by the variable. Such violations are statically checkable and will not compile. Actually, having this in the type of `outgoing` would not be valid as that would give an external object permission to reference the current representation. This is prevented by the restriction that the owner must be inside all other owner parameters.

Last, `super` shows subsumption—the type `owner : List< owner2 >` is a super type of `owner : StringList< owner1, owner2 >` and we can therefore assign from `outgoing` to `super`. Note the remapping and hiding of owner parameters as discussed on the previous page.

4 External Uniqueness

The idea behind external uniqueness is simple: exclude internal aliases in the uniqueness definition. Thus, *a reference is unique if it is the only external reference to the object.* Hence the name external uniqueness.

Our definition is a minor tweak, but has major consequences. In short, external uniqueness *overcomes the abstraction problem* and allows a flexible form of aggregate uniqueness that *relaxes the uniqueness definition for internal pointers without compromising effective uniqueness.* Some subsequent proposals [7] have adopted our approach.

Naturally, our proposal requires a system with strong encapsulation, such as ownership types to make sure that internal references stay internal, and to be able to distinguish an object's inside from its outside. We can implement external uniqueness with a simple extension to ownership: *unique owners.*

Externally unique references are denoted using types such as $\texttt{unique}:c\langle p_{i\in 1..n}\rangle$. The unique annotation can only appear in the owner position, and thus no

$p_{i \in 1..n}$ may be `unique`. As ownership types maintain the dominators property, to obtain external uniqueness we need only add machinery to ensure the uniqueness of references of unique type when viewed externally.

4.1 Operations on Externally Unique Pointers

We aim to make our system as clean and simple as possible to make it play well with other constructs and for the constructs themselves to remain orthogonal and combinable. To this end, we allow only two operations on unique pointers, *movement* and *borrowing*.

The movement operation simply moves a unique from one variable or field to another, nullifying the source. The borrowing operation converts the unique into a normal pointer for a well-defined scope. To invoke methods or access fields of a unique object, the object must first be borrowed.

4.1.1 Movement

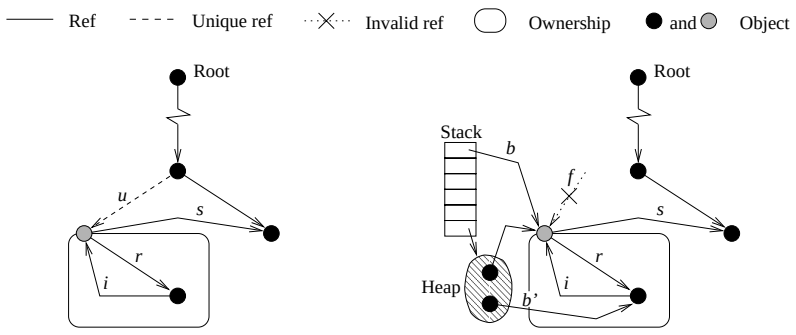
For simplicity, we chose to implement movement with destructive reads [9]. To make the syntax clearer, and also to simplify the formal account of our proposal, we chose to make destructive reads explicit. Consequently, when destructively reading an l-value, it must be suffixed with --. Thus, the syntax of movement becomes:

```
lval--;          // make contents of lval a free value and nullify lval
x = lval--;      // move contents of lval into x and nullify lval
return lval--;   // return contents of lval and nullify lval
```

The use of destructive reads (or equivalent mechanisms, such as alias burying [2]) protects the uniqueness invariant. As reading a unique variable has the side-effect of updating the variable with *null*, unique values are effectively *moved* instead of aliased when assigned from.

4.1.2 Borrowing

Many proposed systems [9,1,18,11,6] use borrowing to tackle the "slipperiness" [19] of unique pointers. A unique variable may be passed as a *borrowed argument* to a method. Borrowed arguments may only be used to invoke anonymous methods (methods that do not create aliases to their receiver), can only be passed to another method as a borrowed argument, and may not be returned nor stored on the heap. Thus, all aliases to a borrowed reference created by a method will be destroyed when the method exits. Thus, the borrowed



External Uniqueness Borrowing

Figure 4. Mediating between external uniqueness and borrowing — b is the original borrowed reference. b, b' are only valid during the borrowing. (Stack grows downwards.)

value is once again unique when the borrowing ceases. This alleviates some of the pain of programming with unique values as uniques passed as borrowed arguments or borrowed receivers can be automatically reinstated when the borrowing method exits.

In our proposal, we introduce an additional borrowing statement which temporarily moves a unique value into a non-unique value confined to a specific scope. The syntax of the borrowing statement looks thus:

borrow *lval* **as** `temp:var { ... }`

where *lval* is a uniquely typed, l-value and `temp` and `var` are the names of the temporary owner and variable introduced for the duration of the block.

When the borrowing block is evaluated, the content of *lval* is destructively read, moved into `var` and given a new owner `temp`. As we shall see later, this restricts the scope of the borrowed reference to remain inside the block.

In existing systems using borrowing, borrowed pointers are an additional kind of pointers that may not be stored on the heap nor returned from methods. These are very arbitrary restrictions that are likely to make programming with borrowing more complex and less flexible. Provided all borrowed references are destroyed before the method exits, it is safe to allow borrowed arguments to be stored on the heap. However, the type systems of previous proposals have simply not been strong enough to express such constraints. In previous work [8], we showed how our vehicle language, Joline, overcomes these limitations by virtue of ownership types.

Borrowing can be implemented to maintain uniqueness or not. Our system can support both, but for simplicity, we only consider the second case here and

in our formalism. For borrowing that maintains uniqueness for the borrowed pointers, see the author's dissertation [14].

4.1.3 Movement Bounds

As we discussed in Section 3, subsumption may hide owners in a type. This leads to problems as information that controls how a unique may move is lost and might lead to references to representation leaking out of its aggregate. To overcome this, we attach a movement bound on unique references. In our formalism, these are written $\texttt{unique}_p$ where p is the outermost owner to which the unique may move. Previous work [16] include a discussion of how to put movement bounds under programmer control and a program elaboration scheme that automatically decorates all uniques in a program with default movement bounds. Movement bounds default to `owner` or, in contexts without `owner`, to `world`.

We now move on to describe Joline, a programming language with external uniqueness built on ownership types. Joline sports *owner-polymorphic methods* and method-local owners called *scoped-regions*. These are useful not only in an ownership types setting, but can be used to simulate borrowing without prohibiting borrowed references to be stored on the heap. For brevity, refrain from discuss these here and instead refer the reader to the original proposal [8] or the author's dissertation [14].

5 Joline

Joline is a class-based object-oriented Java-like programming language with deep encapsulation due to ownership types. It is based on Clarke and Drossopoulou's Joe_1 [17], but lacks some of its constructs, such as the effect annotations which are not needed for our purposes here, and add others, most notably ownership nesting in the class headers. Joline supports inheritance, overriding, subsumption and dynamic binding.

This section presents the formal semantics of the Joline programming language, its static type system, its dynamic semantics, soundness proof and proofs of its structural invariants that describe the effects of ownership types and external uniqueness in terms of what parts of the system may alias what parts.

The major obstacle of formalising Joline is no doubt the external uniqueness. As uniqueness is built on ownership, and owners are captured in types, move-

c	$\in$	**ClassName**	f	$\in$	**FieldName**	md	$\in$ **MethodName**
x, y	$\in$	**TermVar**	α, β	$\in$	**OwnerVar**	R	$\in$ $\{\prec^*, \succ^*\}$

P	::=	$class_{i\in 1..n}\ s\ e$	PROGRAM
$class$	::=	`class` $c\langle\alpha_i\,\mathsf{R}_i\ p_{i\in 1..m}\rangle$ `extends` $c'\langle p'_{i'\in 1..n}\rangle$ `{` $fd_{j\in 1..r}\ meth_{k\in 1..s}$ `}`	CLASS
fd	::=	$t\ f = e$`;`	FIELD
$meth$	::=	$\langle\alpha_i\,\mathsf{R}_i\ p_{i\in 1..m}\rangle\ t\ md(t_i\ x_{i\in 1..n})$ `{` s `return` e `}`	METHOD
$lval$	::=		L-VALUE
		x	VARIABLE
		$e.f$	FIELD
e	::=		EXPRESSION
		`this`	THIS
		$lval$	L-VALUE
		$lval$`--`	DESTRUCTIVE READ
		`new` t	NEW
		`null`	NULL
		$(p)\ e$	LOSE UNIQUENESS
		$e.md\langle p_{j\in 1..m}\rangle(e_{i\in 1..n})$	METHOD CALL
s	::=		STATEMENT
		`skip;`	SKIP
		$t\ x = e$`;`	VARIABLE DECLARATION
		e`;`	EXPRESSION
		$lval = e$`;`	UPDATE OF LVALUE
		$s_1;\ s_2$	SEQUENCE
		`if` (e) `{` s_1 `}` `else` `{` s_2 `}`	IF-STATEMENT
		(α) `{` s `}`	SCOPED REGION
		`{` s `}`	BLOCK
		`borrow` $lval\ t$ `as` $\alpha : x$ `{` s `}`	BORROWING
p, q	::=		OWNERS
		`this`	THIS
		α	OWNER PARAMETER
		`world`	WORLD
		`owner`	OWNER
		`unique`	UNIQUE
		$\texttt{unique}_p$	UNIQUE IN ELABORATED LANGUAGE
t	::=		TYPE
		$p\texttt{:}c\langle p_{i\in 1..n}\rangle$	

Table 1
Syntax of Joline

ment and borrowing causes types of unique objects to change. Our way of dealing with this is to use a slightly different form of store type, where types of uniques are hidden to everything except the unique objects themselves. Thus, with a few exceptions, judgements never depend on non-invariant parts of type information. This practise avoids a lot of trouble and is explained in more detail in the dynamic semantics.

5.1 Joline, Statically

We now describe the syntax and static semantics of Joline. The dynamic semantics follow in Section 5.3.

The syntax of Joline is displayed in Table 1. It is basically a subset of Java extended with ownership types and uniqueness and should be familiar to anyone with some experience of Java. A program is a collection of classes followed by a statement and a resulting expression that are the equivalent of Java's main method. We could have followed Java's example and use a static main method etc., but chose this way out for simplicity.

As we showed in Section 3, classes are parameterised with owner parameters. Each owner parameter (except the implicit, first parameter, `owner`) must be related to either `owner` or some previously declared parameter of the same class. Classes contain fields and methods. Fields must be initialised. Object creation requires the owner parameters specified in the class header to be bound to actual owners.

Before presenting the static semantics of Joline, we first introduce owner substitutions, a few helper functions as well as field and method lookup. We then present the type rules, beginning with the rules for well-formed environments, owner orderings, classes, methods and programs. We then proceed with types and subtypes. Last, we present the type rules for Joline's statements and expressions.

5.1.1 Owner Substitutions

Substitution is denoted σ, where σ is a map from owner variables to owners.

We write σ^p to mean $\sigma \cup \{\texttt{owner} \mapsto p\}$ and σ_n to mean $\sigma \cup \{\texttt{this} \mapsto n\}$ and σ_n^p for the combination. Applying a substitution to an owner is written $\sigma(p)$. For brevity, we write $\sigma(\alpha \,\mathsf{R}\, p)$ for applying a substitution to a pair of owners related with R. The application is defined thus:

$$
\begin{aligned}
\sigma(p) &= q, \text{ if } p \mapsto q \in \sigma \\
\sigma(p) &= p, \text{ if } p \mapsto q \notin \sigma \\
\sigma(p \,\mathsf{R}\, q) &= \sigma(p) \,\mathsf{R}\, \sigma(q) \text{ if } p \in dom(\sigma) \\
\sigma(p \,\mathsf{R}\, q) &= p \,\mathsf{R}\, q \text{ if } p \notin dom(\sigma)
\end{aligned}
$$

Applying a substitution to a type is written $\sigma(t)$ or $\sigma(t \to t')$ and is defined thus:

$$\sigma(p\!:\!c\langle p_{i\in 1..n}\rangle) = \sigma(p)\!:\!c\langle\sigma(p_i)\ _{i\in 1..n}\rangle$$
$$\sigma(\texttt{unique}_b\!:\!c\langle p_{i\in 1..n}\rangle) = \texttt{unique}_{\sigma(b)}\!:\!c\langle\sigma(p_i)\ _{i\in 1..n}\rangle$$
$$\sigma(t \to t') = \sigma(t) \to \sigma(t')$$

We write $\circ$ for composition of substitution maps:

$$\sigma_1 \circ \sigma_2 = \{p \mapsto \sigma_1(q) \ \mid \ p \mapsto q \in \sigma_2\}$$

As an illustration, if type $p\!:\!List\langle q\rangle$ is formed from the class definition **class** `List<data` **outside** `owner> { ... }` then $p\!:\!List\langle\sigma\rangle$ is the same type where $\sigma = \{\texttt{data} \mapsto q\}$.

5.1.2 Field lookup

For any class c, $\mathcal{F}_c$ is a map from the names of all fields defined in c and all of its superclasses to their corresponding types. For example, given the following class definitions

```
class Super<some outside owner> extends Object {
  owner : Link<some> f1;
}

class Example<data outside owner, other outside data>
                              extends Super<other> {
  this : Link<data> f2;
}
```

we have $\mathcal{F}_{Example} = \{\ f1 \mapsto owner\!:\!Link\langle other\rangle,\ \ f2 \mapsto this\!:\!Link\langle data\rangle\ \}$. Note that the map contains `f1` and that the type of `f1` has been translated using the superclass mapping into using the owner names defined in `Example`, not the names used in `Super`.

Field lookup is formally defined as:

$$\mathcal{F}_c(f) = \begin{cases} \bot, & \text{if } c \equiv \texttt{Object} \\ t, & \text{if } \texttt{class}\ c\ \cdots\ \{\ \cdots\ f\ t\ \cdots\ \} \in P \\ \sigma(\mathcal{F}_{c'}(f)), & \text{if } \texttt{class}\ c\langle\_\rangle\ \texttt{extends}\ c'\langle\sigma\rangle\ \{\ fd_{1..n}\ \cdots\ \} \in P \wedge \\ & \qquad\qquad f \notin dom(fd_{1..n}) \end{cases}$$

The $\bot$ means that the field is not defined for class c.

The σ on the third line is a map from the superclass' parameters to the parameters used in the subclass. When looking up a field variable for class c on the third line, the types in $\mathcal{F}_{c'}$ use owner names in the class definition of class

c'. Thus we apply the substitution $\sigma(\mathcal{F}_{c'}(f))$ to bind the owner parameters of c' to the owners in c. This gives us a type for f in c.

As is standard, we write $\_$ for an uninteresting variable.

5.1.3 Method lookup

Method lookup is similar to the field lookup mechanism described above. For any class c, $\mathcal{M}_c$ is a map from all names of all methods defined in c and all of its superclasses to a tuple containing the argument types and return type of the method.

Just as for field lookup, superclass owner parameters are bound to subclass owners using σ-substitution. Thus, when looking up a method in class c, the types returned will use owner names defined in c.

Method lookup is formalised as:

$$\mathcal{M}_c(md) = \begin{cases} \bot, & \text{if } c \equiv \texttt{Object} \\ (\Omega, t_{i\in 1..n} \to t'), & \text{if } \texttt{class}\, c \cdots \{ \cdots \langle \Omega \rangle\ t'\ md(t_i\ x_{i\in 1..n}) \cdots \} \in P \\ \sigma(\mathcal{M}_{c'}(md)), & \text{if } \texttt{class}\ c\langle\_\rangle\ \texttt{extends}\ c'\langle\sigma\rangle \cdots \in P \end{cases}$$

where $\Omega = \alpha_j\, \mathsf{R}_j\, p_{j\in 1..m}$. (For the last case, we implicitly assume that md is not in the body of class c.)

5.2 Well-formedness Rules

In this section, we present the static semantics of Joline. An overview of the different judgements used is found in Table 2. In the type rules, P is the complete program and a global constant for all rules except $\vdash P$. This reduces the syntactic overhead necessary to thread P from the top level to all the rules where it is required.

5.2.1 Static Type Environment

The type environment E records the types of free term variables and the nesting relation on owner parameters:

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

Above, ϵ is the empty environment, $x :: t$ is a variable to type binding and $\alpha \succ^* p$ means that owner parameter α is outside owner p. Conversely, $\alpha \prec^*$

$E \vdash \diamond$	Good environment
$E \vdash p$	Good owner
$E \vdash p \,\mathsf{R}\, q$	Owner p is R-related to q ($\mathsf{R} \in \{\prec^*, \succ^*\}$)
$E \vdash t$	Good type
$E \vdash t \leq t'$	Type t is a subtype of type t'
$E \vdash v :: t$	Value v has type t
$E \vdash e :: t$	Expression e has type t
$E \vdash \mathit{lval} :: t$ **ref**	l-value *lval* has type t
$E \vdash s; E'$	Statement s is well-formed and extends E to E'
$E \vdash \mathit{meth}$	Good method
$\vdash \mathit{Class}$	Good class
$\vdash P$	Good program

Table 2
Judgements used in the static semantics.

$\bigsqcup \{p_{i \in 1..n}\}$ means α is inside all $p \in \{p_{i \in 1..n}\}$.

5.2.2 Good environment

$$\frac{}{\epsilon \vdash \diamond}\ (\textsc{env-}\epsilon) \qquad \frac{E \vdash t \quad x \notin dom(E)}{E, x :: t \vdash \diamond}\ (\textsc{env-}x) \qquad \frac{E \vdash p \quad \alpha \notin dom(E)}{E, \alpha \succ^* p \vdash \diamond}\ (\textsc{env-}\alpha \succ^*)$$

$$\frac{E \vdash p_{i \in 1..n} \quad \alpha \notin dom(E)}{E, \alpha \prec^* \bigsqcup \{p_{i \in 1..n}\} \vdash \diamond}\ (\textsc{env-}\alpha \prec^*)$$

The rules for good environment are straightforward. (ENV-ϵ) states that the empty environment, ϵ, is well-formed. (ENV-x) states that adding a variable name to type binding, $x :: t$ to a good environment E produces another good environment provided x is not already bound to a type in E and t is a well-formed under E. The rules (ENV-$\succ^*$) and (ENV-$\prec^*$) deal with inside and outside orderings of owners—(ENV-$\succ^*$) states that adding a $\alpha \succ^* p$ ordering of two owners to a good environment E produces a good environment if p is a good owner under E and α is not in E. The (ENV-$\prec^*$) rule states the same, but for the $\prec^*$ relation and several owners.

5.2.3 Good owner

$$\frac{\alpha \,\mathsf{R}\, \_ \in E}{E \vdash \alpha}\ (\textsc{owner-var}) \qquad \frac{\texttt{this} : t \in E}{E \vdash \texttt{this}}\ (\textsc{owner-this}) \qquad \frac{E \vdash \diamond}{E \vdash \texttt{world}}\ (\textsc{owner-world})$$

The rules for good owners state that an owner is well-formed if it is defined in the static environment. Also, if present in the environment, the special

variable `this` is also a good owner. The owner `world` is globally defined, and thus always valid.

5.2.4 Owner Orderings

$$\frac{\alpha \prec^* p \in E}{E \vdash \alpha \prec^* p}\ (\textsc{in-env1}) \qquad \frac{\alpha \succ^* p \in E}{E \vdash p \prec^* \alpha}\ (\textsc{in-env2}) \qquad \frac{E \vdash p}{E \vdash p \prec^* \texttt{world}}\ (\textsc{in-world})$$

$$\frac{\texttt{this} :: t \in E}{E \vdash \texttt{this} \prec^* \texttt{owner}}\ (\textsc{in-this}) \qquad \frac{E \vdash p}{E \vdash p \prec^* p}\ (\textsc{in-refl}) \qquad \frac{E \vdash p \prec^* q \quad E \vdash q \prec^* q'}{E \vdash p \prec^* q'}\ (\textsc{in-trans})$$

The inside and outside relations are derived from the owner orderings in E. The relations are transitive and reflexive and each others' inverses. Nesting relations form a tree (since an owner can only be ordered inside one owner by (IN-ENV1)). From (IN-WORLD), we see that all owners are inside `world`. Importantly, if `this` is a valid owner, it is always ordered inside `owner`, which is the owner of the object denoted by `this`.

5.2.5 Program and Class

$$\frac{\vdash class_{i\in 1..n} \quad \vdash s\ ; E \quad E \vdash e :: t}{\vdash class_{i\in 1..n}\ s\texttt{; return}\ e :: t}\ (\textsc{program}) \qquad \frac{}{\vdash \texttt{class Object \{ \}}}\ (\textsc{root-class})$$

By (PROGRAM), a program is well-formed if all the classes it defines are well-formed and all statements in the body of main, s`;return` e, are well-formed. By (ROOT-CLASS), the empty root class `Object` is always well-formed.

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

The rule for well-formed class, (CLASS), is a little more complex. First, owner must be inside all owner parameters of the class. Secondly, the supertype to the class must be valid (remember σ is a map from owner names used in the class header of c' to the owner names $\alpha_{i\in 1..m}$ used in c). Shadowing fields is not permitted as the names of the fields declared in c must not be in set of fields declared by any superclass to c. Any expression initialising a field must

be valid under the class' environment E, constructed from the owner class' header, adding `owner` and the `this` variable. Finally, all methods declared in the class must be well-formed under E and, notably, for all overridden methods (a method with the same name as one defined in any superclass to c), the types must be invariant modulo σ-substitution which binds the names of the owner parameters of c' into the corresponding names in c.

5.2.6 Good method

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

A method is well-formed under environment E if the statements and return expression of its body are well-formed with respect to E *extended with the owner parameters* declared in the method header and the regular parameter variables. This considers the well-formedness of the arguments as $E'' \vdash s; E'$ requires $E'' \vdash \diamond$.

5.2.7 Types

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

A type is well-formed whenever the substituted owner arguments satisfy the ordering on parameters specified in the class header.

5.2.8 Subtyping

Subtyping in Joline must care to preserve the owner to protect the containment invariant. A supertype is allowed to "forget" owners, which is governed by (CLASS). The subtyping rule states that the owner must remain the same.

$$\frac{E \vdash p{:}c\langle \sigma^p\rangle \quad \texttt{class}\ c\langle\ldots\rangle\ \texttt{extends}\ c'\langle p'_{i\in 1..n}\rangle \cdots \in P}{E \vdash p{:}c\langle\sigma\rangle \leq p{:}c'\langle\sigma(p'_{i\in 1..n})\rangle}\ (\textsc{sub-class})$$

Subtyping is derived from subclassing, modulo names of the owner parameters. As this corresponds to the composition of two order-preserving functions, it is order-preserving. This is required to preserve deep ownership, see Clarke's

dissertation [5]. In particular, subtyping preserves the owner that is fixed for life. Letting the owner vary, as in Cyclone [20], would be unsound in our system, as observed by Clarke and Drossopoulou [17].

$$\frac{E \vdash t}{E \vdash t \leq t}\ (\textsc{sub-refl}) \qquad \frac{E \vdash t \leq t' \quad E \vdash t' \leq t''}{E \vdash t \leq t''}\ (\textsc{sub-trans})$$

As expected, the subtype relation is reflexive and transitive.

5.2.9 Statements

$$\frac{x \notin dom(E) \quad E \vdash e :: t}{E \vdash t\ x\ = e\texttt{;}\ ; E, x :: t}\ (\textsc{stat-local})$$

(STAT-LOCAL) describes the conditions for variable declaration. The variable name must not be in use in the same environment and the initial expression must have the same type as the declared type of the variable (modulo subsumption). The resulting environment is extended with a binding from the variable name to its type to record the type information of the declared variable.

$$\frac{E \vdash \diamond}{E \vdash \texttt{skip;}\ ; E}\ (\textsc{stat-skip}) \qquad \frac{E \vdash e :: t}{E \vdash e\texttt{;}\ ; E}\ (\textsc{stat-expr}) \qquad \frac{E \vdash lval : t\ \textbf{ref} \quad E \vdash e :: t}{E \vdash lval = e\texttt{;}\ ; E}\ (\textsc{stat-update})$$

From (STAT-SKIP), `skip` is a valid statement under any valid environment. From (STAT-EXPR), a well-formed expression can be treated as a statement. The rule (STAT-UPDATE) simply enforces that updates can be performed to l-values only if the types match, modulo subtyping of e. Recall that *lval* is either x or $x.f$.

$$\frac{E \vdash s_1\ ; E'' \quad E'' \vdash s_2\ ; E'}{E \vdash s_1\ s_2\ ; E'}\ (\textsc{stat-sequence})$$

From (STAT-SEQUENCE), statements can be sequenced in the standard fashion.

$$\frac{E, \alpha \prec^* \bigsqcup \{p_{i \in 1..n}\} \vdash s\ ; E' \quad \{p_{i \in 1..n}\} \subseteq \mathsf{owners}(E)}{E \vdash (\alpha)\ \texttt{\{}\ s\ \texttt{\}}\ ; E}\ (\textsc{stat-scoped-region})$$

The rule (STAT-SCOPED-REGION) introduces a new owner variable that corresponds to a block and is only defined for the scope of the block. The bounds

$\{p_{i\in 1..n}\}$, though unspecified in code, determine which objects may be accessed by objects created in this scope. Statically, the owner is inside (a subset of) all owners in the lexical scope of the block.

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

The rule (STAT-BORROW) states that any uniquely type l-value may be borrowed. This is achieved by introducing a new owner variable which is restricted to the scope of the borrowing block analogous to a scoped region, to act as the owner of the temporary non-unique reference to the borrowed value. To ensure that this reference or other references to the borrowed value do not escape this scope, we require that this owner is inside the unique type's movement bound. The remainder of the type *must* correspond exactly to the type of the l-value, so that the borrowed variable can be reinstated with a correctly typed value when the borrowing ends.

To simplify the formalism, we require that the unique is first moved into a local variable on the top frame of the stack. This does not affect the expressiveness of the language, as borrowing from any variable or field can be simulated by manually moving the unique into the appropriate variable and then manually reinstate it.

5.2.10 l-values

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

$$\frac{E \vdash \mathit{lval} :: t\ \textbf{ref}}{E \vdash \mathit{lval} :: t}\ (\textsc{expr-lval})$$

The rules above give the types of l-values, which are variables (other than `this`) and fields. Their type of a variable is its recorded type in the environment, and the type of a field is the declared type in its class modulo substitution of the owner parameters in the object where the field is accessed. l-values may be updated or read.

The **ref** annotations on types in (LVAL-VAR) and (LVAL-FIELD) prevent them from taking part in subsumption. The rule (EXPR-LVAL) can be used to treat an l-value as an expression.

The helper function owners is defined for types and static type environments. When applied to a type, it returns a set of all owners used to form that type; when applied to an environment, it returns a set of all owners defined in that environment:

$$\mathsf{owners}(p_1 : c\langle p_{i\in 2..n}\rangle) = \{p_1, \ldots, p_n\}$$

$$\begin{aligned}
\mathsf{owners}(\epsilon) &= \texttt{world} \\
\mathsf{owners}(E, x :: t) &= \mathsf{owners}(E) \\
\mathsf{owners}(E, \alpha \succ^* p) &= \mathsf{owners}(E) \cup \{\alpha\} \\
\mathsf{owners}(E, \alpha \prec^* p) &= \mathsf{owners}(E) \cup \{\alpha\}
\end{aligned}$$

The condition $\texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this}$, which was called the *static visibility test* in the original ownership types system [4], ensures that types that contain `this` in them, that is types of representation objects, can only be accessed internally to the object. It amounts to saying that fields (and methods) which yield, return or require that representation objects are private. This is not essential; we could have used *dynamic aliasing* as in Joe$_1$ [17], but the resulting type system would have been too complex to present our ideas.

5.2.11 Expressions

$$\frac{\text{(EXPR-THIS)}\quad \texttt{this} :: t \in E}{E \vdash \texttt{this} :: t} \qquad \frac{\text{(EXPR-NULL)}\quad E \vdash t}{E \vdash \mathit{null} :: t} \qquad \frac{\text{(EXPR-SUBSUMPTION)}\quad E \vdash e :: t \quad E \vdash t \leq t'}{E \vdash e :: t'}$$

From (EXPR-THIS), `this` has its declared type. From (EXPR-NULL), *null* can have any well-formed type. By (EXPR-SUBSUMPTION), an expression of type t can be said to be of any type t' such that t' is a supertype of t.

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

The rule for method performs a static visibility test, just as for field access, which restricts expressions containing `this` in their type (as declared in the class) to being used only internally, that is, on `this`. The owners of the target type forms a substitution to translate the owners in the method's argument's types and return types into the corresponding types using the owners in scope. 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 in terms of the owners in scope. The additional substitution, σ', translates the names of owner parameters used internally in the method to the actual owners at the call-site. The helper function $\mathcal{M}_c$ for

looking up parameters and method bodies is extended in a straightforward fashion to also include owner parameter lists.

As pointed out by Clarke [5], owner argument passing could be replaced by an inference mechanism that infers the σ' binding of owners to the parameters of the method, just as in GJ [21] or Scala [22]. That would introduce unnecessary complexity for our purposes here so we chose this way out for simplicity. A possible inference mechanism would look at the owner parameters of the method header and where these are used in the parameter types. It would then match the types of the arguments with the types of the parameters to derive the mapping.

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

With unique values in the system, it is not possible to treat all l-values directly as l-values as subsumption would allow them to be viewed as being of the corresponding non-unique type. This would allow method calls and field accesses on unique references, which would break external uniqueness. 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) an 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. Destructive reads can also safely apply to non-unique l-values.

$$\frac{E \vdash p{:}c\langle\sigma\rangle}{E \vdash \mathtt{new}\ p{:}c\langle\sigma\rangle :: \mathtt{unique}_p{:}c\langle\sigma\rangle}\ (\textsc{expr-new})$$

By (expr-new), instantiating a class creates an externally unique object and the owner of the non-unique type becomes the movement bound.

5.2.11.1 Uniqueness and Subsumption To simplify the formal account, we chose to make loss of uniqueness explicit using a movement operation. Had we not chosen this approach, the rules for subtyping and moving for unique and non-unique types would have looked like this:

$$\frac{E \vdash \mathtt{unique}_p{:}c\langle\sigma\rangle}{E \vdash \mathtt{unique}_p{:}c\langle\sigma\rangle \le p{:}c\langle\sigma\rangle}\ (\textsc{sub-unique}) \qquad \frac{E \vdash q \prec^* p}{E \vdash \mathtt{unique}_p{:}c\langle\sigma\rangle \le \mathtt{unique}_q{:}c\langle\sigma\rangle}\ (\textsc{sub-move})$$

Such rules would, however, allow the implicit conversion of objects from unique to non-unique type. This would have to be taken into consideration at many

points in the formalism, complicating it further. Rather, we require conversion to be explicit:

$$\frac{E \vdash e :: \texttt{unique}_b : c\langle\sigma\rangle \quad E \vdash p \prec^* b}{E \vdash (p)\ e :: p : c\langle\sigma\rangle} \text{(EXPR-LOSE-UNIQUENESS)}$$

The "owner-cast" expression moves the contents of a unique into a subheap of some object or block (whatever the p owner corresponds to). This is well-formed if the expression has a unique type and if the movement bound of the type is outside the target owner.

5.3 Joline, Dynamically

Heaps in Joline are nested to model the ownership nesting of deep ownership types. The store consists of a stack of frames, each frame corresponding to an executing method. The bottom frame contains the heap nested inside world. Nested inside this heap are all subheaps of all objects nested inside world.

Joline's dynamic semantics are formulated as a big-step operational semantics. This section presents the dynamic semantics of Joline, explaining as we go along.

5.3.0.2 Syntax definitions Meta-variables n, m, p, q range over owners and ids of objects or blocks. As in the static semantics, x is a variable, α is a static owner name and t is a type, a class with its owner parameters bound.

The syntax for heaps, stacks and values are shown in Figure 5. We write $S \bullet$ to mean $S \bullet \mathsf{nil}$. Stacks and store types have parallel structure. Stacks, S, consist of ordered frames, F. Frames consist of variable mappings $x \mapsto v$, owner mappings $\alpha \mapsto n$, regions $\mathsf{R}_n[H; F]$ and borrowing blocks $\mathsf{B}^b_n[H; F]$ with id n, bound b and nested frame and subheap. The bound on the borrowing block controls what objects may be stored in the block to make sure reinstatement does not break containment. Figure 5 describes the syntax for stacks and stores. The syntactic category V denotes zero or more fields, $f \mapsto v$. The region construct models the region `world`, with a nested subheap. The symbol H denotes zero or more objects, $n \mapsto c^\sigma[V; H]$, in a nested subheap. The $\oplus$ operator pushes its right-hand side to the innermost compartment of the left-hand side, just as in the store-type. Thus, $\mathsf{R}_n[H; \mathsf{nil}] \oplus F$ is equivalent to $\mathsf{R}_n[H; F]$. Values are *null*, $\uparrow n$, which denotes pointer (we will sometimes omit the $\uparrow$) and $\mathsf{U}_n[v; H]$, which denotes a unique value with id n and subheap H where v is the pointer into v obtained by dereferencing the unique.

$$
\begin{array}{rcll}
S & ::= & F \mid S \bullet F & \textsc{Stack} \\
F & ::= & \mathsf{nil} \mid x \mapsto v, F \mid \alpha \mapsto n, F \mid \mathsf{R}_n[H; F] \mid \mathsf{B}^b_n[H; F] & \textsc{Frame} \\
H & ::= & \mathsf{nil} \mid n \mapsto o, H & \textsc{Subheap} \\
V & ::= & \mathsf{nil} \mid f \mapsto v, V & \textsc{Field} \\
o & ::= & c^{\sigma}[V; H] & \textsc{Object} \\
v & ::= & \mathit{null} \mid \uparrow n \mid \mathsf{U}_n[v; H] & \textsc{Value}
\end{array}
$$

Figure 5. Syntax for stacks, heaps and values.

$$
\begin{array}{rcll}
\Gamma & ::= & \mathsf{nil} \mid n :: T[\,\Gamma\,], \Gamma' \mid x :: t, \Gamma \mid \alpha \mapsto n, \Gamma \mid \Gamma \bullet \Gamma' & \textsc{Store type} \\
\mathrm{T} & ::= & c\langle\sigma\rangle \mid \mathfrak{U} \mid \mathfrak{R} \mid \mathfrak{B} & \textsc{Owner-less type} \\
\Gamma_{\langle\,\rangle} & ::= & \langle\,\rangle \mid n :: T[\,\Gamma_{\langle\,\rangle}\,], \Gamma' \mid n :: T[\,\Gamma\,], \Gamma'_{\langle\,\rangle} \mid & \textsc{Holed store type} \\
 & & x :: t, \Gamma_{\langle\,\rangle} \mid \alpha \mapsto n, \Gamma_{\langle\,\rangle} \mid \Gamma_{\langle\,\rangle} \bullet \Gamma' \mid \Gamma \bullet \Gamma'_{\langle\,\rangle}
\end{array}
$$

Figure 6. Store type. We write $\Gamma\bullet$ to mean $\Gamma \bullet \mathsf{nil}$.

We sometimes refer to a stack frame as a *generation*. Owners introduced by borrowing blocks are tied to a stack frame, and thus, objects that have such owners can never become visible to earlier stack frames. Thus, each stack frame really becomes a root in the object graph. When a stack frame is destroyed, all the objects in the heap belonging to the stack frame can this be destroyed. This is somewhat similar to generational garbage collection [23].

We now describe the rules for well-typed configurations roughly in the same order as the syntactic definitions in Figure 5. An overview of the judgements is found in Table 3.

Additional syntax for the store type is shown in Figure 6.

5.4 Store Type

The store type is consists of stack frames separated by $\bullet$, where each stack-frame contains owner bindings and variable typings. The store type and the stack have parallel structures. Objects, variables and owners on a frame are ordered by $\oplus$ which is an order-preserving concatenation operator.

As is standard, we assume that variable names, owner names and object ids are unique.

Note that the owner of an object is not encoded in its type but is implicit in the nesting. Thus, an object of class c with owner parameters σ nested inside some object m will have the "owner-less type" $c\langle\sigma\rangle$ in the store type. However, we can derive its "complete type" $m{:}c\langle\sigma\rangle$ from the nesting. The types $\mathfrak{R}$, $\mathfrak{B}$, and $\mathfrak{U}$ denote a region, a borrowing block and a unique type respectively. They are all containers for a store-type corresponding to a nested subheap.

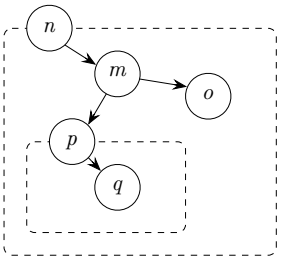


Figure 7. An object graph. Its corresponding store-type, for some owner-less types $c_i\langle\sigma_i\rangle$ for $i = 1..5$ is $n :: c_1\langle\sigma_1\rangle[\, m :: c_2\langle\sigma_2\rangle, o :: c_3\langle\sigma_3\rangle, p :: c_4\langle\sigma_4\rangle[\, q :: c_5\langle\sigma_5\rangle\,]\,]$. As is visible from the picture, the owner of objects m, o and p is n. This is mirrored by the nesting structure of store-type.

Figure 7 shows a sample object graph and its corresponding store-type.

The syntactic category $\Gamma_{\langle\,\rangle}$ describes a store type with a hole. The syntax $\Gamma\langle\cdots\rangle_m$ means the stack Γ extended by $\cdots$ inside the subheap of some object m. We sometimes write $\Gamma = \Gamma'\langle H\rangle_m$ to mean that Γ can be factored into some stack Γ' with a *hole* in m and H, which are part of the contents of m in Γ, or, equivalent, that stack Γ' can be extended by H inside m to yield stack Γ.

We now proceed by defining a few helper functions.

5.4.1 Definition of defs, owners and vars for Γ

We define the function $\mathsf{defs}(\Gamma)$ to be the set of all identities of objects typed in Γ and names of variables typed in Γ. We define the function $\mathsf{owners}(\Gamma)$ to be the set of all owner variables mapping to identities of objects on the topmost stack frame Γ. We define the function $\mathsf{vars}(\Gamma)$ to be the set of all variable names mapping on the topmost stack frame Γ.

5.4.2 Definition of $\oplus$ *for* Γ

As previously explained, $\oplus$ is an operator that pushes a Γ to the top of another Γ. For example, $(x :: t, \alpha \mapsto n) \oplus z :: t' = x :: t, \alpha \mapsto n, z :: t'$ and $(\Gamma \bullet x :: t, n :: \mathfrak{R}[\alpha \mapsto n]) \oplus z :: t' = \Gamma \bullet x :: t, n :: \mathfrak{R}[\alpha \mapsto n, z :: t']$.

5.4.3 Rules for Well-formed Store Type

$$\frac{}{\mathsf{nil} \vdash \diamond}\ (\textsc{store-type-empty}) \qquad \frac{\Gamma \vdash \diamond \quad p \in \mathsf{defs}(\Gamma)}{\Gamma \vdash p}\ (\textsc{good-owner})$$

By (STORE-TYPE-EMPTY), the empty store type is well-formed. By (GOOD-

OWNER), p is a good owner under Γ if it is in the set $\mathsf{defs}(\Gamma)$, the set of all ids of all objects, blocks and variables in Γ.

$$\frac{\Gamma \vdash n \quad \alpha \notin \mathsf{owners}(\Gamma)}{\Gamma \oplus \alpha \mapsto n \vdash \diamond}\ (\textsc{store-type-owner}) \qquad \frac{\Gamma \vdash t \quad x \notin \mathsf{vars}(\Gamma)}{\Gamma \oplus x :: t \vdash \diamond}\ (\textsc{store-type-var})$$

By (STORE-TYPE-OWNER), a static-name to actual owner binding $\alpha \mapsto n$ may be added to Γ, if Γ is well-formed, n is a good owner and α is not in the set $\mathsf{owners}(\Gamma)$, the set of static owner names already in Γ.

By (STORE-TYPE-VAR), a variable name to type binding $x :: t$ may be added to Γ if Γ is well-formed, t is good type under Γ and x is not in the set $\mathsf{vars}(\Gamma)$, that is the set of variable names used in Γ.

$$\frac{\Gamma \vdash \diamond \quad n \notin \mathsf{defs}(\Gamma) \quad T \in \{\mathfrak{R}, \mathfrak{B}\}}{\Gamma \oplus n :: T \vdash \diamond}\ (\textsc{store-type-region}) \qquad \frac{\Gamma \vdash m : c\langle\sigma\rangle \quad n \notin \mathsf{defs}(\Gamma)}{\Gamma\langle n :: c\langle\sigma\rangle\rangle_m \vdash \diamond}\ (\textsc{store-type-object})$$

By (STORE-TYPE-REGION), a region without a nested subheap can be pushed onto Γ if Γ is well-formed and n is not in use in Γ.

$$\frac{\Gamma \vdash m \quad n \notin \mathsf{defs}(\Gamma)}{\Gamma\langle n :: \mathfrak{U}\rangle_m \vdash \diamond}\ (\textsc{store-type-unique}) \qquad \frac{\Gamma \vdash m \quad n \notin \mathsf{defs}(\Gamma)}{\Gamma\langle n :: \mathfrak{B}\rangle_m \vdash \diamond}\ (\textsc{store-type-borrow})$$

By (STORE-TYPE-UNIQUE) and (STORE-TYPE-BORROW), a uniqueness wrapper or borrowing block n in the subheap of some object (or unique, region or borrowing block) m is well-formed if m is a good owner (that is, m is defined in Γ) and n is not used in Γ. Borrowing blocks can also be pushed onto the store-type, much like regions.

By (STORE-TYPE-OBJECT), an object n of class c with owner parameters σ can be added to a subheap of some object (or region or borrowing block) m in Γ, if the type $m : c\langle\sigma\rangle$ (m must be the owner of n's type as n is nested directly inside m) is well-formed under Γ, and n is not in use in Γ.

$$\frac{\Gamma \vdash \uparrow n :: \sigma^p(t)}{\Gamma \bullet \sigma^p_n \oplus \texttt{this} :: t \oplus \mathit{this} \mapsto n \vdash \diamond}\ (\textsc{store-type-generation})$$

The (STORE-TYPE-GENERATION) rule governs the well-formedness of generations in Γ. A new generation corresponds to a stack frame created by method invocation on some reference $\uparrow n$. It contains the owners of type of $\uparrow n$, the static

type of `this`, and a variable to $\uparrow n$ binding to make the receiver accessible on the frame.

5.4.4 Owner Orderings

Owner orderings are crucial to keeping the strong containment invariant of ownership types. Naturally, we can derive some of the orderings directly from the nesting of owners in Γ. However, in presence of generations this becomes a little more complicated.

$$\frac{\Gamma \vdash p \prec^* q}{\Gamma \vdash q \succ^* p}\ (\textsc{in-outside}) \qquad \frac{\Gamma \vdash p}{\Gamma \vdash p \prec^* p}\ (\textsc{in-refl})$$

Trivially, by (IN-OUTSIDE), outside and inside are inverse relations; if owner p is inside q then q is outside p. By (IN-REFL), the inside relation is reflexive.

$$\frac{\Gamma \vdash \diamond \quad \Gamma = \Gamma'\langle\Gamma''\rangle_q \quad p \in \mathsf{defs}(\Gamma'')}{\Gamma \vdash p \prec^* q}\ (\textsc{in-owner}) \qquad \frac{\Gamma \vdash q \quad \Gamma \bullet \Gamma' \vdash p \quad p \in \mathsf{defs}(\Gamma')}{\Gamma \bullet \Gamma' \vdash p \prec^* q}\ (\textsc{in-generation})$$

By (IN-OWNER), every owner p nested inside some owner q is ordered inside q. Trivially, the owners corresponding to objects in a subheap of some object n are inside n. By (IN-GENERATION), any owner in a generation is inside an owner of any previous generation.

$$\frac{\Gamma \vdash \diamond \quad \Gamma(\alpha) = n}{\Gamma \vdash \alpha = n}\ (\textsc{owner-equal}) \qquad \frac{\Gamma \vdash p = p' \quad \Gamma \vdash p' \prec^* q' \quad \Gamma \vdash q' = q}{\Gamma \vdash p \prec^* q}\ (\textsc{in-owner-equal})$$

The last two rules are special and deal with the conversion between static owner names and actual owners without overly complicating the formalism. By (OWNER-EQUAL), If α is a static owner name bound to the actual owner n on the top generation in Γ, then we can consider α and n as equal. The rule (IN-OWNER-EQUAL) simply applies this equality to the inside ordering. For example, if $\Gamma(\alpha) = n$, $\Gamma(\beta) = m$ and $\Gamma \vdash n \prec^* m$, then, by (IN-OWNER-EQUAL), $\Gamma \vdash \alpha \prec^* \beta$.

5.5 Configurations

As we saw in Figure 5, heaps are nested inside stacks. Starting configurations have the form $\langle S \mid s \rangle$, where S is a stack and s is a statement, and resulting

$\Gamma \vdash \langle S \rangle$	Configuration is well-formed under store-type Γ
$\Gamma \vdash \langle S \mid v \rangle :: t$	Configuration is well-formed and v has type t under store-type Γ
$\Gamma \vdash \langle S \mid e \rangle :: t$	Configuration is well-formed and e has type t under store-type Γ
$\Gamma \vdash \langle S \mid s \rangle$	Configuration is well-formed and s produces store-type Γ
$\Gamma \vdash S$	Stack S is well-formed and its contents are typed by Γ
$\Gamma \vdash F \gg \Gamma'$	Frame F is well-formed under Γ, and is parallel with Γ'
$\Gamma; n \vdash H \gg \Gamma'$	Heap H with owner n is well-formed under Γ, and typed by Γ'
$\Gamma \vdash_n v :: t$	Value v has type t in Γ; if v is unique, n is its owner
$\Gamma \vdash t$	Type t is well-formed under Γ
$\Gamma \vdash t = t'$	Types t and t' are equal under Γ
$\Gamma \vdash p$	p is a good owner
$\Gamma \vdash \alpha = p$	Static owner α and dynamic owner p are equal under Γ

Table 3
Table over judgements

configurations are either $\langle S \mid v \rangle$, where v is the resulting value of evaluating an expression, or $\langle S \rangle$, a single stack. The initial configuration where $s\,;e$ is the "main method" of the program is:

$$\langle\ \mathsf{R}_{world}[\mathsf{nil};\mathsf{nil}] \mid s\,;e\ \rangle$$

The rules for well-formed configurations are slightly unorthodox. To the left of the turnstile is the store-typing for the entire store. To the right of the $\gg$ operator is a subset of said store-typing, namely the subset that exactly corresponds to the structure which the judgement is typing. This will be used later to deal with types in the store-type changing due to movement. Commonly, store-types are only extended, never changed, which is why we are forced to resort to less orthodox mechanisms here.

5.5.1 Definition of $\oplus$ for Ss and Fs

The $\oplus$ operator works on stacks and frames just as for Γ's. It pushes a F into the innermost F on the top of the stack. For any stack, this is a unique position.

5.5.2 Configurations

$$\frac{\Gamma \vdash S}{\Gamma \vdash \langle S \rangle}\ (\textsc{config-final}) \qquad \frac{\Gamma \vdash S \quad \Gamma \vdash s; \Gamma'}{\Gamma' \vdash \langle S \mid s \rangle}\ (\textsc{config-stat})$$

By (CONFIG-FINAL), a final configuration is well-formed if its stack is well-formed under the current store type. For (CONFIG-STAT), the statement s must be well-formed under the current store-type and result in a store type possibly extended by the variables declared in s. The configuration's stack must also be well-formed, without the extension to the store type from s.

$$\frac{\Gamma \vdash S \quad \Gamma \vdash e :: t}{\Gamma \vdash \langle S \mid e \rangle :: t}\ (\text{CONFIG-EXPR}) \qquad \frac{\Gamma \vdash S \quad \Gamma \vdash_{\text{free}} v :: t}{\Gamma \vdash \langle S \mid v \rangle :: t}\ (\text{CONFIG-VAL})$$

The rule (CONFIG-EXPR) is straightforward. For configuration with a value compartment, a unique value will not have an owner that corresponds to a variable as such a value will be free. The special owner free is introduced to denote a free value and (CONFIG-VAL) is extended by a free subscript to denote that the resulting value, if unique, must be free.

5.5.3 Stacks

$$\frac{}{\mathsf{nil} \vdash \mathsf{nil}}\ (\text{STACK-EMPTY}) \qquad \frac{\Gamma \vdash S \quad \Gamma \bullet \Gamma' \vdash F \gg \Gamma'}{\Gamma \bullet \Gamma' \vdash S \bullet F}\ (\text{STACK-GEN})$$

By (STACK-EMPTY), the empty stack is typed by the empty store type. By (STACK-GEN), the store type must correspond to the 'sum' of the store type for each generation in the stack. Every generation has access to the store type of all previous generations plus itself. Note that Γ and S are constructed in parallel.

5.5.4 Frames

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{nil} \gg \mathsf{nil}}\ (\text{FRAME-EMPTY}) \qquad \frac{\Gamma \vdash F \gg \Gamma' \quad \Gamma \vdash_x v :: t}{\Gamma \vdash F \oplus x \mapsto v \gg \Gamma' \oplus x :: t}\ (\text{VARIABLES}) \qquad \frac{\Gamma \vdash F \gg \Gamma'}{\Gamma \vdash F \oplus \alpha \mapsto n \gg \Gamma' \oplus \alpha \mapsto n}\ (\text{OWNERS})$$

By (FRAME-EMPTY), an empty frame is valid and is parallel to an empty piece of the store type. The rules (VARIABLES) and (OWNERS) control variable and owner bindings on the stack. The uniqueness of the names x and α respectively are guaranteed by the well-formedness of Γ in both cases. A variable with type t is well-formed if its value also has type t. Note that the structures on the

left and right side of the $\gg$ are parallel.

$$\frac{\Gamma \vdash F \gg \Gamma_1 \qquad \Gamma; n \vdash H \gg \Gamma_2 \qquad \Gamma \vdash F' \gg \Gamma_3}{\Gamma \vdash F \oplus \mathsf{R}_n[H; F'] \gg \Gamma_1 \oplus n :: \mathfrak{R}[\Gamma_2, \Gamma_3]} \text{(FRAME-REGION)}$$

The rule (FRAME-REGION) captures the well-formedness of adding a region to a frame. The two subcompartments of the frame, that is, any objects owner by the frame, n, must be well-formed, as must any nested frames, F'.

$$\frac{\Gamma \vdash F \gg \Gamma_1 \quad \Gamma' = \Gamma\langle n :: \mathfrak{B}[\Gamma_2, \Gamma_3]\rangle_b \quad \Gamma'; n \vdash H \gg \Gamma_2 \quad \Gamma' \vdash F' \gg \Gamma_3}{\Gamma \oplus n :: \mathfrak{B}[\Gamma_2, \Gamma_3] \vdash F \oplus \mathsf{B}^b_n[H; F'] \gg \Gamma_1 \oplus n :: \mathfrak{B}[\Gamma_2, \Gamma_3]} \text{(FRAME-BORROW)}$$

The rule (FRAME-BORROW) captures the well-formedness of adding a borrowing block to a frame. The rule is similar to that for a region, but with one important difference: the subheap of the borrowing block must be well-typed at location b in the store-type. When reinstated, this guarantees that the subheap is well-formed at b, which is the movement bound for unique values, see (VAL-UNIQUE) below.

5.5.5 Heaps and objects

$$\frac{\Gamma \vdash m}{\Gamma; m \vdash \mathsf{nil} \gg \mathsf{nil}} \text{(HEAP-EMPTY)} \qquad \frac{\Gamma; m \vdash n \mapsto o \gg \Gamma' \quad \Gamma; m \vdash H \gg \Gamma''}{\Gamma; m \vdash n \mapsto o, H \gg \Gamma', \Gamma''} \text{(HEAP-OBJECT)}$$

By (HEAP-EMPTY), an empty subheap is valid in m if m is well-formed under the current store type. By (HEAP-OBJECT), a subheap in m is well-formed if its contents is well-formed in m under the current store type. Again, the structures on the left and right side of the $\gg$ are parallel.

$$\frac{\Gamma(n) = m : c\langle\sigma\rangle \quad \Gamma; n \vdash H \gg \Gamma' \quad \Gamma \vdash_n V :: \sigma^m_n(\mathcal{F}_c)}{\Gamma; m \vdash n \mapsto c^\sigma[V; H] \gg n :: c\langle\sigma\rangle[\Gamma']} \text{(OBJECT)}$$

By (OBJECT), an object is well-formed inside m under Γ if it has m as an owner in Γ, its subheap is well-formed inside the object itself, and its fields have good types. Just as in (VARIABLES), (OBJECT) is extended with a subscript on the turnstile to capture the owner of the object.

$$\frac{\Gamma \vdash_{n.f} v :: t \quad \Gamma \vdash_n V \gg \Gamma'}{\Gamma \vdash_n f \mapsto v, V \gg f :: t, \Gamma'} \text{(FIELDS)}$$

Fields work like variables, but their order is insignificant. The turnstile of the (FIELDS) judgement it subscripted with the object's own identity which is extended by the current field name in (FIELDS). If field f in object n has unique value v of type t, the proof tree for the stack will contain the judgement $\Gamma \vdash_{n.f} v :: t$ for some store type Γ.

5.5.6 Values

The looking up of types in Γ is a recursive function that remembers the object of the previous level of nesting and uses that for owner.

$$
\begin{aligned}
\mathsf{nil}(n)_p &::= \bot \qquad \text{no valid type for } n \\
(\Gamma \bullet \Gamma')(n)_p &::= \begin{cases} \Gamma(n)_p & \text{if } n \in \mathsf{defs}(\Gamma) \\ \Gamma'(n)_p & \text{otherwise} \end{cases} \\
(\alpha \mapsto m, \Gamma)(n)_p &::= \Gamma(n)_p \\
(x :: t, \Gamma)(n)_p &::= \Gamma(n)_p \\
(m :: T[\Gamma], \Gamma')(n)_p &::= \begin{cases} p{:}c\langle\sigma\rangle & \text{if } n = m \text{ where } T = c\langle\sigma\rangle \\ \Gamma(n)_m & \text{if } n \in \mathsf{defs}(\Gamma) \\ \Gamma'(n)_m & \text{otherwise} \end{cases} \\
\Gamma(n) &::= \Gamma(n)_{\texttt{world}}
\end{aligned}
$$

A more informal definition of the same function that might be more easily understood is $\Gamma(n) = p : c\langle\sigma\rangle$ iff $\Gamma = \Gamma'\langle n :: c\langle\sigma\rangle[\_]\rangle_p$, that is, if Γ can be factored into a Γ' with a hole in p (possibly `world`) such that the object n is directly inside with (incomplete) type $c\langle\sigma\rangle$, then the type of n in Γ is $p{:}c\langle\sigma\rangle$.

$$
\frac{\Gamma \vdash t \quad \Gamma(m) = t}{\Gamma \vdash_n \uparrow m :: t}\ (\text{VAL-POINTER}) \qquad
\frac{\Gamma \vdash t}{\Gamma \vdash_n \mathit{null} :: t}\ (\text{VAL-NULL}) \qquad
\frac{\Gamma \vdash_n v :: t' \quad \Gamma \vdash t' \leq t}{\Gamma \vdash_n v :: t}\ (\text{VAL-SUBSUMPTION})
$$

By (VAL-POINTER), a pointer is well-formed if its type derived from looking its id up in Γ is well-formed in Γ. By (VAL-NULL), *null* can have any well-formed type. By (VAL-SUBSUMPTION), a value can be viewed as having a supertype to that of its actual type.

$$
\frac{\begin{array}{c}\Gamma'' = \Gamma\langle n :: \mathfrak{U}[\Gamma']\rangle_b \quad \Gamma \vdash \texttt{unique}_b{:}c\langle\sigma\rangle \\ \Gamma'' \vdash_n \uparrow m :: n{:}c\langle\sigma\rangle \quad \Gamma''; n \vdash H \gg \Gamma'\end{array}}{\Gamma \vdash_n \mathsf{U}_n[\uparrow m; H] :: \texttt{unique}_b{:}c\langle\sigma\rangle}\ (\text{VAL-UNIQUE})
$$

Finally, a unique value is well-formed if its pointer compartment and nested

subheap are well-formed under an extended store type where the type information from the unique is added in. As for borrowing blocks, the bound b of the unique determines where the type information of the unique is inserted in the store type. Also, the unique type must be well-formed under the original store-type where the unique's contents are not visible.

Here, it might look as if we are "pulling type information out of nowhere", but this is deceiving; note the definition of Γ'' and the type information parallel to H. As $n :: \mathfrak{U}[\Gamma']$ is "hidden from the rest of the store-type", no other judgement can depend on this information. This allows type information of uniques to change without affecting other parts of the system.

5.5.7 Static and Dynamic Types

$$\frac{\Gamma \vdash p_1 : c\langle p_{i=2..n}\rangle \quad \Gamma \vdash p_i = q_i \text{ for } i = 1..n}{\Gamma \vdash p_1 : c\langle p_{i=2..n}\rangle = q_1 : c\langle q_{i=2..n}\rangle} \text{(TYPE-EQUAL)}$$

Again, due to the existence of both static and actual owners in our system, we need rules to treat these as equal. Recall (OWNER-EQUAL); basically, a type with static owners is equal to a type with the equivalent actual owners.

$$\frac{\Gamma \vdash t_1' \leq t_2' \quad \Gamma \vdash t_1' = t_1 \quad \Gamma \vdash t_2' = t_2}{\Gamma \vdash t_1 \leq t_2} \text{(SUB-TYPE-EQUAL)} \qquad \frac{\Gamma \vdash e :: t' \quad \Gamma \vdash t = t'}{\Gamma \vdash e :: t} \text{(EXPR-TYPE-EQUAL)}$$

Similar to (TYPE-EQUAL) above, (SUB-TYPE-EQUAL) defines subtyping relations that take static and actual owner equality into consideration and (EXPR-TYPE-EQUAL) does the same for types of expressions.

5.6 Operational Semantics for Joline

5.6.1 Variable lookup and Assignment

Variable lookup and update on a stack is written $S(x)$ and $S[x \mapsto v]$ respectively. Field lookup is written $(S)_{n.f}$ for lookup of the contents of field f in the object with id n on stack S. Field update is written $(S)_{n.f} := v$. They all have the obvious semantics (see Appendix A.1 and A.2 for the full story).

5.6.2 Dispatch

The help function $\mathcal{D}_t(md)$ returns a tuple with type information and method body of the method that the message md is bound to when passed to an object of type t. It also returns the type of the receiver as viewed by the method body.

$$\mathcal{D}_{p:c\langle\sigma\rangle}(md) = \begin{cases} \bot, & \text{if } c \equiv \texttt{Object} \\ (\Omega, t'\ x \mapsto t, b, p{:}c\langle\sigma\rangle), & \\ \qquad \text{if } \texttt{class}\, c \cdots \{ \cdots \langle\Omega\rangle\ t\ md(t'\ x)\{b\} \cdots \} \in P & \\ \mathcal{D}_{p:c'\langle\sigma(\sigma')\rangle}(md), & \text{if } \texttt{class}\ c\langle\_\rangle\ \texttt{extends}\ c'\langle\sigma'\rangle \cdots \in P \end{cases}$$

where $\Omega = \alpha_i\ \mathsf{R}_i\ p_{i\in 1..n}$. Just as is the definition of $\mathcal{M}_c$, we omit md not in the body of c in the bottom case, for presentation reasons.

5.6.3 Expressions

$$\frac{S(\texttt{this}) = \uparrow n}{\langle S \mid \texttt{this}\rangle \rightarrow \langle S \mid \uparrow n\rangle}\ (\textsc{expr-this}) \qquad \frac{}{\langle S \mid null\rangle \rightarrow \langle S \mid null\rangle}\ (\textsc{expr-null}) \qquad \frac{S(x) = v}{\langle S \mid x\rangle \rightarrow \langle S \mid v\rangle}\ (\textsc{expr-var})$$

$$\frac{S(x) = \uparrow n \quad (S)_{n.f} = v}{\langle S \mid x.f\rangle \rightarrow \langle S \mid v\rangle}\ (\textsc{expr-field})$$

Without loss of generality, we use "named form" for the expressions in order to simplify the formal account of Joline. We only allow field lookup, field update and method calls to be performed local variables and not directly on the result of an expression. Thus, instead of writing $e.f$, we write $x = e; x.f$, where x is a variable of the appropriate type. Clearly, the forms are equivalent. Given the definitions of variable and field lookup above, (EXPR-FIELD) is straightforward. We lookup the value of x in S, which must be a pointer, and then perform a field lookup on field f on the appropriate object using the helper function $(S)_{n.f}$.

$$\frac{S(x) = v}{\langle S \mid x\text{--}\rangle \rightarrow \langle S[x \mapsto null] \mid v[\text{free}/x]\rangle}\ (\textsc{expr-dread-local}) \qquad \frac{S(x) = \uparrow n \quad (S)_{n.f} = v}{\langle S \mid x.f\text{--}\rangle \rightarrow \langle (S)_{n.f} := null \mid v[\text{free}/n.f]\rangle}\ (\textsc{expr-dread-field})$$

Unique local variables and fields must be read using the destructive read operation. The operational semantics for the destructive reads is similar to (EXPR-FIELD) and (EXPR-LOCAL), except that the field or variable is updated with null. The unique value is also given the owner *free*, which corresponds to it being a free value. This is denoted by the substitution of x or $n.f$ for free.

As the destructive read operation is only allowed on unique variables or fields, v above is either *null* or on the form $\mathsf{U}_p[v; H]$, where p is x in (EXPR-DREAD-LOCAL) and $n.f$ in (EXPR-DREAD-FIELD). Thus, if variable x is destructively read when $S(x) = \mathsf{U}_x[v; H]$, it results in the stack $S[x \mapsto null]$, where x is *null*. The result of the operation is $\mathsf{U}_{\text{free}}[v; H[\text{free}/x]]$, the unique value "moved to free".

$$\frac{V = f \mapsto null \quad \text{for all } f \in dom(\mathcal{F}_c) \qquad n \text{ is fresh}}{\langle S \,|\, \texttt{new}\ p{:}c\langle\sigma\rangle\rangle \rightarrow \langle S\langle n \mapsto c^{\sigma}[V; \mathsf{nil}]\rangle_p \,|\, \uparrow n\rangle} \text{(EXPR-NEW)}$$

On creation, the object is given a fresh id, the owner p, an empty subheap, and all its fields are initialised with *null*. The object is then stored in the heap in the subheap of its owner. The result of the expression is a pointer to the object.

$$\frac{\begin{array}{c}\langle S \,|\, e\rangle \rightarrow \langle S_1 \,|\, v\rangle \qquad S_1(x) = \uparrow n \qquad S_1 = S'\langle n \mapsto c^{\sigma}[\_]\rangle_m \\ \mathcal{D}_{m:c\langle\sigma\rangle}(md) = (\alpha\, \mathsf{R}\, \_, \_\ y \rightarrow \_, s; \texttt{return}\ e', m{:}c_2\langle\sigma_2\rangle) \\ \langle S_1 \bullet \sigma_2{}^m_n \oplus \texttt{this} \mapsto \uparrow n \oplus \alpha \mapsto p \oplus y \mapsto v \,|\, s\rangle \rightarrow \langle S_2\rangle \quad \langle S_2 \,|\, e'\rangle \rightarrow \langle S_3 \bullet F \,|\, v'\rangle\end{array}}{\langle S \,|\, x.md\langle p\rangle(e)\rangle \rightarrow \langle S_3 \,|\, v'\rangle} \text{(EXPR-CALL)}$$

The dynamic semantics for the method call is pretty straightforward. A new stack frame is created with the owner parameters of the receiver's type and the receiver as `this`. The reference argument is pushed onto the stack as well.

The $\mathcal{D}_t(md)$ function returns the actual method body invoked by the message md when sent to the type t and the type of `this` in that method body. Informally the type is the most specific supertype $p : c\langle\sigma\rangle$ of t such that $t \leq p{:}c\langle\sigma\rangle$ and c defines method md and the method body of that definition. Note that we write $\sigma_2{}^m_n$ for $\sigma_2 \cup \{\texttt{owner} \mapsto m\} \cup \{\texttt{this} \mapsto n\}$.

Last, the method body is evaluated; the resulting value v' is returned and the top-most frame is removed.

$$\frac{\langle S \,|\, e\rangle \rightarrow \langle S' \,|\, \mathsf{U}_{\text{free}}[v; H]\rangle}{\langle S \,|\, (p)\ e\rangle \rightarrow \langle S'\langle H[p/\text{free}]\rangle_p \,|\, v\rangle} \text{(EXPR-LOSE-UNIQUENESS)} \qquad \frac{\langle S \,|\, e\rangle \rightarrow \langle S' \,|\, null\rangle}{\langle S \,|\, (p)\ e\rangle \rightarrow \langle S' \,|\, null\rangle} \text{(EXPR-LOSE-UNIQUENESS2)}$$

(EXPR-LOSE-UNIQUENESS) is a "cast" from the *free* owner to another. The uniqueness wrapper is discarded and the subheap compartment of the unique is moved into the subheap of the target owner. The pointer compartment of the unique is the resulting value of the expression.

5.6.4 Statements

$$\frac{\langle S \mid e\rangle \to \langle S' \mid v\rangle}{\langle S \mid t\ x = e\rangle \to \langle S' \oplus x \mapsto v[x/\text{free}]\rangle}\ (\textsc{stat-local}) \qquad \frac{\langle S \mid e\rangle \to \langle S' \mid v\rangle}{\langle S \mid x := e\rangle \to \langle S'[x \mapsto v[x/\text{free}]]\rangle}\ (\textsc{stat-update})$$

Local variable declaration and initialisation is straightforward: a binding from the variable name to its value is appended to the stack. Local variable update is equally trivial and works as expected. If a free value is assigned to the local variable, the local variable becomes the owner of the unique reference, which is captured by the owner substitution $[x/\text{free}]$.

$$\frac{\langle S \mid e\rangle \to \langle S' \mid v\rangle \quad S'(x) = \uparrow n \qquad S'' = (S')_{n.f} := v[n.f/\text{free}]}{\langle S \mid x.f := e\rangle \to \langle S''\rangle}\ (\textsc{update-field})$$

Field update works as expected, using the previously described $(S)_{n.f} := v$ helper function. As for variables, owner substitution captures the case when capturing a free value in a field.

$$\frac{}{\langle S \mid \texttt{skip}\rangle \to \langle S\rangle}\ (\textsc{stat-skip}) \qquad \frac{\langle S \mid e\rangle \to \langle S' \mid v\rangle}{\langle S \mid e\rangle \to \langle S'\rangle}\ (\textsc{stat-expr}) \qquad \frac{\langle S \mid s_1\rangle \to \langle S''\rangle \qquad \langle S'' \mid s_2\rangle \to \langle S'\rangle}{\langle S \mid s_1\,;s_2\rangle \to \langle S'\rangle}\ (\textsc{stat-sequence})$$

The skip statement is trivial and (stat-expr) just evaluates an expression and discards the resulting value. From (stat-sequence), statements can be sequenced in an unsurprising fashion.

$$\frac{\langle S \oplus \mathsf{R}_n[\mathsf{nil}; \alpha \mapsto n] \mid s\rangle \to \langle S' \oplus \mathsf{R}_n[H; F]\rangle \quad n \text{ is fresh}}{\langle S \mid (\alpha)\ \{\ s\ \}\rangle \to \langle S'\rangle}\ (\textsc{scoped-region})$$

Evaluating scoped regions creates a new region and pushes it to the top of the stack. The region also acts as an owner. Initially, the heap is empty except for a mapping from the static owner name introduced by the region to the id of the region itself, in this case $\alpha \mapsto n$. After the region is created, its statement is evaluated. Then, the region is destroyed along with its contents.

$$\frac{\begin{array}{c}\langle S \oplus x \mapsto \mathit{null}, \mathsf{B}^b_n[H[n/x]; \alpha \mapsto n \oplus y \mapsto v] \mid s\rangle \to \\ \langle S' \oplus x \mapsto v'', \mathsf{B}^b_n[H'; \alpha \mapsto n \oplus y \mapsto v', F]\rangle \quad \text{where } n \text{ is fresh}\end{array}}{\begin{array}{c}\langle S \oplus x \mapsto \mathsf{U}_x[v; H] \mid \texttt{borrow}\ x :: \texttt{unique}_b : c\langle\sigma\rangle\ \texttt{as}\ \langle\alpha\rangle\ y\ \texttt{in}\ \{\ s\ \}\rangle \to \\ \langle S' \oplus x \mapsto \mathsf{U}_x[v'; H'[x/n]]\rangle\end{array}}\ (\textsc{stat-borrow})$$

(stat-borrow) show the operational semantics for our borrowing operation.

The borrowed variable is nullified and its contents is moved into a newly created block $\mathsf{B}^b_n[\ldots]$ pushed on top of the stack frame. The block contains a mapping from the static name of the borrowed owner and the actual owner, the identity of the borrowing block. The unique's pointer compartment is moved to the borrowing variable in the block, and the subheap compartment is moved (the substitution of x for n above) into the subheap compartment of the borrowing block. The statement of the borrowing block is then evaluated. When the block is exited, the uniqueness wrapper is recreated, the entire subheap of the borrowing block is moved back into it, along with the pointer in the borrowing variable. The unique value is stored in x and the remainder of the borrowing block is popped of the stack.

Having described Joline's dynamic semantics, we move on to the showing the soundness of our system and, in particular, that it enjoys the owners-as-dominator property.

6 Soundness of Joline

This section shows subject reduction and progress theorems for the Joline language, as well as the structural properties given by ownership types and external uniqueness respectively: *owners-as-dominators* and *external-uniqueness-as-dominating-edges*. We begin, however, by defining a few necessary helper functions.

6.1 Helper Functions

6.1.1 Definition of $\rightsquigarrow$

The symbol $\rightsquigarrow$ describes the relation between store-typings of different configurations. We only define $\rightsquigarrow$ for well-formed store-typings, thus well-formedness of the store-types on both sides of the $\rightsquigarrow$ symbol is implicit below.

$$\Gamma \rightsquigarrow \Gamma \qquad \Gamma \rightsquigarrow \Gamma\langle n :: c\langle\sigma\rangle\rangle \qquad \Gamma \rightsquigarrow \Gamma' \text{ if } \exists \Gamma'' \text{ s.t. } \Gamma \rightsquigarrow \Gamma'' \text{ and } \Gamma'' \rightsquigarrow \Gamma'$$

6.1.2 Definition of $\leq$

The symbol $\leq$ describes the how a store-typing may grow *during* the evaluation of a configuration.

$$\Gamma \leq \Gamma$$

$\Gamma \leq \Gamma'$, if $\Gamma \rightsquigarrow \Gamma'$
$\Gamma \leq \Gamma \oplus x :: t$
$\Gamma \leq \Gamma \oplus \alpha \mapsto n$
$\Gamma \leq \Gamma \oplus n :: \mathfrak{R}$
$\Gamma \leq \Gamma \oplus n :: \mathfrak{B}$
$\Gamma \leq \Gamma \bullet n :: \mathfrak{R}$
$\Gamma \leq \Gamma \bullet n :: \mathfrak{B}$
$\Gamma \leq \Gamma'$, if there exists Γ'' s.t. $\Gamma \leq \Gamma''$ and $\Gamma'' \leq \Gamma'$

Again, we only define $\leq$ for well-formed store-typings, so $\Gamma \leq \Gamma'$ implies $\Gamma \vdash \diamond$ and $\Gamma' \vdash \diamond$ in our system. We omit this for brevity.

Soundness is proven as a standard subject reduction theorem that states that types are preserved under evaluation.

Theorem 6.1 (Subject Reduction)

(1) If $\Gamma \vdash \langle S \mid e \rangle :: t$ *and* $\langle S \mid e \rangle \rightarrow \langle S' \mid v \rangle$*, then there exists a* Γ' *such that* $\Gamma \rightsquigarrow \Gamma'$ *and* $\Gamma' \vdash \langle S' \mid v \rangle :: t$*.*

(2) If $\Gamma \vdash \langle S \mid s \rangle$ *and* $\langle S \mid s \rangle \rightarrow \langle S' \rangle$*, then there exists a* Γ' *such that* $\Gamma \rightsquigarrow \Gamma'$ *and* $\Gamma' \vdash \langle S' \rangle$*.*

By structural induction over the shapes of e and s. The key problems of proving the theorem are *movement* and *visibility*. To preserve space, we omit this proof and direct the interested reader to the author's dissertation [14] for the full story. Key to dealing with movement is shown in the (VAL-UNIQUE) rule where the type information of the subheap nested in the unique is not included in the global store-type information. This allows movement to be formulated as an owner substitution operation on the nested store-type that is not visible to the rest of the program.

□

Lemma 6.2 (Canonical Forms) *If* $\Gamma \vdash S$ *and* $\Gamma \vdash v :: t$*, then the following holds for the possible forms of* v*:*

(1) If $t = p{:}c\langle\sigma\rangle$*, then either*

(a) $v = null$*, or*

(b) $v = \uparrow n$ *and,* $\Gamma(n) = p{:}c'\langle\sigma'\rangle$*,* $dom(\mathcal{F}_c) \subseteq dom(\mathcal{F}_{c'})$*,* $S(n) = c'^{\sigma'}[V; H]$ *and* $f \in dom(V)$ *for all* $f \in dom(\mathcal{F}_c)$*,* $dom(\mathcal{M}_c) \subseteq dom(\mathcal{M}_{c'})$ *and* $arity(\mathcal{M}_c(md)) = arity(\mathcal{M}_{c'}(md))$ *for all* $md \in dom(\mathcal{M}_c)$
(Arity is trivially defined as $arity(t_{i \in 1..m} \rightarrow t) = m$*).*

(2) If $t = \texttt{unique}_p : c\langle\sigma\rangle$*, then either,*

(a) $v = null$*, or*
(b) $v = \mathsf{U}_n[v; H]$*.*

From the syntax of v, there are three cases corresponding to the ones above. 1.a) and 2.a,b) are immediate from (VAL-NULL) and (VAL-UNIQUE). For 1.b), by (VAL-POINTER) and (VAL-SUBSUMPTION), $\Gamma(n) = p : c'\langle\sigma'\rangle$ and $\Gamma \vdash p : c'\langle\sigma'\rangle \leq p : c\langle\sigma\rangle$.

From $\Gamma \vdash S$ and $\Gamma(n) = p : c'\langle\sigma'\rangle$, clearly $S(n) = o$, for some o, as Γ and S are parallel. From $\Gamma \vdash S$ and $\Gamma \vdash \uparrow n :: p : c'\langle\sigma'\rangle$ we can derive (full details in [14]) $\Gamma; p \vdash n \mapsto o \gg \Gamma'$ for some Γ'. By (OBJECT), $f \in dom(V)$ for all $f \in dom(\mathcal{F}_{c'})$ where $o = c'^{\sigma'}[V; H]$.

By (CLASS) and definition of $\mathcal{F}$, $dom(\mathcal{F}_c) \subseteq dom(\mathcal{F}_{c'})$. Similarly, (CLASS) and def. of $\mathcal{M}$ implies $dom(\mathcal{M}_c) \subseteq dom(\mathcal{M}_{c'})$ and $arity(\mathcal{M}_c(arity)) = md(\mathcal{M}_{c'}(md))$. □

6.2 Progress

In this section, we present the progress lemma. Additional evaluation rules for the dynamic semantics that deal with trapping and propagating errors in the system are found in Appendix A.3. We trap only one kind of errors, null-pointer errors. Most of the error trapping rules are straight-forward and work as someone fluent in Java would expect.

Following Ernst et. al [24], we define a *finite evaluation* relation thus:

Definition 6.1 (Finite Evaluation) *An evaluation relation* $\rightarrow_k$*, which is a copy of the rules in the operational semantics (including the ones for error handling), where each occurrence of* $\rightarrow$ *in a premise is replaced by* $\rightarrow_{k-1}$*. For axioms and conclusions, replace* $\rightarrow$ *with* $\rightarrow_k$ *and add the following axioms:*

(STAT-KILL) (EXPR-KILL)

$$\frac{}{\langle S \mid s\rangle \rightarrow_0 \langle S \mid \texttt{ERROR}\rangle} \qquad \frac{}{\langle S \mid e\rangle \rightarrow_0 \langle S \mid \texttt{ERROR}\rangle}$$

This means that the evaluation will return with a "kill error", if the derivation is more than n derivations deep [24]. This allows us to state a progress lemma for a finite n and thus need not account for diverging evaluations due to infinite loops, which would terminate with a kill error when the number of derivations

exceeded n. We can thus describe progress:

Lemma 6.3 (Progress)

(1) If $\Gamma \vdash \langle S \mid s \rangle$*, then for all natural numbers* n*, there exists a* S' *such that either* $\langle S \mid s \rangle \to_n \langle S' \rangle$ *or* $\langle S \mid s \rangle \to_n \langle S' \mid \texttt{ERROR} \rangle$*.*

(2) If $\Gamma \vdash \langle S \mid e \rangle$*, then for all natural numbers* n*, there exists a* S' *such that either* $\langle S \mid e \rangle \to_n \langle S' \mid v \rangle$ *or* $\langle S \mid s \rangle \to_n \langle S' \mid \texttt{ERROR} \rangle$*.*

Following Ernst et al., a terminating expression is one for which there is an n such that the evaluation does not result in a kill error. If the application does not result in a kill error, then it cannot have used (EXPR-KILL) or (STAT-KILL) (the kill error would have been propagated), and thus, the derivation in $\to_n$ can be translated to a derivation in $\to$.

By mutual induction over the possible shapes of s and e. The key property is the completeness of the rules, that is, capturing all possible errors during evaluation and propagating them properly.

□

6.3 Structural Invariants

In this section, we formalise and prove that the owners-as-dominators (OAD) property holds for our system. In a well-formed configuration, all external aliasing of an object comes from its owner or siblings. We model this fact using holes—in a well-formed configuration that can be factored as a stack with a hole containing an object, there are no references from objects outside the hole on the same or previous frame to the *contents* of the object in the hole. Following OAD, we also formulate the containment invariant of external uniqueness, external-uniqueness-as-dominating-edges.

6.3.1 Helper Functions

The helper function uses denotes the set of all ids of all objects referenced by fields and variables in a stack. Its formal definition can be found in Appendix A.5.

Similarly, we define $\mathsf{defs}(S)$ to be the set of all identities of all objects, regions, borrowing blocks and uniques in S. Its formal definition can be found in Appendix A.5.

Last, we define the binary relation $\#$ for sets to mean that they are disjunct. For sets A and B are, $A \# B$ is defines to be $A \cap B = \emptyset$.

We now define the structural invariants in terms of uses, defs, and $\#$.

6.4 Owners-as-Dominators

To recapitulate, the owners-as-dominators property states that all paths from the root of the object graph to an objects must pass through the object's owner.

Let ι denote an object and r the root of an object graph. Paths have the shape $r \rightarrow \iota_1 \rightarrow \iota_2 \ldots \rightarrow \iota_n$. Thus, all paths start with r. In a system satisfying the owners-as-dominators property, all paths from r to any object always goes through the object's owner. For example, if ι_j is the owner of ι_i, ι_j will be on all paths from r to ι_i. Furthermore, for all objects ι_k with a reference to ι_i, ι_j will be on all paths from x to ι_k. The latter assures that only objects internal to the representation to which ι_i belong, ι_j in our example, can reference ι_j. Consequently, for an object to manipulate ι_i, it must either be internal to ι_i's owner, or invoke the change through ι_j's protocol.

We now prove owners-as-dominators for Joline using a slightly different formulation than the one found in Clarke's thesis [5]. We believe that our formulation is easier to understand as it is more clearly based on what parts of a stack or heap may reference an object.

Theorem 6.4 (Owners-as-Dominators) *If* $\Gamma \vdash S \bullet F\langle n \mapsto c^{\sigma}[V; H]\rangle$, *then* $\mathsf{defs}(H) \,\#\, (\mathsf{uses}(S) \cup \mathsf{uses}(F))$.

We prove this in two steps; 1) $\mathsf{defs}(H) \,\#\, \mathsf{uses}(S)$ and 2) $\mathsf{defs}(H) \,\#\, \mathsf{uses}(F)$. Note that we do not consider the case when n is nested inside a unique, as this case is covered by the stronger external-uniqueness-as-dominating-edges property.

(1) By contradiction. Assume the existence of a pointer $\uparrow m$ to an object of type t in H.

By (STACK-GEN), $\Gamma_1 \vdash S$ and $\Gamma_1 \bullet \Gamma_2 \vdash F\langle n \mapsto c^{\sigma}[V; H]\rangle \gg \Gamma_2$ where $\Gamma = \Gamma_1 \bullet \Gamma_2$. Note that as Γ_2 and $S \bullet F\langle n \mapsto c^{\sigma}[V; H]\rangle$ have parallel structure, $n \in \mathsf{defs}(\Gamma_2)$

Without loss of generality, we consider only the top-level of H. Thus, $\mathsf{owner}(t) = n$. There are two possible cases, either a) $\Gamma_1 \vdash \uparrow m :: t$, or b) $\Gamma_1\langle\Gamma_3\rangle \vdash \uparrow m :: t$ for some Γ_3. The latter covers the case when $\uparrow m$ originates from within a unique, in which case additional type information is available where the pointer is typed. In case a), $n \in \mathsf{defs}(\Gamma_1)$, by well-formed construction (omitted here, see [14] for the full story), which contradicts the unique names assumption as $n \in \mathsf{defs}(\Gamma_2)$. Case b) gives rise to a similar contradiction as it requires $n \in \mathsf{defs}(\Gamma_1\langle\Gamma_3\rangle)$. Thus, the

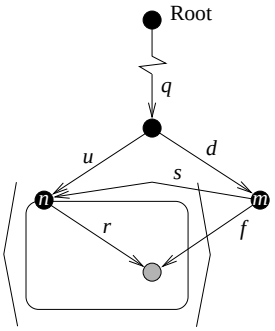


Figure 8. Possible paths. The $\langle\ \rangle$ denotes a hole in the store with n and its subheap as its contents. n is an object or a uniqueness wrapper.

pointer $\uparrow m$ cannot exist as it cannot be well-typed.

(2) By contradiction. Assume a) some x on F points into H or b) a field f in some object on F points into H.

Without loss of generality, we consider only objects on the top-level of H. Let $\uparrow m$ be a pointer to such an object. By (OBJECT), the type of $\uparrow m$ has owner n.

Case a) $F\langle\ \rangle(x) = \uparrow m$. For x to hold $\uparrow m$, the owner of the type of x must be n. By (STACK-GEN), $\Gamma_1 \vdash S$ and $\Gamma_1 \bullet \Gamma_2 \vdash F\langle n \mapsto c^\sigma[V; H]\rangle \gg \Gamma_2$ where $\Gamma = \Gamma_1 \bullet \Gamma_2$ and $\Gamma_2 = \Gamma_3\langle n :: c\langle\sigma\rangle[\_]\rangle$. Thus, $n \in \mathsf{defs}(\Gamma_2)$. It is clear from the static semantics that the only owners accessible on a stack frame are i) the owners of the type of the receiver or owner parameters (denoted Σ) and ii) owners of blocks created on the frame. Then:

i) By (EXPR-CALL), the type of the receiver and any owner parameters must be well-formed on the previous frame. Thus, $n \in \Sigma$ implies $\Gamma_1 \vdash n$, *i.e.,* $n \in \mathsf{defs}(\Gamma_1)$. As $n \in \mathsf{defs}(\Gamma_2)$, and object identities are unique, we have a contradiction.

ii) By (FRAME-BORROW) and (FRAME-REGION), either $\Gamma_2 = \Gamma_4\langle n :: \mathfrak{R}[\_]\rangle$ or $\Gamma_2 = \Gamma_4\langle n :: \mathfrak{B}[\_]\rangle$ which contradicts $\Gamma_2 = \Gamma_3\langle n :: c\langle\sigma\rangle[\_]\rangle$.

Clearly, the reference of kind a) cannot exist as x cannot be well-typed.

Case b) Let $\uparrow n_1$ of type $p : c_1\langle\sigma_1\rangle$ be the id of the object that contains f. By (OBJECT) and (FIELDS), $\mathsf{owner}(t) \subseteq rng(\sigma_1) \cup p \cup n_1 \cup \{\texttt{world}\}$ where t is f's type. Clearly, $\Gamma \vdash p : c_1\langle\sigma_1\rangle$. By (CLASS) and (TYPE), $\Gamma \vdash n_1 \prec^* p$ and $\Gamma \vdash p \prec^* q$ for all $q \in rng(\sigma_1)$. Thus, by (IN-TRANS) and (IN-REFL), $\Gamma \vdash n_1 \prec^* q$ for all $q \in \mathsf{owners}(t)$. This means that if t has owner n, then n_1 must be nested inside n. By (IN-*), this implies that either $n_1 = n$, $n = \texttt{world}$ (which is clearly not the case), or that n_1 is defined in H which contradicts that n_1 is defined in F. Thus, an object with such a field f cannot exist.

□

As an example, consider the picture in Figure 8. There are three possible paths to the grey object from the root: $q \to u \to r$, $q \to d \to s \to r$ and $q \to d \to f$. By the structural invariant, there may be no pointers to objects inside n from outside of n and thus, the path $q \to d \to f$ is invalid. As all other paths to the grey object go via n, n is a dominating node for it, meaning that the property is satisfied.

In a stack that satisfies the owners-as-dominators-property, any path to an object from the root of the object hierarchy must contain the object's owner. This means that the owner is a dominating node for all objects nested inside it [5].

We express that in the theorem as no fields or variables on the stack outside an object (outside the hole) can hold a reference to the contents of the object. The theorem does not deal with subsequent generations, as they are allowed full access to the object.

We believe that this formalisation of the containment invariant is easier to understand than the original containment invariant from Clarke's thesis [5]: $\iota \to \iota' \Rightarrow \iota \prec^* \mathsf{owner}(\iota')$, that is, "if object ι references the object ι', then ι is inside the owner of ι'. It is not trivial to understand what this means in terms of valid aliases in a system. *Additionally, our system also deals with stack variables on previous generations*, whereas the original formulation only considered paths from the root object in the system.

Theorem 6.5 (External-Uniqueness-as-Dominating-Edges)
If $\Gamma \vdash S\langle \mathsf{U}_n[v; H]\rangle$, *then* $(\mathsf{defs}(H) \cup \{n\}) \# \mathsf{uses}(S)$.

The theorem states that if S is a well-formed stack with a hole and a unique $\mathsf{U}_n[v; H]$ in the hole, then there are no pointers to n or H from any object in S. Thus, v is the only reference into H outside H and *is therefore a dominating edge*, that is an edge that appears on all paths from the root of the system to the object, as all paths into H from outside must contain it. (Observe that $v \in dom(H)$ by (VAL-UNIQUE).)

We prove this in two steps: a) $\mathsf{defs}(H) \# \mathsf{uses}(S)$ and b) $n \notin \mathsf{uses}(S)$.

First note that $\Gamma \vdash S\langle \mathsf{U}_n[v; H]\rangle$ implies $n \notin \mathsf{defs}(\Gamma)$ by (VAL-UNIQUE).

Case a) By contradiction. Assume the existence of a pointer $\Gamma' \vdash \uparrow m :: p : c\langle\sigma\rangle$ such that $m \in \mathsf{uses}(S)$ s.t. $m \in dom(H)$ Without loss of generality, assume that $\uparrow m$ points to a top-level object in H.

If $\uparrow m$ is stored in a field or value nested inside a unique in S, then $\Gamma' = \Gamma\langle\Gamma''\rangle$ where Γ'' is the additional type information visible inside the unique. If not, then $\Gamma' = \Gamma$, the type information for the whole stack.

By (UNIQUE-VAL), (HEAP-NESTED) and (OBJECT), $\Gamma' \vdash p \prec^* n$ (as p is

the owner of some object in H) and consequently, $\Gamma' \vdash n$, by (IN-*), the rules for owner orderings. The case $\Gamma' = \Gamma$ contradicts $n \notin \mathsf{defs}(\Gamma)$. The case $n \in \mathsf{defs}(\Gamma'')$ contradicts the unique names assumption, as n will be introduced a second time in Γ when typing the contents of the hole. Thus, the pointer $\uparrow m$ cannot exist.

Case b) Similar reasoning applies to proving the absence of uses of the unique itself. The existence of a well-formed pointer $\Gamma' \vdash \uparrow n :: t$ implies $\Gamma' \vdash n$ which was shown to be a contradiction above. If some field or variable in S contained $\mathsf{U}_n[v; H]$, this would contradict the unique names assumption.

□

7 Related Work

Alias encapsulation schemes (a large body of which are ownership types systems) have been employed for reasoning about programs, *e.g.,* Clarke and Drossopoulou [17] and Müller and Poetzsch-Heffter [25]; for alias management, *e.g.,* Clarke et al.[4], Noble et al.[3]; and in program understanding in the presence of aliasing [6]. Boyapati and Rinard [11] and Boyapati et al.[26] use ownership types as the basis for a system to eliminate data-races respective deadlocks from concurrent programs. Boyapati, Liskov and Shrira [27] use ownership types to enable safe lazy updates in object-oriented databases. In this thesis, we use alias encapsulation to overcome the abstraction problem inherent in extant proposals for unique pointers.

Good examples of alias encapsulation schemes are Islands [9], Confined Types [28], Universes [25] and Ownership Types [4]. Most other approaches are either reminiscent of these, or just weakened versions.

Islands was defined for Smalltalk [29] as a set of annotations guaranteeing that objects inside an island, a connected subgraph of the object graph, were not referenced from objects outside the island, except for the *bridge object*, which would then become *a single entry point to the island.* Via methods in the bridge object, objects could move in and out of an island. The encapsulation provided by Islands is among the strongest proposed. However (or, perhaps, subsequently), the practical usefulness of islands is questionable.

Confined types uses Java packages [30] as the protection domain: instances of classes that are package scoped may not be referenced from outside the package (*i.e.,* by instances of classes not defined in the package). Arguably, confined types is a more lightweight approach to alias encapsulation with a coarse grained level of protection. Studies by Grothoff, Palsberg and Vitek [31] of the structure of existing applications in the Purdue Benchmark Suite,

a large selection of programs, suggest that a quarter of all of classes satisfy the confinement properties of confined types. However, it is not clear what that means in practise, except that confined types may be quite compatible with existing ways of constructing object-oriented programs.

Universes [25] is basically an extended subset of ownership types for a Java-like language. The representation of an object conceptually belongs to a "universe" and references may not cross the universe boundary in any direction. Its only extension to ownership types is the introduction of a read-only pointer that may be used for cross-universe aliasing, but not for changing the referenced object (or, indeed, any object in the entire system). The read-only references can be used to implement iterators, which for long was problematic in ownership types. Read-only references is a form of alias control—allowing aliases while controlling their effect on a program. The concept is as basic as uniqueness, a read-only reference may not be used to change the referenced object. Read-only references have been used to either extend alias encapsulation proposals [9,25], or as stand-alone alias control schemes [32,33]. In the last case it is however unclear what practical gains stand from using them.

Ownership Types was proposed by Clarke, Noble and Potter [4]. In its original form, every object has an owner and references to representation objects are not allowed to be passed out of its owning object. In contrast to Islands, Balloon Types and Universes, aliases from internal objects to objects that own them are allowed; classes are parameterised with "permissions" to reference external objects. Ownership types can be used to enable both shallow and deep encapsulation.

While not primarily being a system for fortifying abstractions, DeLine and Fähndrich's work on Fugue [34] and typestates [13] is close in spirit to our work. Fugue implements the notion of typestates on an object-oriented language and allows checking that the protocol of an object is used correctly. For example, a socket's address and port must first be set before trying to establish a connection. Fugue relies heavily on uniqueness to track typestate through a program. The formalisation of Fugue [13] also uses adoption [12], a construct for mediating between unique and non-unique. Key differences is that our system uses destructive reads, and that type changes of a unique in our system can be witnessed by several objects internal to the object itself.

7.1 Comparison

Table 4 on page 48 presents a comparison between different proposals in the literature that include uniqueness, alias encapsulation and borrowing. Notably, external uniqueness is the only system that provides the strong encapsulation

	Uniqueness	Encapsulation	Borrowing
This paper	External	Deep	Orthogonal
PRFJ [a]	Conventional	Deep	Parameter
Flexible Alias Protection [b]	Free	Deep	n/a
Vault [c]	∼Conventional	Shallow	Orthogonal
AliasJava [d]	Conventional	Shallow	Parameter
Pivot Uniqueness [e](c)	<Conventional	Shallow	Parameter
Capabilities for sharing [f](d)	∼Conventional	∼Shallow	∼Parameter
Islands [g](e)	Conventional	Full	∼Parameter
Balloons [h]	Conventional	Full	Parameter
OOFX/Alias Burying [i]	Conventional	None	Parameter
Eiffel* [j]	Conventional	None	Parameter
Virginity [k]	Free	None	Parameter

Table 4
Comparison of related work. (*a*-*k* see text.)

of deep ownership and orthogonal borrowing. The italicised letters in the table are keys that are explained on page 50.

First we give short explanation of the various kinds of uniqueness, ownership, encapsulation and borrowing that we consider in our comparison.

7.1.1 Kinds of Uniqueness

We consider three kinds of uniqueness, all of which have been mentioned earlier in the thesis:

free values can be unique (*e.g.,* from object construction; cannot regain freeness once lost.).

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 references *into* an aggregate. Internal references to the 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 and cannot be regained. Commonly used synonyms for uniqueness include *linear* [35] and *unsharable* [1].

7.1.2 Kinds of Ownership and Encapsulation

We consider three kinds of ownership/encapsulation:

shallow direct access to certain objects is limited. This is similar to traditional uniqueness; moving a unique pointer from one object to another is effectively giving the receiving object shallow ownership over the unique.

deep the only access to the internal, transitive state of an object is through a single entry point. The entry point may be multiply referenced and references to external state is possible.

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

While shallow ownership prevents direct access to its representation objects, proxy objects may be created (internally or externally) which access the encapsulated objects and may escape the encapsulation boundary. This makes the encapsulation provided by shallow ownership intransitive.

Deep ownership goes further than shallow ownership 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 [3].

Full alias encapsulation, a term coined by Noble, Vitek and Potter [3] to describe *e.g.,* Hogg's Islands, offers a stronger, less flexible encapsulation than deep ownership since 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 bridge objects are cut points which break graph connectivity when removed. For a more in-depth, graph-based comparison between different models of encapsulation, see a recent paper by Noble, Biddle, Tempero, Potanin and Clarke [36].

In addition to the ones considered above, other forms of encapsulation exist, such as the package level confinement found in Confined Types [28]. These are however too coarse-grained to enable external uniqueness and are therefore not further discussed.

Kim, Bertino and Garza [37] 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. The machinery seems however to be in place to implement external uniqueness via dynamic checks.

7.1.3 Kinds of Borrowing

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.

Other names for borrowing are limited [38], temporary [39], lent [40], unconsumable [1] and unique [9]. None of these however use orthogonal borrowing.

Several proposals' implementations of borrowing weaken uniqueness by not preventing the original reference from being accessed during the borrowing. These proposals include Eiffel* [1], AliasJava [6], Balloon Types [41], Pivot Uniqueness [42] and Capabilities for sharing [10].

By using nullification or scope restrictions, this can be avoided at the price of race conditions and additional null-pointers as is done in PRFJ [11], Vault [34] and in Alias Burying [2].

Checking the constraints underlying alias burying modularly leads to an interdependence between uniqueness and read effects identified by John Boyland [19]. Guava [40] also uses lent parameters to avoid capturing of objects in a system for avoiding data races in Java.

7.1.4 Comments to the table

a) Parameterised Race-Free Java [11] 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.

PRFJ [11] was the first proposal to combine uniqueness and deep ownership types. PRFJ uses a straight-forward combination of uniqueness and ownership types in a single system, relying on traditional mechanisms to protect uniqueness of references instead of introducing a unique owner. Thus, PRFJ requires special borrowed pointers that cannot be stored safely on the heap and cannot support mediating between unique and non-unique as in Joline. While enabling unique pointers to entire aggregates PRFJ sadly proposal perpetuates the abstraction problem in the way we have described for method-level annotations. Additionally, PRFJ allows the unique keyword to appear in any

position of a class header, which violates parametricity [43], as the internal behaviour of a class in terms of *e.g.,* destructive reads, is affected by its owner parameters. Furthermore, PRFJ requires additional clauses to prevent certain owner parameters from being bound to unique in a type, which is effectively a kind of class-level annotation, breaking abstraction a second time due to uniqueness.

b) Flexible Alias Protection [3] was the starting point for ownership types. Its encapsulation model is slightly stronger since objects inside a protected boundary may not rely on mutable state of objects external to it.

c) The Vault system [34,12] gives a practical linear type system for a non object-oriented, imperative language. Our borrowing is similar to an *adopt operation* found in Vault that allows a linear (unique) reference to become non-linear (non-unique) temporarily by storing it into a non-unique object. While the scope of our borrowing is restricted to a certain block, the scope of adoption is the lifetime of the adopting object storing the previously linear pointer. Vault also provides a focus operation that enables a non-linear reference to be treated linearly and access to linear components in non-linear objects. This is achieved by a form of "aggressive alias burying" in the sense that the focus operation will not allow operations on any aliases to the focused object during the scope of the focus. This elegantly avoids destructive reads, but does not scale to multi-threaded class-based object-oriented programs since all valid pointers of the focused object must be accounted for in order for the focus operation to work.

d) AliasJava [6] only provides shallow encapsulation which does not suffice to implement external uniqueness since internal objects that may contain non-unique references to an externally unique object may escape. For a more detailed description, see Aldrich's dissertation [44].

e) Pivot Uniqueness [42] enables unique fields that can only be assigned with newly created objects or null. Pivotal encapsulation is shallow, guaranteeing only that the contents of a pivot field is never exported from an object, it may only be borrowed.

f) Capabilities for sharing [10] offers primitive and dynamic constructs that can be combined to enable various kinds commonly proposed constructs—uniqueness, read-only references etc., though no one specific policy is enforced. It presents an elegant unification of many popular constructs; two sets of access rights, one "base set" and one "exclusive set" are used to model the various mechanisms. Notably, uniqueness is the strongest, and the uniqueness capability includes the owner capability. Its constructs are shallow with the intention that systematic application of shallow mechanisms can be used to achieve deep versions. No static type system exists.

g) Islands [9] allow borrowing through read-only references.

h) Balloon Types [45,41] is discussed in more detail in [46].

i) OOFX/Alias Burying [47,2] avoids destructive reads by allowing violations of actual uniqueness as long as these violations are never witnessed. The alias burying solution to maintaining a strong uniqueness invariant would work well with external uniqueness, but would require an effects system to be modular. Alias burying is discussed throughout the thesis.

j) Eiffel*[1] is an early system bringing traditional uniqueness into object-oriented programming using method-level annotations to deal with subjective treatment of `this`.

k) Virginity [18] is basically free values obtained by object creation whose freeness is lost once assigned to a field or variable.

7.2 Region-based Memory Management

Our scoped region construct is similar to the lexically scoped `letregion` construct used in region-based memory management [48,49]. There are a number of differences. Firstly, our construct is under programmer control, as in Cyclone [20], 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 Clarke's dissertation [5]), whereas the regions calculus uses effects to determine that references into a deleted region are never dereferenced.

Both Cyclone [20] and Gay and Aiken's RC [50] manage a nesting relationship which captures when one object *outlives* another, very similar to how our system works. While some attempts to explicitly add region-based memory management to Java exist (see *e.g.,* [51,52]), they require interfaces to be extended with effects annotations to ensure modular checking, whereas our structural approach uses ownership and owner annotations. Recent work by Boyapati et al.add regions and ownership to Java to address the problems of Real-time Java [53]. (Other styles of effects system also exist for Java [47,17,11,26].) 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 [54] 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.

A number of systems in the literature combine linearity and regions [55,56], using linearity to track the use of regions to avoid the lexical scoping or region allocation and deallocation in the regions calculus.

7.3 Uniqueness and Linearity

Girard's linear logic [57] 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 [58,35]. Wadler's `let!` construct, quasi-linear types [59], and Vault's adoption and focus [12], 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 in an object-oriented setting.

Furthermore, we believe that the common linear typing restriction of preventing linear objects inside non-linear ones is not well-suited to object-oriented programming due to the inflexible nature of classes.

8 Conclusion

As we have previously stated, in a system with deep ownership types, adding unique pointers through the concept of unique owners is virtually for free as all the relevant mechanisms for dealing with owners are already in place. In addition to the unique owner, the only necessary additions are the borrowing statement and destructive reads. On the level of formalising external uniqueness and proving its soundness, the price is steeper.

As movement and borrowing causes change of owners and owners are recorded in types, movement and borrowing effectively change the type of an object. This change can be witnessed by several inside objects as well as the singular external owner. This complicates the formalisation, for example as movement changes what parts of the heap the unique aggregate may reference. We are not aware of the existence of any other formalisms with these properties, nor of any formalism of the dynamic semantics of uniqueness and destructive reads in an object-oriented programming language.

External uniqueness gives strong aliasing guarantees, and, as we have shown, can be achieved in a programming language with ownership types, virtually "for free". To us, external uniqueness is better suited to object-oriented programming than traditional uniqueness for several reasons:

(1) It considers aggregates—it is not possible, as it is with traditional uniqueness, to have a unique pointer to an object who's representation is shared.
(2) It does not break abstraction—an object is only active when it is borrowed, and thus non-unique, it is not possible for purely internal changes of an object to preclude it being referenced uniquely.
(3) It allows internal back-pointers, which allows more object structures to be used with external uniqueness than with traditional uniqueness.

Even though external uniqueness on the surface looks like a violation of the essence of uniqueness, external uniqueness is "unique enough": as there can be only one active pointer to a unique object at any time, external uniqueness is effectively unique.

We have presented a formal account of external uniqueness in our Joline language, outlined its soundness proof (the full details are available in the author's dissertation [14]), as well as formally proven owners-as-dominators and external-uniqueness-as-dominating-edges. The key difficulty of our formal system is the changing types of objects due to movement and borrowing, which required a slightly unorthodox, but perfectly sound and reasonable, use of store types. For the future, we hope to extract a more minimal "core calculus" of external uniqueness, to further explore its properties.

References

[1] N. Minsky, Towards alias-free pointers, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1996.

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

[3] J. Noble, J. Vitek, J. Potter, Flexible alias protection, in: E. Jul (Ed.), ECOOP'98—Object-Oriented Programming, Vol. 1445 of Lecture Notes In Computer Science, Springer-Verlag, Berlin, Heidelberg, New York, 1998, pp. 158–185.

[4] D. Clarke, J. Potter, J. Noble, Ownership types for flexible alias protection, in: Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications, 1998.

[5] D. Clarke, Object ownership and containment, Ph.D. thesis, School of Computer Science and Engineering, University of New South Wales, Sydney,

Australia (2001).

[6] J. Aldrich, V. Kostadinov, C. Chambers, Alias annotations for program understanding, in: Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications, 2002.

[7] C. Boyapati, Safejava: A unified type system for safe programming, Ph.D. thesis, Electrical Engineering and Computer Science, MIT (February 2004).

[8] D. Clarke, T. Wrigstad, External uniqueness is unique enough, in: L. Cardelli (Ed.), Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Vol. 2473 of Lecture Notes In Computer Science, Springer-Verlag, Darmstadt, Germany, 2003, pp. 176–200.

[9] J. Hogg, Islands: Aliasing protection in object-oriented languages, in: Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications, 1991.

[10] J. Boyland, J. Noble, W. Retert, Capabilities for Sharing: A Generalization of Uniqueness and Read-Only, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Vol. 2072, 2001.

[11] C. Boyapati, M. Rinard, A parameterized type system for race-free Java programs, in: Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications, 2001.

[12] M. Fähndrich, R. DeLine, Adoption and focus: Practical linear types for imperative programming, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2002.

[13] R. DeLine, M. Fähndrich, Typestates for objects, in: M. Odersky (Ed.), Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Vol. 3086, 2004.

[14] T. Wrigstad, Ownership-based alias management, Ph.D. thesis, Department of Computer and Systems Science, Royal Institute of Technology, Kista, Stockholm, submitted (May 2006).

[15] J. Noble, A. Potanin, Checking ownership and confinement properties, in: 4th Workshop on Formal Techniques for Java Programs, Malaga, Spain, 2002.

[16] D. Clarke, T. Wrigstad, External uniqueness, in: 10th Workshop on Foundations of Object-Oriented Languages (FOOL), New Orleans, LA, 2003.

[17] D. Clarke, S. Drossopolou, Ownership, encapsulation and the disjointness of type and effect, in: Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications, 2002.

[18] K. R. M. Leino, R. Stata, Virginity: A contribution to the specification of object-oriented software, Information Processing Letters 70 (2) (1999) 99–105.

[19] J. Boyland, The interdependence of effects and uniqueness, in: 3rd Workshop on Formal Techniques for Java Programs, 2001.

[20] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, J. Cheney, Region-based memory management in Cyclone, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2002.

[21] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler, Making the future safe for the past: Adding genericity to the Java programming language, in: Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications, 1998.

[22] M. Odersky, The Scala Language Specification 2.0, Programming Methods Laboratory, EFPL, Switzerland, 2006.

[23] H. G. Baker, Infant mortality and generational garbage collection, SIGPLAN Notices 28 (4) (1993) 55–57.

[24] E. Ernst, K. Ostermann, W. R. Cook, A virtual class calculus, in: Proceedings of Principles of Programming Languages (POPL), Charleston, South Carolina, USA, 2006.

[25] P. Müller, A. Poetzsch-Heffter, Universes: A type system for controlling representation exposure, in: A. Poetzsch-Heffter, J. Meyer (Eds.), Programming Languages and Fundamentals of Programming, Fernuniversität Hagen, 1999.

[26] C. Boyapati, R. Lee, M. Rinard, Ownership types for safe programming: Preventing data races and deadlocks, in: Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications, 2002.

[27] C. Boyapati, B. Liskov, L. Shrira, Ownership types and safe lazy upgrades in object-oriented databases, Tech. Rep. MIT-LCS-TR-858, Laboratory for Computer Science, MIT (July 2002).

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

[29] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[30] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison-Wesley, 1996.

[31] C. Grothoff, J. Palsberg, J. Vitek, Encapsulating objects with confined types, in: Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications, 2001.

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

[33] M. Skoglund, T. Wrigstad, Alias control with read-only references, in: Sixth Conference on Computer Science and Informatics, 2002.

[34] R. DeLine, M. Fähndrich, Enforcing high-level protocols in low-level software, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2001, pp. 59–69.

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

[36] J. Noble, R. Biddle, E. Tempero, A. Potanin, D. Clarke, Towards a model of encapsulation, in: D. Clarke (Ed.), International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming, UU-CS-2003-030, Utrecht University, 2003.

[37] W. Kim, E. Bertino, J. F. Garza, Composite objects revisited, in: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, 1989, pp. 337–347.

[38] E. C. Chan, J. T. Boyland, W. L. Scherlis, Promises: Limitied specifications for analysis and manipulation, in: IEEE International Conference on Software Engineering (ICSE), 1998.

[39] G. Kniesel, Encapsulation = visibility + accessibility, Tech. Rep. IAI-TR-96-12, Universiät Bonn, revised March 1998 (November 1996).

[40] D. F. Bacon, R. E. Strom, A. Tarafdar, Guava: a dialect of Java without data races, in: Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications, 2000, pp. 382–400.

[41] P. S. Almeida, Control of object sharing in programming languages, Ph.D. thesis, Department of Computing, Imperial College of Science, Technology, and Medicine, University of London (June 1998).

[42] K. R. M. Leino, A. Poetzsch-Heffter, Y. Zhou, Using data groups to specify and check side effects, in: Proceedings of the ACM Conference on Programming Language Design and Implementation, 2002.

[43] J. C. Reynolds, Towards a theory of type structure, in: B. Robinet (Ed.), Programming Symposium, Vol. 19 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1974, pp. 408–425.

[44] J. Aldrich, Using types to enforce architectural structure, Ph.D. thesis, University of Washington (August 2003).

[45] P. S. Almeida, Balloon Types: Controlling sharing of state in data types, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Vol. 1241, 1997.

[46] T. Wrigstad, External uniqueness: A theory of aggregate uniqueness for object-orientation, licentiate Thesis, Department of Computer and Systems Sciences, Stockholm University. (September 2004).

[47] A. Greenhouse, J. Boyland, An object-oriented effects system, in: ECOOP'99 — Object-Oriented Programming, 13th European Conference, no. 1628 in Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, New York, 1999, pp. 205–229.

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

[49] M. Tofte, J.-P. Talpin, Region-Based Memory Management, Information and Computation 132 (2) (1997) 109–176.

[50] D. Gay, A. Aiken, Language support for regions, in: ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, 2001.

[51] B. N. 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).

[52] M. V. Christiansen, P. Velschrow, Region-based memory management in Java, Master's thesis, Department of Computer Science (DIKU), University of Copenhagen (May 1998).

[53] C. Boyapati, A. Salcianu, W. Beebee, M. Rinard, Ownership types for safe region-based memory management in real-time java, in: ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI), 2003.

[54] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, M. Turnbull, The Real-Time Specification for Java, Addison-Wesley, 2000.

[55] D. Walker, K. Watkins, On regions and linear types, in: International Conference on Functional Programming, 2001, pp. 181–192.

[56] K. Crary, D. Walker, G. Morrisett, Typed memory management in a calculus of capabilities, in: 1999 Symposium on Principles of Programming Languages, 1999.

[57] J.-Y. Girard, Linear logic, Theoretical Computer Science 50 (1987) 1–102.

[58] P. Wadler, Linear types can change the world!, in: M. Broy, C. B. Jones (Eds.), IFIP TC 2 Working Conference on Programming Concepts and Methods, North-Holland, Sea of Gallilee, Israel, 1990, pp. 561–581.

[59] N. Kobayashi, Quasi-linear types, in: 26th ACM Symposium on Principles of Programming Languages, 1999.

A Appendix

A.1 Variable Look-up and Assignment

The look-up-function used in (EXPR-THIS) and (EXPR-VAR), looks up the variable on the top-frame in the stack and is defined thus ($\perp$ denotes that the look-up or update was unsuccessful):

$$
\begin{aligned}
\mathsf{nil}(x) &::= \bot \\
(S \bullet F)(x) &::= F(x) \\
(F \oplus \mathsf{B}_n^m[S; F'])(x) &::= F(x) \text{ where } x \in \mathsf{vars}(F) \\
(F \oplus \mathsf{B}_n^m[S; F'])(x) &::= F'(x) \text{ where } x \in \mathsf{vars}(F') \\
(F \oplus \mathsf{R}_n[S; F'])(x) &::= F(x) \text{ where } x \in \mathsf{vars}(F) \\
(F \oplus \mathsf{R}_m[S; F'])(x) &::= F'(x) \text{ where } x \in \mathsf{vars}(F') \\
(F \oplus \alpha \mapsto n)(x) &::= F(x) \\
(F \oplus y \mapsto \uparrow n)(x) &::= F(x) \qquad x \neq y \\
(F \oplus x \mapsto \uparrow n)(x) &::= \uparrow n
\end{aligned}
$$

We write $S[x \mapsto v]$ to mean the stack S where the variable x in the topmost frame is updated with the value v.

$$
\begin{aligned}
\mathsf{nil}[x \mapsto v] &::= \bot \\
(S \bullet F)[x \mapsto v] &::= S \bullet (F[x \mapsto v]) \\
(x' \mapsto v', F)[x \mapsto v] &::= x' \mapsto v', (F[x \mapsto v]) \\
(x \mapsto v', F)[x \mapsto v] &::= x \mapsto v, F \\
(\alpha \mapsto n, F)[x \mapsto v] &::= \alpha \mapsto n, (F[x \mapsto v]) \\
\mathsf{R}_n[H; F][x \mapsto v] &::= \mathsf{R}_n[H; F[x \mapsto v]] \\
\mathsf{B}_n^b[H; F][x \mapsto v] &::= \mathsf{B}_n^b[H; F[x \mapsto v]]
\end{aligned}
$$

A.2 Field Look-up and Assignment

The helper functions $(S)_{n.f}$ and $(S)_{n.f} := v$ are shorthands for reading respective updating the field f in the object with id n in stack S with *null*. They are formally defined thus:

$$
\begin{aligned}
(\mathsf{nil})_{n.f} &::= \bot \\
(S \bullet F)_{n.f} &::= \begin{cases} (S)_{n.f} & \text{if } n \in \mathsf{defs}(S) \\ (F)_{n.f} & \text{otherwise} \end{cases} \\
(F \oplus x \mapsto \_)_{n.f} &::= (F)_{n.f} \\
(F \oplus \alpha \mapsto \_)_{n.f} &::= (F)_{n.f} \\
(\mathsf{R}_m[H; F])_{n.f} &::= \begin{cases} (H)_{n.f} & \text{if } n \in \mathsf{defs}(H) \\ (F)_{n.f} & \text{otherwise} \end{cases} \\
(\mathsf{B}_m^b[H; F])_{n.f} &::= \begin{cases} (H)_{n.f} & \text{if } n \in \mathsf{defs}(H) \\ (F)_{n.f} & \text{otherwise} \end{cases}
\end{aligned}
$$

$$(n' \mapsto c^{\sigma}[V;H], H')_{n.f} ::= \begin{cases} V(f) & \text{if } n = n' \\ (H)_{n.f} & \text{if } n \neq n' \text{ and } n \in \mathsf{defs}(H) \\ (H')_{n.f} & \text{otherwise} \end{cases}$$

respective (for field update):

$$\begin{aligned}
(\mathsf{nil})_{n.f} := v ::= {} & \bot \\
(S \bullet F)_{n.f} := v ::= {} & \begin{cases} (S)_{n.f} := v \bullet F & \text{if } n \in \mathsf{defs}(S) \\ S \bullet (F)_{n.f} := v & \text{otherwise} \end{cases} \\
(F \oplus x \mapsto v')_{n.f} := v ::= {} & x \mapsto v', (F)_{n.f} := v \\
(F \oplus \alpha \mapsto m)_{n.f} := v ::= {} & \alpha \mapsto m, (F)_{n.f} := v \\
(n' \mapsto o, H)_{n.f} := v ::= {} & \begin{cases} n' \mapsto o, (H)_{n.f} := v & \text{if } n \neq n' \text{ and } n \in \mathsf{defs}(H) \\ (n' \mapsto o)_{n.f} := v, H & \text{otherwise} \end{cases} \\
(n' \mapsto c^{\sigma}[V;H])_{n.f} := v ::= {} & \begin{cases} n' \mapsto c^{\sigma}[V[f \mapsto v]; H] & \text{if } n = n' \\ n' \mapsto c^{\sigma}[V; (H)_{n.f} := v] & \text{otherwise} \end{cases}
\end{aligned}$$

A.3 Error Trapping Rules

The additional, error trapping rules for Joline.

$$\frac{S(\texttt{this}) = \mathit{null}}{\langle S \mid \texttt{this} \rangle \rightarrow \langle S \mid \texttt{ERROR} \rangle}\ (\textsc{expr-this}) \qquad \frac{S(x) = \mathit{null}}{\langle S \mid x.f \rangle \rightarrow \langle S \mid \texttt{ERROR} \rangle}\ (\textsc{expr-field-err})$$

$$\frac{S(x) = \mathit{null}}{\langle S \mid x.f := e \rangle \rightarrow \langle S \mid \texttt{ERROR} \rangle}\ (\textsc{update-field-err-1})$$

The rule (EXPR-THIS) captures an attempt to look-up `this` is a context where this is not defined. The rules (EXPR-FIELD-ERR) and (UPDATE-FIELD-ERR-1) trap looking up, or updating, a field on a null-pointer.

$$\frac{S(x) = \mathit{null}}{\langle S \mid x.f\text{--} \rangle \rightarrow \langle S \mid \texttt{ERROR} \rangle}\ (\textsc{expr-dread-field-err}) \qquad \frac{S(x) = \mathit{null}}{\langle S \mid x.md\langle \_ \rangle(e) \rangle \rightarrow \langle S \mid \texttt{ERROR} \rangle}\ (\textsc{expr-call-err-1})$$

The rule (EXPR-DREAD-FIELD-ERR) traps destructively reading a field on a null-pointer. Similarly, (EXPR–CALL-ERR-1) traps invoking a method on a null-pointer receiver.

$$\frac{S(x) = \mathit{null}}{\langle S \mid \texttt{borrow}\ x\ t\ \texttt{as}\ \langle p \rangle\ y\ \texttt{\{}\ s\ \texttt{\}} \rangle \rightarrow \langle S \mid \texttt{ERROR} \rangle} \text{(STAT-BORROW-ERR-1)}$$

By (STAT-BORROW-ERR-1), borrowing a null value will not be successful.

A.4 Error Propagating Rules

The error propagating rules capture errors occuring in subexpressions or substatements, and propagate them.

$$\frac{\langle S \mid e \rangle \rightarrow \langle S \mid \texttt{ERROR} \rangle}{\langle S \mid x.md\langle \_ \rangle(e) \rangle \rightarrow \langle S \mid \texttt{ERROR} \rangle} \text{(EXPR-CALL-ERR-2)}$$

The rule (EXPR-CALL-ERR-2) states that errors occuring in the evaluation of the argument expressions will be propagated and the call not dispatched.

$$\frac{\begin{array}{c}\langle S \mid e \rangle \rightarrow \langle S_1 \mid v \rangle \quad S_1(x) = \uparrow n \quad S_1 = S'\langle n \mapsto c^{\sigma}[\_] \rangle_m \\ \mathcal{D}_{m:c\langle\sigma\rangle}(md) = (\alpha\, \mathsf{R}\, \_, \_\ y \rightarrow \_, s\texttt{;return}\ e', m\texttt{:}c_2\langle\sigma_2\rangle) \\ \langle S_1 \bullet \sigma_2{}^m_n \oplus \texttt{this} \mapsto \uparrow n \oplus \alpha \mapsto p \oplus y \mapsto v \mid s\texttt{;return}\ \ e \rangle \rightarrow \langle S_2 \mid \texttt{ERROR} \rangle\end{array}}{\langle S \mid x.md\langle p \rangle(e) \rangle \rightarrow \langle S_2 \mid \texttt{ERROR} \rangle} \text{(EXPR-CALL-ERR-3)}$$

The rule (EXPR-CALL-ERR-3) states that errors occuring in the evaluation of a method will be propagated.

$$\frac{\langle S \mid s \rangle \rightarrow \langle S' \mid \texttt{ERROR} \rangle \quad \vee \quad \langle S \mid s \rangle \rightarrow \langle S'' \rangle \ \wedge \langle S'' \mid s' \rangle \rightarrow \langle S' \mid \texttt{ERROR} \rangle}{\langle S \mid s\texttt{;}s' \rangle \rightarrow \langle S' \mid \texttt{ERROR} \rangle} \text{(STAT-SEQUENCE-ERR)}$$

The rule (STAT-SEQUENCE-ERR) states that errors occuring in sequences of statements will be propagated.

$$\frac{\langle S \mid e\rangle \to \langle S' \mid \texttt{ERROR}\rangle}{\langle S \mid (p)\ e\rangle \to \langle S' \mid \texttt{ERROR}\rangle}\ (\textsc{expr-lose-uniqueness-err}) \qquad \frac{\langle S \mid s\rangle \to \langle S' \mid \texttt{ERROR}\rangle}{\langle S \mid \langle p\rangle\ \texttt{\{}\ s\ \texttt{\}}\rangle \to \langle S' \mid \texttt{ERROR}\rangle}\ (\textsc{stat-scoped-err})$$

By (EXPR-LOSE-UNIQUENESS-ERR) and (STAT-SCOPED-ERR), errors occuring in the subexpressions or body of the scoped region will be propagated.

$$\frac{\langle S \oplus x \mapsto \mathit{null} \oplus \mathsf{B}^p_n[H[n/x]; y \mapsto v] \mid s\rangle \to \langle S' \mid \texttt{ERROR}\rangle \text{ where } n \text{ is fresh}}{\langle S \oplus x \mapsto \mathsf{U}_x[v; H] \mid \texttt{borrow}\ x\ t\ \texttt{as}\ \langle p\rangle\ y\ \texttt{\{}\ s\ \texttt{\}}\rangle \to \langle S' \mid \texttt{ERROR}\rangle}\ (\textsc{stat-borrow-err-2})$$

By (STAT-BORROW-ERR-2), errors occuring inside a borrowing block will be propagated.

$$\frac{\langle S \mid e\rangle \to \langle S \mid \texttt{ERROR}\rangle}{\langle S \mid x.f := e\rangle \to \langle S \mid \texttt{ERROR}\rangle}\ (\textsc{update-field-err-2})$$

$$\frac{\langle S \mid e\rangle \to \langle S \mid \texttt{ERROR}\rangle}{\langle S \mid x := e\rangle \to \langle S \mid \texttt{ERROR}\rangle}\ (\textsc{stat-update-err}) \qquad \frac{\langle S \mid e\rangle \to \langle S \mid \texttt{ERROR}\rangle}{\langle S \mid t\ x := e\rangle \to \langle S \mid \texttt{ERROR}\rangle}\ (\textsc{stat-local-err})$$

The rules above propagate errors occuring in the evaluation of the RHS expression.

A.5 Helper Functions for Structural Invariants

uses denotes the set of all ids of all objects referenced by fields and variables in a stack and is defined thus:

$$\mathsf{uses}(\mathsf{nil}) = \emptyset$$

$$
\begin{aligned}
\mathsf{uses}(S \bullet F) &= \mathsf{uses}(S) \cup \mathsf{uses}(F) \\
\mathsf{uses}(F \oplus \alpha \mapsto n) &= \mathsf{uses}(F) \\
\mathsf{uses}(F \oplus x \mapsto v) &= \mathsf{uses}(F) \cup \mathsf{uses}(v) \\
\mathsf{uses}(F \oplus \mathsf{R}_n[F'; H]) &= \mathsf{uses}(F) \cup \mathsf{uses}(F') \cup \mathsf{uses}(H) \\
\mathsf{uses}(F \oplus \mathsf{B}^b_n[F'; H]) &= \mathsf{uses}(F) \cup \mathsf{uses}(F') \cup \mathsf{uses}(H) \\
\mathsf{uses}(n \mapsto c^\sigma[V; H], H') &= \mathsf{uses}(V) \cup \mathsf{uses}(H) \cup \mathsf{uses}(H') \\
\mathsf{uses}(f \mapsto v, V) &= \mathsf{uses}(v) \cup \mathsf{uses}(V) \\
\mathsf{uses}(\mathsf{U}_n[\uparrow m; H]) &= \{m\} \cup \mathsf{uses}(H) \\
\mathsf{uses}(\uparrow n) &= \{n\} \\
\mathsf{uses}(null) &= \emptyset
\end{aligned}
$$

defs denotes the the set of all identities of all objects, regions, borrowing blocks and uniques on a stack and is defined thus:

$$
\begin{aligned}
\mathsf{defs}(S \bullet F) &= \mathsf{defs}(S) \cup \mathsf{defs}(F) \\
\mathsf{defs}(x \mapsto v, F) &= \mathsf{defs}(F) \\
\mathsf{defs}(\alpha \mapsto n, F) &= \mathsf{defs}(F) \\
\mathsf{defs}(\mathsf{R}_n[H; F]) &= \{n\} \cup \mathsf{defs}(H) \cup \mathsf{defs}(F) \\
\mathsf{defs}(\mathsf{B}_n[H; F]) &= \{n\} \cup \mathsf{defs}(H) \cup \mathsf{defs}(F) \\
\mathsf{defs}(n \mapsto c^\sigma[V; H], H') &= \{n\} \cup \mathsf{defs}(V) \cup \mathsf{defs}(H) \cup \mathsf{defs}(H') \\
\mathsf{defs}(f \mapsto \mathsf{U}_n[v; H]) &= \{n\} \cup \mathsf{defs}(H) \\
\mathsf{defs}(f \mapsto \uparrow n) &= \emptyset \\
\mathsf{defs}(f \mapsto null) &= \emptyset \\
\mathsf{defs}(\mathsf{nil}) &= \emptyset
\end{aligned}
$$