Tobias Wrigstad

Department of Computer and Systems Sciences
Stockholm University/KTH, Stockholm, Sweden

External Uniqueness

— A Theory of Aggregate Uniqueness for Object-Oriented Programming

August 9, 2004

Submitted to Stockholm University in partial fulfillment of the degree of Licentiate of Philosophy

Summary. Unique pointers is a simple and powerful concept. In an object-oriented programming language, they facilitate reasoning about the state of an object formally and informally, make resource management and memory management easier, and can void the need for synchronisation in the presence of multiple threads.

However, existing implementations of uniqueness are not properly suited to object-oriented programming. A unique pointer to a bridge object in an aggregate only considers the bridge object and not the rest of the aggregate. More importantly, we identify a problem with existing implementations that leads to a violation of one of the ground principles of object-orientation, the principle of abstraction. The problem stems from the fact that in existing implementations of uniqueness, uniqueness is a property of the referenced object, or even the object's class—not a consequence of how the object is being used externally; whether or not an object can be uniquely referenced (and the semantics of a methods invoked on a unique receiver) must be considered when designing a class.

To this end we propose a different form of uniqueness, *external uniqueness*, where all objects can be referenced uniquely, regardless of how they are implemented. An important consequence of our proposal is also the incorporation of innocuous internal pointers in the uniqueness definition. There may be an arbitrary number of references to a unique object, but only one of these may be visible externally. The resulting language is more powerful, in our opinion better suited to object-oriented programming, and also cleaner since it avoids the introduction of second-class constructs necessary in previous proposals to deal with the volatile nature of uniqueness which complicates the languages.

We base our proposal on ownership types, a system for enforcing encapsulation and provide a strong notion of aggregate. The symbiosis of external uniqueness and ownership types is mutually beneficial in that enabling external uniqueness in the presence of ownership types is virtually free, and lifts some of the restrictions in ownership types systems.

We present the design of external uniqueness along with a static semantics to guarantee the proper use of the constructs as well as a dynamic semantics as a basis for proving the important properties and soundness of our systems. We state the proofs and provide proof sketches, but save the complete technical account for the final PhD thesis.

This thesis is typeset in LaTeX, the best thing since sliced bread (and sticky tape).

Contents

1 Introduction 3
- 1.1 Object-Oriented Programming and Object Sharing 4
- 1.2 Uniqueness 5
  - 1.2.1 Uniqueness and Object-oriented Programming 7
- 1.3 Contributions 9
- 1.4 Outline 10

2 Aliasing, Uniqueness and Object-Oriented Programming 13
- 2.1 Coping with Aliasing 13
- 2.2 Uniqueness 16
  - 2.2.1 Maintaining Uniqueness 17
- 2.3 Inherent Problems with Uniqueness 25
  - 2.3.1 Unique Pointers to Aggregates 25
  - 2.3.2 Uniqueness and a Problem with Abstraction 26
  - 2.3.3 Problem Summary 27
- 2.4 Problem Analysis: Distinguishing Between Internal and External References 28

3 Ownership Types 29
- 3.1 What is Alias Encapsulation, Anyway? 29
- 3.2 Ownership Types and Insides and Outsides of Objects 30
  - 3.2.1 Shallow Ownership 31
  - 3.2.2 Deep Ownership 33
- 3.3 Combining Uniqueness and Ownership Types 40
  - 3.3.1 Motivation for Chosing Ownership Types 40

4 External Uniqueness 41
- 4.1 Tour de External Uniqueness 41
  - 4.1.1 Fields and Blocks as Owners 43

4.1.2 Owners are Dominating Edges 44
4.1.3 Externally Unique is Effectively Unique 44
4.1.4 Operations on Externally Unique References 45
4.1.5 Movement Bounds 49
4.2 Discussion 52
4.2.1 Owner-Polymorphic Methods 52
4.2.2 Generational Ownership 53
4.2.3 Constructors 53
4.2.4 Shallow Ownership does not Suffice 54

5 Formalising External Uniqueness 57
5.1 Introducing Joline 57
5.1.1 Syntax 57
5.1.2 Joline's Type System 59
5.1.3 Joline's Dynamic Semantics 70
5.1.4 Well-formed Store Typing 77
5.1.5 Well-formed Configuration 78
5.2 Proof Statements 82
5.2.1 Dominance Properties 82
5.3 Concluding Remarks 86

6 Applications and Extensions 87
6.1 Applications for External Uniqueness 87
6.1.1 Transfer of Ownership 87
6.1.2 Merging Representations 90
6.1.3 Simulating Borrowing and Orthogonality of Concepts 91
6.1.4 Movable Aliased Objects 94
6.1.5 The Initialisation Problem 98

7 Discussion 101
7.1 External Uniqueness for Object-Orientation 101
7.2 How External Uniqueness Overcomes the Abstraction Problem 102
7.3 Facilitating Reasoning about Objects 105
7.4 External Uniqueness in the Presence of Multiple Threads 106

8 Related Work 109
8.1 Alias Encapsulation: Containment, Ownership, etc. 109
8.1.1 Comparison 111
8.2 Region-based Memory Management 115
8.3 Uniqueness and Linearity 116
8.4 Originality 117

8.4.1 Hogg's Islands 117
8.4.2 Almeida's Balloon Types 118
8.4.3 Parameterised Race-free Java 119
8.4.4 Flexible Alias Protection 120

9 Conclusions and Future Work 121
9.1 Summary 121
9.2 Critique 121
9.2.1 Weaknesses Inherited from Ownership Types 122
9.2.2 The Joline System 123
9.2.3 The Price of External Uniqueness 123
9.2.4 Lack of Practical Results 124
9.3 Future Work 124

A Appendix 133
A.1 Elaborating Movement Bounds 133

List of Figures

1.1 Three characteristics not present in uniqueness. 10

2.1 Alias problem examples. 14
2.2 Actual uniqueness via destructive reads. 18
2.3 Effective uniqueness via alias burying 19
2.4 Class-level annotations 23
2.5 Using method-level annotations 24
2.6 Aggregate uniqueness 25

3.1 Example of shallow ownership 32
3.2 Reference permissions. 32
3.3 Using a proxy object to circumvent protection 33
3.4 A linked list and links using ownership types 37
3.5 Object graph for linked list example in Figure 3.4 37
3.6 Comparing uniqueness and deep ownership 38

4.1 Uniqueness vs. Ownership vs. External uniqueness 42
4.2 Mediating between external uniqueness and borrowing 46
4.3 Violation caused by movement in the presence of subtyping 50
4.4 Movement that violates deep ownership. 51
4.5 A problem with using `owner` in the extends-clause 51
4.6 Generational ownership 54

5.1 Possible supertypes. 65
5.2 Possible paths. 84

6.1 A Token ring implementation. 88
6.2 Transfer of ownership 88
6.3 Merging two doubly-linked lists 91
6.4 Passing a borrowed object as argument to a method. 92

6.5 Storing a borrowed reference on the heap. 93
6.6 Movable aliased objects. 95
6.7 The object graph for the doubly-linked code list in Figure 6.3 . . . 96
6.8 Object graph for the doubly-linked list with head and tail. 96
6.9 Head and tail pointers using movable aliased objects 97
6.10 Overcoming the initialisation problem 98

7.1 The `Server` class example from Figures 2.4 and 2.5 encoded with external uniqueness. 104

List of Tables

5.1 Syntax of Joline 58
5.2 Helper function definitions. 59
5.3 Judgements used in the static semantics. 62
5.4 Judgements for well-formed store typing. 77
5.5 Judgements for well-formed configurations. 78
5.6 Definition of uses; the set of pointers used in a store/stack. 83
5.7 Definition of defs; the set of ids of all objects defined in a store/stack. 83

8.1 Comparison of related work 112

Preface

Where man has not been to give them names,
objects on desert islands do not know what they are.
Taking no chances, they stand still, and wait.
Quietly, excited.
For hundreds of thousands of years.

Ivor Cutler

Too many people close to me have died during the completion of this dissertation (albeit not as a consequence of this work), most notably all my grand parents and also my first supervisor, Terttu Orci, to whom I am forever grateful; for getting me enrolled in the PhD programme, for teaching me about research and formalisms, and for being a (still) inspiring person to have known. This work is for all of you.

I'm also hugely indebted to Dave, for being the super supervisor, not only on my thesis work but on life itself ("eat your food" and "don't you know the laws of drying things on racks?") and for sharing some of his inner thoughts ("It strikes me as highly unlikely that the solar system ever occurred" and "I guess there are things you cannot do without breaking the laws of physics"). I believe everyone should be given a shortcut to essential knowledge such as the one I've gotten from him. I leave the rest of my praise for the PhD thesis. I hope you know I'm grateful.

I also want to thank Associate Professor Louise Yngström for pushing me forward when I needed it; Beatrice Åkerblom for letting me talk constantly about type systems and for treating me to that beer when my mind was about to blow; and to Henrik Bergström for doing some last-minute proof-reading.

My parents and my brother have probably seen less of me than they deserve (not necessarily a bad thing), and so has Emma (sometimes it is good

that you too cannot stop working). Finally, I want to thank Anders N, Anders S, Jan, Jenny, Jocke, Fredrik, Linus, Martin, Olle, Robin, Thomas, Thorbiörn, Wille et al. (forgetting someone, I'm sure), who constantly kept talking to me about *other things.*

Tobias

1

Introduction

A unique pointer is the single pointer to an object in a system. The holder of the unique pointer is the only object in the system that can access the referenced object which has many powerful consequences: it makes it easier to reason about the state of that object between invocations which facilitates various static analyses and proofs of properties of a program; it can void the need for synchronisation in a multi-threaded setting; it also makes it easier to handle resources and manage memory since for example the effect of deleting an object referenced by a unique pointer is localised.

We believe, however, that current implementations of uniqueness are designed at the wrong level of abstraction for object-oriented programming: objects are viewed as "flat entities" rather than as black boxes, *i.e.,* aggregates built up by complex collections of objects with arbitrary internal composition. Also, as it turns out, existing approaches to uniqueness display a severe problem—changes to implementation might leak out into the interface breaking the principle of abstraction, thus making software evolution more difficult as these might propagate throughout the system.

This thesis introduces *external uniqueness*, which we believe is a better, more natural way of adding uniqueness to an object-oriented programming language. Not only does external uniqueness allow unique pointers to black boxes or aggregates, it also overcomes the abstraction problem inherent in traditional uniqueness proposals and results in a cleaner system with fewer and orthogonal concepts.

We identify and describe the abstraction problem with traditional uniqueness as well as problems with weakening uniqueness, unnecessary complexity due to the introduction of non-orthogonal constructs and unnecessary restrictions due to design issues or weak type systems and show how external uniqueness overcomes these problems. We include a formal semantics of external uniqueness along with outlines and proof sketches of the most impor-

tant theorems and discuss the virtues and implications on programming in comparison with traditional uniqueness. We also show a number of applications of external uniqueness that were not possible to implement in earlier systems.

Interestingly enough, our system displays many properties of the early proposals for alias encapsulation, namely Hogg's Islands and Almeida's Balloon types. Our system is however less restrictive, allows uniqueness to black boxes and overcomes the abstraction problem.

This thesis is an extension of the results of previous work (Clarke and Wrigstad 2003a; Clarke and Wrigstad 2003b); most notably, the inclusion of a dynamic semantics of Joline *as well as proof statements and a number of extensions and deeper discussions not present in the abovementioned papers.*

Missing from this thesis are the proofs, which we save for the final PhD thesis. However, we include static and dynamic semantics, statements of the most important proofs and proof sketches.

1.1 Object-Oriented Programming and Object Sharing

Objects often need to interact in complex ways to perform useful operations. As systems become more complex and intricate, it makes less sense to speak about single objects—instead we talk about aggregates, collections of (possibly shared) objects that act together to form a whole.

The sharing of objects is a necessity for efficient software systems. Common programming idioms such as the observer pattern (Gamma, Helm, Johnson, and Vlissides 1994) or data structures as the doubly-linked list require sharing to work. Sharing is also necessary if we want the structure of our programs to model real-world entities, which has been a major selling point of object-orientation.

On the downside, sharing of objects makes it hard to reason about programs. To prove an invariant in compile-time, or even to be able to say something about the state of an object between method invocations, we need to track the use of pointers in a program. This is either tricky or impossible, depending on the nature of our analysis and whether or not we can analyse the entire code or just a portion of it, such as an individual class or module. Regardless of that fact, such analyses suffer from a combinatorial explosion of assertions that state whether or not an object is aliased. This usually makes the results weak or simply unobtainable.

Generally, to be able to say something about the behaviour of a program or to check whether a particular class invariant is satisfied, we need to be able to reason about pointers. For example, if pointers to an object's internal state

can be obtained by other objects, we must look beyond the object in point to understand how its state can be manipulated. Since pointers are generally unbounded and may flow from any part of the program to any other part, reasoning involving pointers is a complex and delicate matter.

To aid both formal and informal reasoning, several alias management approaches have been suggested. For example, by restricting the number of aliases to an object, we can more easily reason about the effects of changes to that object since the effects become more localised (*pointer restrictions*); by restricting what parts of an object may be referenced (and from what parts of a program), we can formulate properties that hold invariantly of external aliasing (*alias encapsulation*).

Recent years have seen a dramatic increase in research on object sharing, pointer restriction and alias encapsulation, starting with Hogg's Islands (1991) and the Geneva Convention on the Treatment of Object Aliasing (Hogg, Lea, Wills, de Champeaux, and Holt 1992). Today there are Balloon Types (Almeida 1998), Confined Types (Bokowski and Vitek 1999) and Ownership Types (Clarke, Potter, and Noble 1998), to name a few, and numerous proposals for unique pointers (see below).

An object's *representation* is the objects that an object is constructed from, *i.e.,* its aggregatees (not necessarily all objects it has pointers to). The general principle behind the proposals above is that an object's representation should be *encapsulated* inside the object and pointers to representation objects should not escape to untrusted objects. By employing any of these schemes, we achieve greater control of the pointers in our system, for the price of additional syntactic burdens, behavioural restrictions etc. Different proposals offer different levels of protection, flexibility and reasoning power, mostly depending on their definition of trusted object.

1.2 Uniqueness

Originally introduced in functional programming, unique pointers, or uniqueness, is a pointer restriction scheme based on a very simple idea: *a unique object is an object to which there is only one pointer in the entire system*. A perhaps more descriptive name would be *uniquely referenced object*, but we chose to follow convention. A unique pointer is an unaliased pointer, *i.e.,* points to a unique object.

The principle underlying a uniqueness system is equally simple, any variable or field annotated with the keyword `unique` contains the single reference in the system to an object or `null` (Hogg 1991; Minsky 1996; Boyland

2001a; Boyland, Noble, and Retert 2001; Boyapati and Rinard 2001; Aldrich, Kostadinov, and Chambers 2002).

Uniqueness can be used to aid reasoning, see for example its use in enforcing software protocols in Deline and Fähdrich's Vault (2001) and Fugue (2003). In Fugue every class corresponds to a state-machine, methods trigger state transitions and a static type system verifies that methods are not invoked unless the object is in the corresponding state. Without uniqueness, such a system would not work since state transitions could be triggered via aliases unaware to the static checker. In general, there is no way that one location could be made aware of a state transition induced at some other location; the first location would still believe the object to be in the previous state.

In addition to aiding reasoning, unique pointers have other virtues that make them useful to encode certain applications. For example, unique pointers enable essential idioms in concurrent programming such as the transfer of ownership pattern (Lea 1998) and also make alias-free initialisation possible. Some widely used interface definition languages such as Microsoft's MIDL even contain a `unique` keyword attribute to specify unique pointers, with similar semantics.

Motivating example

Concretely, the following five lines of code neatly illustrate the problem with aliasing and the benefits of a unique pointer:

```
void example(File file1, File file2)
{
  file1.close(); // puts file1 in closed state
  file2.read();  // requires file2 to be in open state
}
```

If the variables `file1` and `file2` could be aliases for the same object, for example if `example` was invoked like this: `example(file,file)`. In the code above, even if the origin of the file variables is a little clearer, statically determining whether this piece of code is sensible is tricky.

If one or both of the file variables contained a unique pointer, determining that the variables are not aliases is trivial. Thus, we can infer that `file1.close()` will not close the file pointed to by `file2`.

Trivially, object creation (modulo aliases introduced via constructors which we will address later) results in a unique object. Maintaining the uniqueness of an object, *i.e.,* making sure that no additional aliases to the object is created, is a little trickier and requires additional machinery. Hogg (1991) and Minsky (1996) both make use of a *destructive read*, an atomic operation that

returns the contents of a variable and subsequently updates the variable with `null`. The beauty of destructive reads is its immediate simplicity. On the other hand, a major drawback is the addition of null-pointers to the program, implicitly stored in unique variables after reading their contents, forcing additional checks to avoid null-pointer errors. This phenomenon is called "slipperiness" and will be discussed in more detail later.

As an alternative to destructive reads, Boyland (2001a) proposes *alias burying*, basically a technique that guarantees the *effective uniqueness* of a unique variable. When a unique variable is read, all aliases to the same value must be "dead", *i.e.,* they must inaccessible to active parts of the code. For example, when returning a unique local variable from a method there is no need to nullify the local variable, since it will become inaccessible immediately after the return and therefore not invalidate uniqueness.

1.2.1 Uniqueness and Object-oriented Programming

Uniqueness in combination with object-orientation is powerful. However, due to the inflexible nature of classes and objects not being flat entities, but rather aggregates, extant implementations of uniqueness in object-oriented systems suffer from some inherent problems which we now present briefly.

An Abstraction Problem

Extant implementations of uniqueness in object-oriented settings suffer from an abstraction problem first identified by Clarke and the author (2003a).

The problem is due to the presence of a `this` (or `self`) pointer in methods. When invoked on a unique reference, `this` will hold a unique value and thus, assigning `this` or passing `this` as an argument (including as an implicit receiver argument) must invalidate any external pointer to the object in order to keep a strong notion of uniqueness. Thus, invoking a method on a unique variable might consume its contents, meaning that the nature of the invoked method (whether or not it assigns or passes its receiver argument) must be tracked in order to produce a sound system. It is the machinery used to track the *subjective* treatment that breaks abstraction—programs that use unique pointers are forced to change their interfaces as a result of *purely internal changes* to a class' implementation. This causes changes to propagate through a program which makes software evolution and maintenance increasingly complex. In conclusion, unique objects are hence not the black boxes that we want them to be.

Raising the Level of Abstraction

Consider a list object that implements a linked list. The list object is an aggregate constructed from links that hold references to a data object and to its next link in the chain. Conceptually, we regard the list aggregate as a single entity.

Uniqueness is a *shallow property*; a pointer to the list aggregate does not in any way effect the list's transitive representation. While its presence precludes any aliasing of the list object, there is nothing to prevent its representation, the links, from being aliased (not necessarily a bad thing, if it is intentional), neither from inside the aggregate nor from outside the aggregate.

As the increasing complexity of software makes it less useful to talk about single objects instead of aggregates, we believe that uniqueness must be combined with a strong notion of aggregate. Because of its shallowness, uniqueness in itself is not enough to enable unique pointers to aggregates unless the representation objects are all uniquely referenced which might not be desirable let alone possible if the design requires internal sharing.

We believe that uniqueness should reflect how an object is used externally. In Minsky's (1996) version of uniqueness and subsequent proposals based on it, whether an object can be uniquely referenced or not is controlled by the object (or rather by its class). In Hogg's (1991) and Boyland's (2001a) respective proposals, an object can always be referenced uniquely, but certain methods cannot be invoked if we want to keep the unique receiver to the object.

Moreover, if uniqueness was truly how an object is viewed externally, *an aggregate's internal sharing should be able to involve the uniquely referenced object*, *i.e.,* it should be possible to have internal back-pointers to an object that is externally viewed as unique. This is not possible in any extant proposal, as it turns out, for the same reason that leads to the abstraction problem described above. In systems following Minsky's proposal it is prevented statically (such a program would not compile); in systems following Hogg's and Boyland's proposals the creation of such an internal reference would require that the unique external reference is given up, possibly detaching the object from the object graph.

Fusing the statements of this and the previous section, *we want unique references to be able to reference black boxes, regardless of any internal aliasing inside the aggregate. Under the current approaches, this simply is not possible.*

Unnecessary Restrictions Impose Additional Complexity

As we will detail in later sections, unique pointers are slippery in extant implementations, meaning that invocations of methods on unique variables may consume their contents and that unique pointers passed as arguments will also be consumed by the receiving method, regardless of its desired semantics. Thus, unique pointers that are only to be used temporarily must be explicitly returned and reinstated. This is complex and at the very least cumbersome, especially in the presence of several unique arguments and a unique receiver.

To this end, several proposals for unique pointers include a mechanism for temporarily *borrowing* a unique pointer to a method. The method must not make any static aliases of any borrowed pointers that are still accessible when the method exits and at the end of a borrowing method, all borrowed values are reinstated. This way, the uniqueness invariant is preserved between method invocations.

However, due to the complex nature of statically tracing pointers, the restrictions on the borrowed pointers become unnecessarily hard. Basically, borrowed pointers may not be stored on the heap regardless of the lifetime of the object storing them. This makes them a third class of pointers that are neither unique nor non-unique. This makes systems using borrowing less clean and therefore more complex.

In this thesis, we propose a collection of orthogonal concepts for our externally unique pointers that lift these restrictions. The result is a language that is both cleaner and more powerful.

To summarise, we identify three problems with uniqueness: the breaking of the principle of abstraction, the problem with being a shallow property that does not view uniqueness as an external attribute and the presence of third-class, non-orthogonal entities that complicates programming languages and impose unnecessary restrictions. In addition to relieving the unnecessary restrictions that further complicates programming with unique pointers, we want uniqueness to have the three characteristics shown in Figure 1.1.

1.3 Contributions

This thesis makes the following contributions (roughly in order of appearance):

1. The design of a notion of uniqueness that is a property of how the object is referenced externally and not a property of the object itself that enables pointers to black-box aggregates and overcomes the abstraction problem.

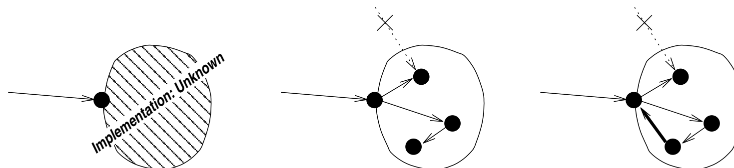


Fig. 1.1. Three characteristics not present in uniqueness. Leftmost: reference to a black box (abstraction). Center: the black box should not be just a shallow object and therefore, the external alias denoted by the dotted line should not be allowed (strong notion of aggregate). Rightmost: the aggregate should be able to have arbitrary internal aliasing, even of the "bridge object", *i.e.,* the entry point (denoted by the slightly thicker pointer), and these references should be ordinary references.

2. The design of orthogonal language constructs to support programming with the abovementioned unique pointers. These language constructs include a few novelties such as generational ownership and a borrowing construct that avoids third-class pointers.
3. A static semantics of the proposed constructs to guarantee proper usage of the constructs.
4. A formalisation of our system via a novel operational semantics inspired by Trees with pointers (Cardelli, Gardner, and Ghelli 2003).

In addition, we show several applications for external uniqueness, that cannot be encoded by traditional uniqueness, such as a unique pointer to the set of links in a doubly-linked list.

Also, by basing our solution on Clarke, Noble and Potter's (1998) ownership types (indeed, the solution relies on ownership types to work), we also make an important contribution to the theory of ownership types in that we enable transfer of ownership which allows many previously impossible programs to be encoded. We also introduce generational ownership, in which blocks or frames may act as temporary owners for objects.

1.4 Outline

This thesis is organised as follows: Chapter 2 presents our problem, the problem background and some of the technical preliminaries necessary for the understanding of the chapters to come. In particular, it introduces uniqueness and the abstraction problem mentioned above. It also discusses alias encapsulation, and shallow and deep ownership types.

Chapter 3 explains ownership types is additional detail, since ownership types is at the heart of our proposal. Both shallow ownership and deep ownership are explained and we show why shallow ownership cannot be used as

a basis for external uniqueness. We also introduce some necessary ownership terminology that we use in later chapters.

Chapter 4 introduces our proposal—external uniqueness. It describes how external uniqueness differs from deep ownership types, the properties it enables and what additional machinery it requires to be maintained.

Chapter 5 introduces Joline, a class-based Java-like language that we use to formalise our ideas. The chapter includes Joline's static and dynamic semantics as well as statements of the most important theorems for external uniqueness and proof sketches.

Chapter 6 shows applications for external uniqueness, in particular some that cannot be dealt with using traditional uniqueness.

Chapter 7 discusses the applicability of external uniqueness to object-orientation, show how external uniqueness overcomes the abstraction problem, and why related approaches do not. It also discusses various side effects our our proposal, such as localised reentrancy and elimination of unnecessary constraints resulting in a more powerful and cleaner language. It also contains a discussion of external uniqueness in the presence of multiple threads, weaknesses and limitations and how external uniqueness can help simplifying reasoning.

Chapter 8 presents related work and compares and relates similar existing approaches with external uniqueness and each other.

Chapter 9 summarises and criticises our findings, as well as gives directions for future work.

❖ 2

Aliasing, Uniqueness and Object-Oriented Programming

This section presents our problem together with the background necessary to understand it. It introduces and outlines aliasing, general problems due to aliasing and various approaches for dealing with such problems. It introduces uniqueness in additional detail, in particular slipperiness, borrowing and how to maintain a strong uniqueness invariant in non-sequential programs with shared state. It then details the *abstraction problem* inherent in extant uniqueness proposals and presents a problem analysis that stresses the need to be able to distinguish between the internals and externals of an object.

2.1 Coping with Aliasing

The sharing of objects is fundamental to object-oriented programming. Indeed, Noble, Vitek and Potter (1998) go as far as saying that aliasing (sharing) is *endemic* in object-oriented programming. Feature, flaw or bit of both, it is a necessary component in widely used patterns and data structures and is also a requirement if we want the structures of object-oriented programs to model the real world.

Arguably, the mere existence of two or more pointers to the same object does not constitute an aliasing problem. Rather, problems occur when the pointers are treated (or viewed) inconsistently. A good example can be stolen from the Geneva Convention on the Treatment of Object Aliasing (Hogg, Lea, Wills, de Champeaux, and Holt 1992). Consider the simple Hoare formula:

```
{x=true} y:=false; {x=true}
```

If `x` and `y` are aliased, this formula is not valid. Sadly, proving that `x` and `y` are not aliases is generally not straightforward as we have previously stated. Although not possible in most object-oriented programming languages, since

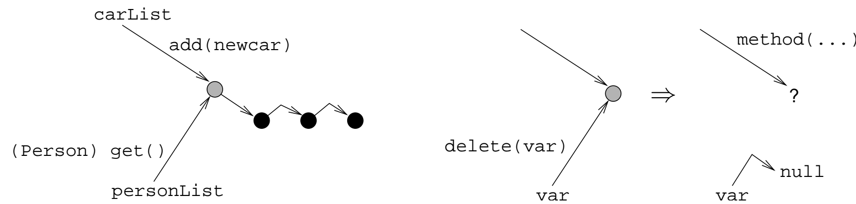


Fig. 2.1. Leftmost picture: Rôle confusion. Rightmost picture: Creating a dangling pointer due to invalid treatment of the `var` reference.

variable names generally cannot be aliases for each other, this example clearly shows the problem with aliasing (just consider `x.f` and `y.f`).

Another trivial but not entirely uncommon example of this is treating an aliased pointer as unique and delete its referenced object causing dangling pointers in other parts of the program. In the presence of garbage collection, similar situations could lead to a memory leak since references to an object that should be deleted exist unbeknownst to the programmer precluding deallocation of the object. Other examples of aliasing problems due to inconsistent treatment of pointers include conflicting updates, where several updates induced from several location (or the same location) via different pointers overwrite each other or put the updated object in an inconsistent state; rôle confusion, *i.e.,* when the same object is viewed as having different, perhaps conflicting rôles via different pointers, which might very well lead to the aforementioned situations etc. Figure 2.1 shows two examples of problem situations due to aliasing.

The leftmost picture, albeit perhaps slightly contrived for brevity, shows rôle confusion—a non-generic Java list used to store both cars and persons. This is fine as long as no-one reads from the list, believing the object obtained to be something it is not (*e.g.,* casting a car into a person). In Java, the error will manifest itself as a `ClassCastException`, but this is just the symptom of the problem; the source is the inconsistent view of the list object due to sharing. The rightmost picture shows the creation of a dangling pointer. Deleting the contents of `var`, a dangling pointer is created, causing a failure when invoking `method()` at a later stage. Ultimately, as the examples show, aliasing errors are design errors—either the list should not be shared at all, or the implementation of its clients account for the possibility of mixed elements, etc.

Even though programmers may claim to have learned to live with aliasing, or software defects that stem from aliasing and related problems (Skoglund 2003), when trying to reason about a program (especially formally), aliasing is a severe obstacle.

Attempts at Dealing With Aliasing

The simple example of the two file variables on Page 6 showed that the absence or presence of aliasing can control whether a piece of code is sensible or insensible; if the two variables were aliases, the code makes no sense since it closes the file and then immediately tries to read from it. It also shows how hard it can be to deduce whether two variables are aliases when the contents of both variables are supplied via methods arguments.

```
void example(File file1, File file2)
{
  // file1 == file2 ???
  file1.close();
  file2.read();
}
```

Keeping track of the file variables throughout the program to verify that they are not aliases, or that no other alias to any of them is created some place else that can be used to close the file or violate protocol in some other part of the program, might not be trivial, let alone possible; library code might not be distributed with sources, or the code that initiates the files might not even have been written yet[1]. In the presence of multiple threads, manually tracking pointer manipulations might be even more difficult.

The Geneva Convention on the Treatment of Object Aliasing (Hogg, Lea, Wills, de Champeaux, and Holt 1992) states that aliasing is a problem in both formal verification and practical programming. It outlines several approaches to dealing with aliasing, namely *detection*, *advertisement*, *prevention* and *control*. We now describe these four approaches briefly. See The Geneva Convention on the Treatment of Object Aliasing for the full-blown discussion.

Alias detection — tries to detect the aliasing patterns of a program, either at compile-time or at run-time. Successful static detection facilitates compiler optimisation and allow formalists to discover cases where aliasing may lead to the invalidation of a program predicate. As is duly noted by Hogg et al., statically detecting aliasing is an NP-hard problem (Landi 1992). Thus, static alias analysis is likely to produce an abundance of may-alias results, which are then forced to be conservatively (and in many cases incorrectly) treated as must-aliases.

[1] Of course, making a note of this non-aliasing requirement into design documents or code documentation might avoid the bug, but there is still the opportunity for human errors, especially since the only documentation of the code that is updated or read during a software's lifetime tends to be the code itself.

Alias advertisement — tries to localise alias information to aid the analysis and make modular analysis possible. Methods can be annotated with keywords describing their aliasing properties, such as whether they create static aliases to an argument or require arguments not to be aliases etc. The program annotations should be enforced by the compiler or run-time system.

Alias prevention — means statically guaranteeing that aliasing will not occur in a particular context. Again, the nature of aliasing will require static constructs to be conservative and perhaps require changes to the ways programs are constructed, in order to obey the rules of the constructs or maintain certain invariants. The non-presence of aliases in a certain context will allow the formalist to prove formulas about the code, and help the programmer avoid writing code that may be rendered insensible due to the presence of aliases (for example, the code snippet with the file variables just shown).

Alias control — takes into consideration the run-time state of the system. It is necessary since there will be situations where aliasing is required in a way that makes static control impossible or impractical. An example of aliasing control is run-time assertions that certain objects are not aliases or that (certain parts of) their representations do not overlap, *e.g.,* Leavens and Antropova's ACL (1999).

In practice, two different approaches to alias management in object-oriented programming exist—*uniqueness or alias-free references* and *alias encapsulation*. Systems with uniqueness are often a mix of advertisement, prevention and control whereas alias encapsulation systems have a tendency to be more statically oriented dealing (almost) only with prevention strategies. We now detail our discussion of those categories, focusing only on the systems immediately relevant to our proposal. Other systems for dealing with, or controlling the effects of, aliasing in object-oriented programming are discussed further in related work, Chapter 8.

2.2 Uniqueness

As stated in Section 1.2, the principle underlying uniqueness is a very simple one: A variable or field with a uniqueness annotation contains the only reference to an object in the system, or it contains null. In the example below, the file stream connected to the file `example.java` is declared unique. Thus, whatever code is executed at `...` in the following code fragment, the

`file` variable will hold the single pointer to the file object when passed as an argument to the `Lexer` constructor.

```
unique File file = new File("./example.java");
... // Arbitrary (but sensible) code
Lexer lex = new Lexer(file);
```

It is quite possible that uniqueness (whether or not enforced by the language in point) of the file stream is desirable in the example above. By only allowing unique pointers to the file object as arguments to the constructor, the implementor of the lexer can be sure that the file stream will not be tampered with by some other object, which makes the lexer easier to implement, avoiding the need for checks and providing easier reasoning about the result of operations on the file stream inside the lexer.

This is the basic realisation of a unique pointer in an object-oriented programming language. Various treatments of unique pointers or unsharable objects have been proposed by Wadler (1990), Hogg (1991), Baker (1995), Minsky (1996), Almeida (1997), Boyland (2001a), Boyland et al. (2001), Boyapati et al. (2001), Aldrich et al. (2002) and Clarke and Wrigstad (2003b).

At a first glance, it might seem that a uniqueness construct is an overly restrictive one. However, an optimistic study by Noble and Potanin (2002) 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. Even though this study is optimistic, since uniqueness violations could occur in between the snapshots and go undetected by the analysis, more fine-grained, less optimistic studies of smaller programs have shown similar results, suggesting that the optimistic results are correct.

2.2.1 Maintaining Uniqueness

In most programming languages object creation results in a unique reference to an object. However, if the object is able to refer to itself in a constructor (for example via `this` in Java), it can create an alias to itself in a constructor argument or a global variable. If that is the case, the reference to the instantiated object returned from the operation will not be unique. As is discussed below, different proposals deal with this problem in different ways.

Moving Unique's Around

Maintaining the uniqueness of a pointer requires some additional machinery. There are two main approaches in the literature: preventing a unique value

```
class List
{
  unique Link head;

  void prepend(Object o)
  {
    unique Link temp = new Link(o); // o is stored in link
    temp.next = head--;         // move head into temp.next
    head = temp--;              // move temp into head
  }
}

class ListClient
{
  unique List l1;

  void add(unique Object o)
  {
    l1.prepend(o);
  }
}
```

Fig. 2.2. Actual uniqueness via destructive reads. The unique links in the list are moved using destructive reads.

from being aliased at all (actual uniqueness), or preventing the existence of two or more *visible* aliases to the same value (effective uniqueness).

Hogg (1991) uses *destructive reads* to implement an actual uniqueness scheme for his Islands, *i.e.,* reading the value of a unique variable implicitly updates the variable with `null`. Another way of maintaining actual uniqueness is by always returning a copy when reading a unique value. A perhaps more drastic approach is to replace the assignment operator with a "swapping operator" (Harms and Weide 1991), that swaps the values of two variables. However, neither copying nor swapping alone works well with method invocation if an implicit alias is created for the method to be able to refer to its receiver which is common-place in object-oriented systems. Other systems that use destructive reads include Minsky's Eiffel* (1996) and the Joline system described in this thesis.

Destructive reads is a simple and intuitive approach to maintaining the uniqueness of a value. On the downside, it has some undesirable side-effects, such as methods (implicitly) consuming values of argument objects as is discussed in the upcoming section on borrowing on Page 15. As argued by Boyland (2001a), destructive reads may make a method appear to have more side-effects than it conceptually performs. Thus, in spite of its elegant and simple semantics, destructive reads may increase the complexity of a program,

```
class List
{
  unique Link head;

  synchronized prepend(Object o) anonymous
  {
    unique Link temp = new Link(o); // o is stored in link
    temp.next = head;           // no need to nullify head
    head = temp;                // since it will immediately
                                // be overwritten
  }
}

class ListClient
{
  unique List l1;

  synchronized add(Object o)
  {
    l1.prepend(o); // l1 cannot be used simultaneously
                   // because of synchronization
  }
}
```

Fig. 2.3. Effective uniqueness via alias burying. Synchronisation and *anonymous methods* (see text) are used to maintain uniqueness.

especially if no extra constructs are added to ease the pain of programming with them, due to the implicit addition of null-pointers to the program and the inherent slipperiness of uniques (reading them consumes them).

Figure 2.2 shows a simple list implementation and a client class. The list contains a set of uniquely referenced links. When prepending, the contents of `head` is moved into the new link, which is then moved into the `head` field. Both moves uses destructive reads, indicated by `--`.

An equally powerful alternative to actual uniqueness is *effective uniqueness*. The key of achieving effective uniqueness is to verify that two aliases to the same "unique" value are never visible/used simultaneously. An example of such a system is Boyland's alias burying (2001a), essentially a lightweight live variable analysis that requires all aliases to a unique variable to be "dead" when that variable is read. For example, if it can be determined that a variable will not be accessed again before it becomes invalidated (for example, goes permanently out of scope) or updated (overwriting the value that breaks uniqueness), it does not matter if it contains an alias to a unique object since it will never be used in a way that can witness the violation. To enable modular checking, some form of effect annotations are necessary to indicate which

unique fields are read by a method (Boyland 2001b). To use alias burying in a multi-threaded setting, locking objects before unique values are accessed is necessary since there is no destructive read. Other systems that use a similar approach to maintain uniqueness include Fugue (DeLine and Fähndrich 2003), Parameterised Race-Free Java (Boyapati and Rinard 2001) and Alias-Java (Aldrich, Kostadinov, and Chambers 2002).

Figure 2.3 revisits the same example of list and client as was used to illustrate destructive reads. To maintain uniqueness, `prepend()` is annotated as *anonymous*, meaning (roughly) that it may not use `this`[2]—even though there may be other aliases to the contents of `l1`, these are not used (*i.e.,* effective uniqueness). The `add()` method is synchronized to prevent simultaneous access to the unique `l1` variable. More complex programs might require that `l1` was copied to a stack variable and subsequently updated with *null*, even in the case of alias burying, to prevent reentrant methods from reading `l1` and thus break the uniqueness invariant.

In the `prepend()` method, there is no use for destructive reads or nullifications, since the method is synchronized (no other thread can access `head`) and both the contents of `head` and `temp` are either immediately overwritten or invalidated after being used on the left hand side of an assignment, meaning that their respective contents are never used to invalidate effective uniqueness.

In addition to actual uniqueness and effective uniqueness, we distinguish between week and strong forms of uniqueness:

Definition 2.1. *Strong uniqueness invariant* — A strong uniqueness invariant never allows uniqueness to be compromised. At any time, there may be only one pointer to a unique object anywhere in the program, including contents of variables on the stack.

Definition 2.2. *Weak uniqueness invariant* — A weak uniqueness invariant allows uniqueness to be slightly compromised. A pointer may still be regarded as unique, even if there are *stack based* aliases to it.

Borrowing and the Case Against Slipperiness

Uniqueness can be used to tackle some of the aforementioned aliasing problems. Knowing that an object is uniquely referenced, it can safely be deleted without the risk of dangling pointers and there is no risk of rôle confusion. An object's class may even be changed if possible (see *e.g.,* Fickle (2001), or

[2] An anonymous method may not store `this`, pass `this` as an argument or use `this` as a receiver of non-anonymous methods.

later in this thesis for a change of type) without risking inconsistencies due to several conflicting views of the same object.

Consider the following method, a revisit to our previous example with the two file variables[3]:

```
void example(unique File file1, unique File file2)
{
  file1.close();
  file2.read();
}
```

Now, because of the `unique` annotations in the method head, the variables cannot be aliases. (Naturally, only one of the arguments need to be a unique pointer to preclude aliasing in this particular example.) On the downside, an invisible side effect of the method above is that it consumes the pointers to both files in order to maintain the uniqueness invariant. In many cases, this is undesirable, *e.g.,* if the method only wishes to use the file pointer for the duration of its execution (*i.e.,* borrow it) to perform some operations and not create a static alias to it that will be reachable after the method returns. To battle unwanted consuming, the method must explicitly return any non-captured unique value to be manually reinstated at the call-site, which, especially if the programming languages does not allow multiple return values (*e.g.,* Java), is cumbersome and prone to errors (Baker 1995).

To this end, many systems introduce a *borrowed* or *lent* pointer that enables automatic reinstating of non-captured unique arguments and receivers. Concretely, borrowed pointers are temporary stack-based aliases that never move to the heap. By this constraint, as soon as the method that borrowed the pointer returns, all the stack-based aliases become invalid, and thus, the borrowed value is again unique. The borrowed pointer may be treated as unique or not, depending on the implementation (for example, if the content of the original variable is invalidated during the borrowing, the borrowed pointer will be unique). The only requirement is that once the method returns, all aliases must be invalidated, which in all extant systems require that it must never be stored on the heap.

Depending on its implementation, borrowing might have different effects on the uniqueness invariant. If the original value is unreachable during the borrowing, we retain a strong uniqueness invariant. This might be the case if only borrowing of stack-based variables is allowed, unless for example the

[3] On a side-note, ACL (Leavens and Antropova 1999), uses dynamic dispatch to solve this problem—the programmer provides two separate implementations of methods, one where the arguments may not be aliases, and one where they may. Upon invocation, a simple identity comparison between the arguments is carried out and method selection is based on this result.

same variable may be borrowed twice, or the invoked method runs in another thread and thus returns immediately etc. If a unique value may have several simultaneous borrowings, or if the original value is reachable during the existence of the borrowed aliases, uniqueness is broken, and we get a weak uniqueness invariant consistent with Definition 2.2.

A downside of borrowing in a language is that borrowed objects become second-class citizens: a borrowed object may be treated as an ordinary (shared or unique) value in any way, but it may not be stored on the heap. This restriction is unnecessary to maintain soundness, but is imposed because the underlying type systems are not strong enough to allow it. The introduction of a third pointer category (ordinary/shared, unique and borrowed) makes the programming language less clean and therefore more complicated.

Treating the Self

The presence of a `this` pointer (or equivalent) increases the complexity when adding unique references to an object-oriented programming language since one must consider how a class treats its instances internally. For example, if a method assigns `this` to a variable or field, invoking the method on a unique variable should consume the variable's contents to maintain a strong uniqueness invariant[4]. In addition, if a constructor saves `this` in a global field or in a field of an argument object (or in a subobject to itself), the new operation invoking the constructor will not return a unique reference.

Approaches in the literature reflect the treatment of `this` in a class' interface in one of two ways:

via class annotation — classes are divided into two kinds, those whose instances may assign `this` internally, and those whose instances may not. Only instances of the latter may be referenced uniquely (Minsky 1996).

via method annotation — methods are annotated to indicate that they may consume `this` (Hogg 1991; Boyland 2001a). 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 this creates a problem with abstraction.

[4] In the presence of borrowed pointers, methods that are receiver-consuming, *i.e.,* stores a reference to `this`, must not be invoked on borrowed values, introducing additional complexity.

```
unique class Server extends Object
{
  int no_connections = 0;

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

  int getConnections()
  {                            // Good method
    return this.no_connections;
  }
}
```

Fig. 2.4. Using class-level annotations to control how `this` is treated internally.

Class-level Annotations

Class-level uniqueness annotations, proposed by Minsky (1996) in Eiffel*, decorates class declarations and controls whether instances of a particular class can or cannot be uniquely referenced.

In the example in Figure 2.4, a class `Server` is annotated with the uniqueness keyword allowing its instances to be uniquely referenced. The unique annotation requires that no method in the server class assigns `this`, or passes `this` as an argument to a method. Methods can be invoked on `this`, since any such method will obey the same rules by virtue of the class-level annotations.

A drawback of this solution is that whether or not uniqueness is possible becomes a property of the class and neither how the object is viewed (externally or internally), nor even a property of the object. It is also quite inflexible, since consuming and non-consuming methods cannot be mixed in the same object.

An additional drawback is that instances of classes not annotated with the `unique` keyword cannot be uniquely referenced, even if the class' implementation would allow it. Also, the uniqueness property must be preserved through subclassing which makes extension via subclassing harder or less powerful since the annotation of the superclass must be respected by all subclasses. Otherwise, if a unique class was extended by a non-unique class, subsumption would allow an easy circumvention of uniqueness (just cast the non-unique to the unique super class and invoke a method) unless we only use static binding.

```
class Server extends Object
{
  int no_connections = 0;

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

  int getConnections() borrows
  {
    return this.no_connections;
  }
}
```

Fig. 2.5. Using method-level annotation to control subjective treatment of `this`

Method-level Annotations

Method-level annotations, used by Hogg (1991) and Boyland (2001a) offer a more flexible solution than the class-level approach above. With method-level annotations, an object can be uniquely referenced regardless of its class' implementation. Rather than classes, methods are annotated to reflect how `this` is treated. Using `consumes` and `borrows` keywords, similar to those used by Boyland, Figure 2.5 shows the `Server` class from Figure 2.4 using method-level annotations.

The `getConnection()` method is now annotated with the `borrows` keyword, meaning that it does not store `this` on the heap (which can be controlled by a simple compile-time check). The `connect()` method is annotated with `consumes` meaning that it will consume its receiver object if invoked on a unique pointer (depending on the implementation, it might not be possible to invoke it on a non-unique pointer).

Method-level annotations allow mixing of consuming and non-consuming methods in the same class. As with class-level annotations, some additional constructs are required to make it work, either we need destructive reads to ensure the nullification of the variables used to invoke a consuming method, or we need something like alias burying (Boyland 2001a) to make sure that uniqueness is maintained (or we can simply weaken the uniqueness invariant if sensible).

Concluding remarks

Any of the two approaches above can be used to deal with constructors and when they do or do not return unique references. With class-level annotations, if all constructors of a class assign `this`, instances of the class may

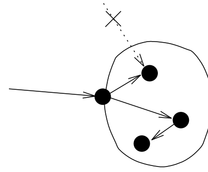


Fig. 2.6. Revisiting Figure 1.1, desirable properties of uniques. The unique reference should be the single entry-point to an entire aggregate, not just the bridge object.

never be unique. The method-level approach allows for some more flexibility; either the constructor will return `null` (which may be argued to be counter-intuitive), or constructors may not consume `this` at all, whereas normal methods may.

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.

2.3 Inherent Problems with Uniqueness

We now detail the description of the problems with uniqueness. We do not revisit the issue of non-orthogonal concepts, since that problem was recently discussed (see Page 2.2.1 and following), or the unnecessary restrictions on borrowed references turning them into second-class citizens and further complicating the language.

2.3.1 Unique Pointers to Aggregates

As previously stated, the shallowness of uniqueness weakens the usefulness of unique pointers since the uniqueness only applies to the referenced object, and there is nothing to prevent a representation object of the unique from being multiply referenced.

If uniqueness is used to avoid unnecessary synchronisation, moving a unique object from one thread to another might result in the representation objects becoming shared between threads, meaning that synchronisation is still necessary with the opportunity for deadlocks etc. It would appear that uniqueness is less useful as objects becomes more complex.

Figure 2.6 shows the “aggregate uniqueness” that we would like, especially if the objects in the aggregate can reference objects external to the aggregate (*i.e.,* outgoing references are allowed). Clearly, a strong notion of aggregate carefully integrated into the definition of uniqueness is necessary to support this.

2.3.2 Uniqueness and a Problem with Abstraction

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;
```

When we change the implementation of the `BlackBox` class so that the `xyzzy()` method assigns `this` internally, the existing proposals force us to change `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.

With Class Annotations

Since treatment of `this` is shown in the class declaration, we run into trouble when our treatment of `this` changes.

Using class annotations `BlackBox` would have been annotated `unique` to show that its instances can be uniquely referenced. After the changes, we would have to modify `BlackBox` to indicate that its instances cannot be referred to uniquely, *e.g.,* `neverunique class BlackBox`. As a consequence, all variable declarations of type `unique BlackBox`, *e.g.,* such as `unique BlackBox bb;` above would no longer be valid in the program, and must have their uniqueness stripped in order to compile. Depending on the system, 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 implemented manually. Obviously, these changes could propagate through the entire program.

With Method Annotations

Since treatment of `this` is shown in each method declaration, we run into trouble when our treatment of `this` changes.

When using method annotations, we would have to modify the `xyzzy()` method to indicate that it now consumes `this`, such as `void xyzzy() consumes { ... }`. Although forcing potentially fewer propagating changes in the

code than if we were using class-level annotations since `BlackBox` instances may still be uniquely referenced in the program, the call `bb.xyzzy()` may create an internal reference to its target, requiring that the `bb` variable be consumed to preserve uniqueness. Apart from being akward and unintuitive, the consequence here is even more drastic, as the semantics of method call changes: calls to this method suddenly consume their target, whereas in the original program they did not.

Concluding remarks

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

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

Thus extant uniqueness proposals break *abstraction*.

2.3.3 Problem Summary

We have described three problems:

1. Extant uniqueness proposals break abstraction making the task of maintaining and updating software more complex and fragile. How an object treats itself internally is visible in the interface causing purely internal changes to an object's implementation to trigger interface changes.
2. The usefulness of a unique object is hampered by the fact that an object's internals are not properly encapsulated. For example, moving a unique object to a different thread might cause the object's representation to be split between several threads which is likely to be opposite to the desired semantics. We desire unique references to *aggregates*—not just to flat objects.
3. To relieve the pain of programming with unique pointers, extant proposals use borrowing. Extant borrowing constructs suffer from unnecessary restrictions and further complicate the language by adding an additional, third-class pointer to the programming language.

We now present an analysis of these problems and a solution.

2.4 Problem Analysis: Distinguishing Between Internal and External References

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

A purely internal reference to an object which has only one external reference cannot be used by objects other than the holder of the external reference. This means that no changes to the object via the internal reference which would violate the "uniqueness" of the external reference can be triggered by any object other than the one holding the externally unique reference. In a sense, purely internal references are innocuous and their existence should not affect how an aggregate is viewed externally. Instead, they ought to be preserved to maintain the internal consistency of an aggregate. Otherwise, knowledge of internal reference behaviour exposes an object's implementation details and thus violates abstraction, as we have shown.

Fortunately, the desired distinction can be made in a programming language with Ownership Types, as originally proposed by Clarke, Potter and Noble (1998). This form of ownership types provides strong protection against external aliasing of an object's internals, enabling a strong notion of aggregate object. Thus, it seems that our issue with uniqueness being a shallow property will be addressed by bringing in ownership types as well.

Relying on ownership types, we propose external uniqueness, which allows innocuous internal pointers and is effectively unique in a sense similar to alias burying. But first, let's have a look at the second approach to alias management, alias encapsulation, and in particular ownership types, since we rely heavily on ownership types to make our system work.

❖ 3

Ownership Types

This section details ownership types, a particular form of alias encapsulation, the other main approach to alias management in object-oriented programming languages (the first was uniqueness). A more general discussion of alias encapsulation is deferred to Chapter 8, Related Work, in order to cut to the chase a little quicker.

We first describe shallow ownership and its drawbacks. While deep ownership is an ancestor to shallow ownership, we present them in the inverse order for pedagogic reasons. Having described shallow ownership, we describe deep ownership and its famed owners-as-dominators property. The primary goal of this chapter is to bring the reader up to speed on ownership types, enough to understand our extension to implement external uniqueness later.

3.1 What is Alias Encapsulation, Anyway?

Name-based encapsulation, present in, for example, C++ (Ellis and Stroustrup 1990) and Java (Gosling, Joy, and Steele 1996), hides names of variables and methods annotated `private`. Private variables and methods are not visible outside the object and are only accessible via `this`. However, there is nothing to prevent the contents of a private field from being exported by a public method and then modified externally. When formulating class invariants, and making sure they are preserved, not only the methods of a class need be considered, but also all possible manipulation of all aggregate objects, even if these objects are stored in private fields.

Rather than just preventing the *names* of fields storing aggregate objects from being used external to the object, alias encapsulation imposes a form of object-level privacy by preventing *objects* from being accessed outside of their enclosing encapsulation boundaries (Hogg 1991; Almeida 1997; Clarke, Potter, and Noble 1998; Müller and Poetzsch-Heffter 1999; Clarke

2001; Boyapati and Rinard 2001; Boyapati, Lee, and Rinard 2002; Clarke and Drossopolou 2002; Aldrich, Kostadinov, and Chambers 2002; Banerjee and Naumann 2002). Typically, aliasing protection schemes impose some kind of restriction on the flow of references to achieve a stronger encapsulation than is given by just restricting access to names. In particular, references to internal objects may escape their owners, but not to objects outside a given protection domain. We defer the general discussion on alias encapsulation and its usefulness to Chapter 8 and go straight for ownership types.

3.2 Ownership Types and Insides and Outsides of Objects

In ownership types objects have owners and can be owners of other objects. The objects owned by object a are called a's representation. The key idea behind ownership types is to prevent representation objects from being exposed outside of the object that owns them[1]. While external objects are may not read fields or invoke methods that export references to representation objects, the object itself may pass its representation to trusted objects. Depending on the definition of trusted objects, different levels of encapsulation can be achieved.

Shallow ownership lets the programmer decide which objects are trusted and which are not. This is a powerful and flexible solution, but presents a problem since there is nothing to prevent a trusted object being passed a reference to pass that reference on to an untrusted one (this will be explained in more detail shortly). The trusted object need not do so with malicious intent, but might simply be unaware of the desires of the first object (since the system lacks the mechanisms to express them). *Deep ownership* avoids this with an even more powerful system by recording nesting relations between objects and only allowing internal objects to be trusted. Thus, representation can only be exported to (transitive) representation.

When the nesting relations of deep ownership are in place, it can be determined whether an object is internal to another just by inspecting their respective types. This makes it possible to distinguish between the internal and external of an object. Internal references that never crosses this boundary (*i.e.,* never flow to an external object) are *purely internal* in the sense we described earlier. We have argued that such purely internal references should be viewed as part of an aggregate's implementation and not preclude the existence of an external, unique references to an object.

[1] The problem of representation exposure is well known, see *e.g.,* Abstraction and Specification in Program Development (Liskov and Guttag 1986) or "Wrestling With Rep Exposure" (Detlefs, Leino, and Nelson 1998).

A full-blown theory of ownership types is described in Clarke's dissertation (2001). It also contains a more in-depth comparison between various alias management schemes than what is presented here. For ownership types systems presented in the context of Java-like languages, see *e.g.,* Clarke's and Drossopoulou's `Joe`$_1$ paper (2002) or Boyapati et al. Parameterised Race-Free Java (2002) or Boyapati's dissertation (2004).

3.2.1 Shallow Ownership

Shallow ownership prevents direct access to an object. Again, name-based encapsulation, *e.g.,* Java's `private`, `public` and `protected` annotations on variables and methods, is easily circumvented by just returning a reference to a private object in a public method (or indeed, the private object may be inserted via a public method and external references kept)[2].

In shallow ownership, all objects have an associated owner that is some other object. The owner is the necessary permission to reference the object and an object must be given an explicit permission to reference an object that is not part of its representation. Classes are parameterised and types are created by instantiating the owner parameters of a class with actual objects—giving the instance of the type permission to reference any objects owned by the parameters. The first parameter is written before the class name and denotes the owner; subsequent parameters are permissions.

Shallow ownership is per object instead of per class, since what other objects the object may reference is specified independently for each instance.

With shallow ownership, an object must be given an explicit permission to reference another object (or instantiate it itself). Figure 3.1 shows an implementation of a simple linked list using shallow ownership as present in AliasJava (Aldrich, Kostadinov, and Chambers 2002). Variables annotated `this` hold references to representation objects[3]. Figure 3.2 shows what instances have the necessary permissions to reference what other instances.

However, as there are no relationships between permissions, the encapsulation of shallow ownership may be broken by escaping proxy objects which access the encapsulated objects. This is shown in Figure 3.3. Since there is no information in the `Proxy` class whether or not objects owned by `owner1` can

[2] Stronger forms of name-based encapsulation exist, for example Eiffel's (Meyer 1992) ability to only allow access/invocation of a feature to/from certain classes. This encapsulation is stronger than Java's, but lacks the precision of ownership types. For an enlightening text on Eiffel, Java and C++ that compares *e.g.,* the encapsulation and access control of the different languages, see Joyner (1999).

[3] AliasJava uses the keyword `owned` instead of `this`, and also place the owner parameter inside the < and > bracets. We alter their syntax slightly to match ours for uniformity.

```
class List<dataowner>
{
  this:Link<dataowner,this> head;
}

class Link<dataowner,linkowner>
{
  dataowner:Object data;
  linkowner:Link<dataowner,linkowner> next;
}
```

Fig. 3.1. Example of shallow ownership. Classes and types are parameterised with permissions (owners) to reference certain objects. The identifiers `dataowner` and `linkowner` are owners of the list data and the links in the list respectively. The owner of an object is the permission necessary to reference the object.

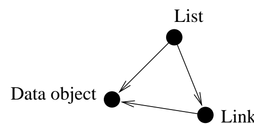


Fig. 3.2. Reference permissions for the code snippet in Figure 3.1. Arrow means "may reference", so `List` instances may reference both links and data objects. All objects have the right to reference themselves, *i.e.,* self-referential arrows are hidden.

reference objects owned by `owner2` or vice versa, the code of the `expose()` method is valid, and thus, any object with permission to reference an object can create a proxy that breaks the encapsulation. To relate to the example above, it would be possible to give objects external to the list, for example the data objects, permission to reference individual links in the list, which clearly should not be possible since the links belong to the list's representation. As it makes the type system very flexible, it is possible to view this as a feature.

Regardless of whether it is viewed as a feature or a flaw, the possibility of `Proxy` objects makes the encapsulation of shallow ownership intransitive, and, as we shall see, the invariants of shallow ownership are not as strong as in deep ownership. In particular, they are not strong enough to support our proposal since variables owned ty `this` cannot be viewed as purely internal for the reasons leading to the situation in Figure 3.3. Thus, we refrain from discussing shallow ownership at length, and instead concentrate on deep ownership in the next section.

Examples of shallow ownership include Aldrich et al.'s (2002) AliasJava and Boyland et al.'s (2001) capabilities for sharing. Equally strong forms of (shallow) encapsulation can be found in Vault (DeLine and Fähndrich 2001) and in Leino et al.'s (2002) data groups.

```
class Proxy<owner1, owner2>
{
  owner1:Untrusted<owner2> break;
  owner2:Object o;

  void set(owner1 Object o)
  {
    this.o = o;
  }

  owner1 Untrusted<owner2> expose()
  {
    break.set(o);
    return break;
  }
}
```

Fig. 3.3. Using a proxy object to circumvent protection. The `Proxy` object has permission to reference objects owned by `owner1` and `owner2`, but knows nothing about whether such objects may reference each other.

3.2.2 Deep Ownership

Deep ownership enforces encapsulation, much like shallow ownership, but emposes stronger restrictions on references and also avoids the proxy problem mentioned above. While on the surface, deep ownership looks very similar to its weaker, shallow version, there are fundamental differences in the level of encapsulation achieved.

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 being thought of as *inside* their owning objects and references may be from the inside to the *outside*. Local nesting information, *i.e.,* whether an object is inside another, is propagated through the program to restrict the global structure of the object graph. The stronger encapsulation restrictions of deep ownership cannot be circumvented as is the case for shallow ownership—representation objects cannot be exported outside the owning object[4].

Three components form the basis of an ownership types system, namely:

1. owners and permissions,
2. nesting relations, and
3. a containment invariant.

[4] This restriction might be weakened for temporary, stack-based references. This was presented as an optional feature in Joe_1 (Clarke and Drossopolou 2002) to increase the flexibility to enable iterators and track them in the type system.

The existance of nesting relations is the big difference from shallow ownership—they allow the formulation of a stronger containment invariant, and thus additional restrictions of the object graph. Furthermore, they allow us to distinguish between the outside and inside of an object. As long as we only allow the flow of a reference to objects nested inside it, we know that it will never escape, not even by proxy, and is therefore *purely internal*.

In Figure 3.3, shallow ownership could be broken because there is no knowledge about whether or not certain objects could reference each other. In deep ownership, the implementation of the `expose()` method is only valid if we can deduce that `owner2` is nested inside `owner1`, *i.e.,* if `owner1` already has the necessary permissions (also see the owner-as-dominators property below). To be able to statically determine ownership nesting, headers are extended with relations between owners to thread nesting information through the program:

```
class Proxy<owner1 inside world, owner2 inside owner1>
{
  ...
}
```

When forming types from classes and owners in scope, the nesting requirements of the class header must be satisfied. Thus, the object graph becomes well-constructed with respect to the nesting requirements specified in the classes.

All objects have an owner (either some other object or some system-wide accessible owner constant) and can be the owners of other objects. The ownership relation forms a directed acyclic graph, where an object is inside (written $\prec^*$) the object which owns it (see Page 37 for a picture of an ownership graph that shows nesting of objects). Owners other than objects are possible, *e.g.,* Boyapati et al. (2002) introduce threads as owners to enable thread-local objects. Further on, we too shall make two simple yet powerful additions to the domain of owners to enable external uniqueness.

An additional, system-wide accessible owner called `world` forms the root of the ownership dag, placing all objects inside `world`. As is investigated by Clarke (2001) in his dissertation, multiple roots are possible (*e.g.,* a separate root for each module, or layer in an arhictecture). A single root is however enough for our purposes here.

Realising Ownership in Programming Languages

Realising deep ownership in a class-based object-oriented programming language is not overly easy. Owners become a new syntactic category and are

included in an object's type. Types now take on a form quite similar to generic types in C++ (Ellis and Stroustrup 1990) and Java's anticipated 1.5 "Tiger" release (Austin 2004). Type are parameterised by owners:

$$p_0 : c\langle p_{i\in 1..n}\rangle$$

where p_0 is the owner of all objects of that type, and each p_i for $i \geq 0$ is an ownership parameter, a binding for the parameters of the class, allowing references to outside objects with these owners. As stated above, ownership parameters can be viewed as permissions to reference external objects[5]. Since all permissions are ordered, we can prevent the circumvention possible in shallow ownership by only allowing an object to reference objects external to it (in addition to its own representation). As a consequence, the owner must always be inside the other owner parameters. These important restrictions enable the containment invariant described shortly. Importantly, ownership is decoupled from "has a reference to", which makes the system flexible and powerful.

Within the body of a class, `this` is used as an owner parameter in the types of all objects owned by the current instance, *i.e.,* the "representation objects". These objects are directly inside the current instance and are inaccessible to the outside. Objects inside representation objects are also inside the owner of the representation (*i.e.,* $\prec^*$ is a transitive relation), however inaccessible (*i.e.,* ownership is *not* transitive)—an object's representation may only be accessed by the object itself, or by the objects inside the representation who's types were parameterised with the appropriate permission.

All objects have an owner, denoted `owner` within the class body. The class parameters are named by the programmer to enable capturing semantic interpretations of ownership; for example, the parameter for the owner of data objects in a list could be labelled "listdata". Figure 3.4, shows the code for a simple linked list and links (omitting methods for brevity).

In addition to `world` and owner parameters, additional owners may be introduced via owner polymorphic methods. Owner-polymorphic methods take owners as arguments meaning that they can be given permissions to reference objects that the receiver object would not normally be allowed to reference. This allows the programmer to *temporarily* break the containment invariant of deep ownership for the duration of the method call. As soon as the method terminates all references using permissions received as arguments automati-

[5] As argued by *e.g.,* Noble et al. (1998) and others, allowing pointers to external objects is not always a good thing, since that creates dependencies on externally owned objects, over which there is potentially no control.

cally become invalidated. Owner-introducing constructs will be described in more detail later as we introduce the semantics of our proposal.

As already stated, all owners are inside `world`. Within a class body we have `this` $\prec^*$ `owner`, and `owner` $\prec^*$ α for each of the class' parameters α. These are necessary requirements in order to circumvent the problem in shallow ownership.

For subclasses, an entirely new class header is specified along with a mapping relation from the owners of the subclass to those of the superclass. Thus, the number of owner parameters in a subclass may grow or shrink depending on the relations between the owners in the super class. The owner must however 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.

A Simple Example

The list example in Figure 3.4 demonstrates some of the power of ownership types. The list class is parameterised by an owner for the data that will be contained in the list—this gives the list permission to reference the data objects. This permission is passed on to the links. The head and tail links are owned by the list object itself and are thus part of the list's representation. Thus, the links cannot escape the list boundary (as shown in the picture of Figure 3.5), not even by proxy—storing a reference to a link in a data object is not possible since the data object cannot be given the appropriate permission (since the link's owner is not external to the data object). This precludes the possibility of any external object getting hold of a reference to the individual links. Thus, it becomes impossible for any outside object to manipulate the links other than via the protocol of the list object.

As the links are parameterised by the `listdata` owner, they have permission to reference list data objects and data objects may be stored in and retrieved from the list as long as they have that same owner.

Some Useful Ownership Terminology

Objects with the same owner are called *siblings* or *sibling objects*. In the example in Figure 3.4, all links belonging to the same list are siblings. *External objects* are objects owned by `owner` or some other owner parameter. In the example in Figure 3.4, all data objects are external. *Internal objects* are objects owned by `this` or some other owner inside `this`. In the example in Figure 3.4, the links are internal to the list.

```
class List<listdata outside owner>
{
  this:Link<listdata> head, // remember: this is inside owner
                      tail;
}

class Link<data>
{
  owner:Link<data> next;
  data:Object obj;
}
```

Fig. 3.4. A linked list and links using ownership types

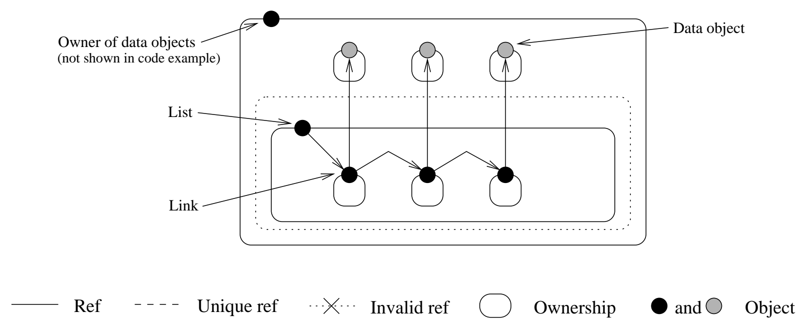


Fig. 3.5. Object graph for the linked list example in Figure 3.4. The dotted line indicates the presence of zero or more boundaries between the list object and the data objects in the list. As is required by the containment invariant, the list object is nested inside the owner of the data objects.

Owners are Dominators

The containment invariant is the condition which governs whether one object can refer to another. It can be stated as:

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

where α and β are objects, $\rightarrow$ means "refers to", $\prec^*$ is the nesting relation "is inside" and $\Rightarrow$ is a standard logical implication.

In plain text, the containment invariant states that an object α can only access an object β whose owner is outside of itself, or alternatively, an object cannot be accessed from outside of its owner. Thus, the owner is effectively, as we have stated, the permission required to access an object, and an object's position in the inside relation determines whether the object has enough permission. A nice theorem (Potter, Noble, and Clarke 1998; Clarke 2001) states that if we have all of these conditions, then an object's owner will be on all paths from the root of the graph to that object, which is to say that an object's

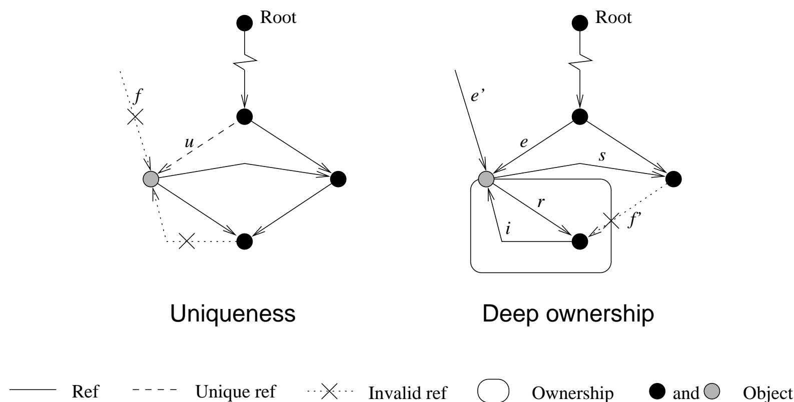


Fig. 3.6. Comparing uniqueness and deep ownership. *Reference kinds*: u – (externally) unique. f, f' – invalid (f breaks uniqueness, f' breaks deep ownership). s – sibling. e, e' – external to grey object. i – internal. r – representation.

owner is its *dominator*. This property is called *owners-as-dominators*, and type systems which enjoy it are said to offer *deep ownership*.

Deep ownership is illustrated in the second picture in Figure 3.6. The objects an object owns are nested inside it, as is depicted by the rounded box. The rounded box is the grey object's *ownership bound* containing all its representation. A third way of explaining the property of deep ownership can be stated as *references cannot pass through an object's boundary from the outside to the inside.*

As owners are dominators, all paths from the root of the program to an object must pass through the owner of that object. This means that if an object is removed, its entire transitive representation becomes inaccessible since all paths to any such object are broken. As is noted by Clarke in his dissertation (2001), and Clarke and Drossopoulou (2002), this could have some positive side effects on garbage collection, somewhat similar to region-based memory management. This remains to be investigated, but see Boyapati et al. (2003) where ownership types is used to elimiate run-time checks to avoid dangling references when deleting regions in real-time Java.

The owners-as-dominators model can be argued too restrictive; for example, in its basic form, it does not allow to objects to share a common representation (an object always have one owner). An example of the consequences of this is that iterators to lists cannot be encoded unless the iterator is part of the list's representation, which is an unrealistic restriction is most cases since not even the owner of the list is allowed to iterate over its contents.

Extensions to deep ownership types have relieved some of the abovementioned requirements, see *e.g.,* Boyapati et al. (2003) for a system that weak-

ens the owners-as-dominators property for objects of inner classes to allow the encoding of iterators. Another extension to ownership types that can be used to perform external initialisation and iterators by allowing temporary, stack-based references to break deep ownership can be found Clarke's and Drossopoulou's Joe_1 (2002).

Ownership and Every-day Programming

Experience with ownership types is growing, though not for large-scale applications. Noble and Potanin (2002) have made analyses of heap structures of a number of Java programs to compare their aliasing structures with the ones imposed by *e.g.,* ownership types. For their ownership types metrics, they have measured the average depth of objects on the heap, *i.e.,* the average levels of encapsulation for an object. The average depth was found to be 5.47. Although it is not immediately clear what that means in practice, it does suggest that there is a significant amount of object encapsulation present in existing programs. It also indicates that the object graph restrictions imposed by ownership types are to large extent compatible with how real programs are structured. The optimistic nature of the study, analysing only a few heap dumps during a program's run might however have skewed the results. While encouraging, finer graned results are probably necessesary.

Several systems have successfully used ownership types to implement other schemes, such as the Parameterised Race-free Java of Boyapati et al. (2002). As ownership types systems give additional benefits in program construction, it might be the case that the structure of programs will change to capitalise on the side-effect benefits of ownership types. To this end, empiric studies of uses of ownership types in non-toy settings are necessary.

In all cases, we feel that it is not presently clear how the containment invariant of deep ownership types interact with the structure of ordinary programs. In a strict sense, ownership types does not impose any restriction since everything can be owned by world, but this would lose all potentional benefits and in the worst case reduce ownership types to a bunch of annoying and worthless annotations. Clearly, ownership types is powerful and applicable for small parts of a program, such as the list example above. Investigating the usability of ownership types for larger applications is clearly a future direction that this research will take.

The integration of uniquness into ownership relieves some constraints that ownership types imposes, even for very small structures. This is discussed at more length in Section 6.1. Such increasing flexibility of ownership types systems might perhaps lead to the fulfilment of Boyapati's vision of ownership types as the basis for future programming paradigms[6].

[6] From Boyapati's presentation at OOPSLA 2002 (Boyapati, Lee, and Rinard 2002).

3.3 Combining Uniqueness and Ownership Types

The marriage of uniqueness and alias encapsulation schemes that offer deep ownership or deep encapsulation (*i.e.,* encapsulation of an entire transitive closure of objects of an aggregate) enables unique references to aggregates in the sense of the middle picture in Figure 1.1. Unique references to aggregates are possible AliasJava (Aldrich, Kostadinov, and Chambers 2002) in the presence of shallow ownership with its inherent weaknesses, and in Parameterised Race-Free Java (Boyapati and Rinard 2001). These systems sadly perpetuate the abstraction problem identified earlier, since they rely on method-level annotations (indeed, Parameterised Race-Free Java suffers from additional abstraction problems described in more detail in Section 8.4.3, page 119).

In our proposal, we enable internal back pointers to a unique object to be part of an aggregate's implementation. This avoids the abstraction problem presented earlier and is also more natural to object oriented programming.

3.3.1 Motivation for Chosing Ownership Types

Ownership types is the most mature and flexible system that offers strong encapsulation in a statically enforcable way. Its properties have been proven sound (Clarke, Potter, and Noble 1998; Clarke 2001) and is the basis and inspiration for many subsequent systems, *e.g.,* Joe_1 (Clarke and Drossopolou 2002) and the systems combining uniqueness and ownership types mentioned above. Deep ownership enables us to distinguish between the inside and outside of an object in the necessary way (see also Section 4.2.4 for a short discussion of shallow ownership and external uniqueness). Also, an important discovery led us to the insight that enabling a strong notion of uniqueness was relatively cheap in the presence of ownership types further convincing us that ownership types was a suitable base.

We now present our proposal, external uniqueness, and Joline, a class-based object-oriented Java-like language that realises our ideas.

> *Throughout the remainder of this thesis, whenever we refer to ownership types, we assume a deep model of ownership adding the appropriate qualifiers either where required or for emphasis.*

4

External Uniqueness

This chapter presents external uniqueness, a new form of uniqueness that overcomes the abstraction problem inherent in traditional uniqueness. Technically, it is a minor extension to ownership types, but with major consequences. For concreteness, we work in the context of a core programming language called Joline, an extended subset of Java.

The chapter begins with description of external uniqueness and the operations required to support it. We then address a few technicalities required to maintain soundness of the system. We save the formal account the Joline programming language for the Chapter 5.

4.1 Tour de External Uniqueness

A reference to an object is *externally unique* if it is the only reference *from outside* an object to it. Aliasing from inside the object is still permitted, because such references form a part of the aggregate object's implementation. Thus, external uniqueness takes uniqueness, but only applies it externally, relying on the distinction between the inside and outside of an object given by ownership types. Once you understand ownership types, external uniqueness is as simple an idea as traditional uniqueness, although maintaining it is a little trickier.

Figure 4.1 illustrates the distinction between our key concepts so far, uniqueness and deep ownership, and external uniqueness. The dotted edge u in the rightmost diagram denotes an externally unique reference. As internal references to the grey object are still permitted, we argue that external uniqueness is a more suitable form of uniqueness for object-oriented programming languages; it considers aggregates, not just single objects, meaning that external uniqueness enables a single entry point to an entire collection of objects that can be moved by just moving the single pointer.

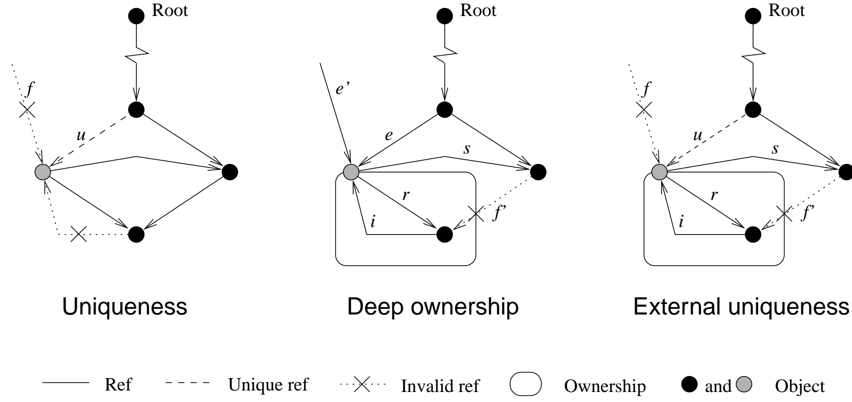


Fig. 4.1. Uniqueness vs. Ownership vs. External uniqueness. *Reference kinds*: u – (externally) unique. f, f' – invalid (f breaks uniqueness, f' breaks deep ownership). s – sibling. e, e' – external to gray object. i – internal to the gray object. r – reference to the gray object's representation.

The formal property of external uniqueness is:

$$\iota \text{ refers to } \iota' \text{ uniquely} \wedge \iota'' \text{ refers to } \iota' \text{ non-uniquely} \;\Rightarrow\; \iota'' \text{ inside } \mathsf{owner}(\iota').$$

Also, there may be only one unique pointer to an object at one time in the system. This property states that all non-unique references to a unique object are stored in objects *internal* to the object. In combination, the fact that there can be only one unique reference to an object at one time in the system and the fact that owners are dominating nodes (since the owners-as-dominators property holds), we get that unique references are *dominating edges* (see Section 4.1.2).

Externally unique references are denoted using types syntactically compatible to types with ownership:

$$\texttt{unique}{:}c\langle p_{i\in 1..n}\rangle$$

The unique annotation can only appear in the owner position; no p_i for $i \geq 1$ may be the keyword `unique`. We allow uniques to *move*—change ownership. This means that uniquely referenced parts of an object's representation can be detached and become part of another object's representation. In previous systems with ownership types, owners have been fixed for life since changing an owner of an object requires that all aliases to the object update their view of the object accordingly. This is generally not possible at compile-time due to the difficulties of tracing aliases. To maintain the strong encapsulation of deep ownership in the presence of ownership transfers, unique types need an additional bit of information to govern what moves are valid. This is dis-

cussed further in Section 4.1.5. As ownership types maintain the dominators property, to obtain external uniqueness we need only add a little machinery to ensure the uniqueness of references of unique type whenever viewed externally.

4.1.1 Fields and Blocks as Owners

As we have already identified, a unique object is effectively owned by its holder. Intuitively, thus, it makes sense to have the uniqueness annotation have a semantics similar to `this` when used in the owner position of a type. However, this does not get us the entire way. Instead, as we promised in Section 3.2.2, we make a small addition to the domain of owners and allow not only objects, but variables, fields and blocks to be used as owners. This is arguably an intuitive approach as it is not only the holder of a unique reference that dominates an object, but more precisely the field containing the unique reference (since the field must appear in all paths to the object).

To enable field and variable owners, the `unique` annotation simply means "the field of this type is the owner of the object it holds", *i.e.,* `unique:Object obj` is interpreted as `obj:Object obj`. By not allowing the programmer to use the field names directly as owners, we prevent the use of `obj` several times to own objects in other fields in the same object which is unsound in our system. In this light, transfer of ownership really means changing the owner of an object from the origin field name to the target field name. Note that it is not the field *name* that is the owner (since there is no mechanism to prevent several object of the same or different types to use the same field name), but rather the field *occurence* itself.

Allowing blocks to be owners might be less intuitive. (Remember, a block is a sequence of statements enclosed within `{...}` brackets.) Basically, we define a temporary owner for the scope of a particular block, just as many languages allow declaration of block-scoped variables within a block. This allows the creation of very localised objects, since as the block exits, the owner can no longer be named. Thus, the type of the object cannot be formed outside the block and therefore no variables or fields can have the type necessary to contain a reference to the localised object.

Uniqueness in the owner position of a type means that the field (or variable or parameter) associated with the type is the owner of the object it holds. Thus, we get a form of types that are "connected" to a certain field. Moving an object from one field to another now requires a change of type of the moving object, even when moving an object between fields or variables in the same object or on the same frame.

Since objects may not share fields, each run-time type of a unique object is a singleton type. The same is also true for variables, parameters and blocks—even if the same method or block is entered twice, the stack frames will be different, meaning that the variables etc. can be thought of as unique instances for that particular frame. In our formal semantics, we use object id plus field name as unique owners and consider variables uniquely named to achieve unique run-time owners for run-time types. Note that a run-time representation of owners is not necessary to maintain the ownership or external uniqueness properties, neither in most ownership types systems nor in the system presented here. We simply include it here to make the formal system easier and to improve the presentation of the Joline system.

4.1.2 Owners are Dominating Edges

The graph-theoretic property of ownership types is that owners are dominating nodes in the object graph. The corresponding property for external uniqueness is a refinement of the owners-as-dominators property; an externally unique reference instead corresponds to a *dominating edge* in the object graph. A dominating edge is an edge that must occur on all paths from the root of the system to the target. This can be observed in the rightmost diagram in Figure 4.1—the dotted edge labeled u is a dominating edge for the grey object and its representation. This is the same property as is given by traditional uniqueness. However, without the strong encapsulation property of deep ownership, the owners-as-dominating edges property becomes a lot less powerful as the edge only dominates the target object, instead of the entire representation of a target aggregate.

The dominating edge property implies that if the dominating edge u is removed, then all internal objects (*i.e.,* objects within the rounded box in Figure 4.1) become inaccessible from the rest of the system. This is in contrast to ownership typing, where the removal of the grey *dominator object* would result in its internal objects becoming inaccessible.

4.1.3 Externally Unique is Effectively Unique

Effective uniqueness means that even though an object may have more than one pointer to it, all but one pointer are somehow hidden or otherwise non-accessible which means that the semantics are the same as if it were actually unique (there is only one pointer). In this sense, external uniqueness is effectively unique since any non-unique pointer to a unique object is internal to the unique object and the internals of a unique object is never accessible. The external, unique pointer, is in effect the only pointer to the object.

Effective uniqueness is enabled by the dominating edge property. While in place, the dominating edge is the only way to access an externally unique object. We cannot invoke a method on a unique reference, nor can we use it to access fields of the object it refers. Thus, we cannot reach the possible internal pointers that violate actual uniqueness.

In the presence of borrowing, some kind of mechanism is needed to prevent a borrowed variable from being accessed while borrowed since otherwise the uniqueness expectation of the second access would be violated and uniqueness compromised. As was previously discussed on Page 48 there are a number of ways to deal with this problem: simply weaken uniqueness, temporarily nullify the value of the borrowed variable which introduces additional null-pointers and possible race conditions or we could employ alias burying which could also cause deadlocks or race conditions.

Any of the second two, non-weakening, solutions, renders external uniqueness effectively unique in the sense that was described above.

4.1.4 Operations on Externally Unique References

In our proposal, unique values (fields, variables and parameters) can only be operated upon in two ways—by *movement* and by *borrowing*. Movement is essentially a destructive read that involves changing the owner of the object; borrowing is a construction that allows a unique object to temporarily become a non-unique. To perform any other operation on a unique object, it must first be borrowed after which the borrowed reference can be reinstated and again become unique. This separation of operations and the borrowing mechanism that allows a unique to be temporarily converted into a regular non-unique and back again is a key to overcoming the abstraction problem. Also, as we shall see, they also make the borrowing pointer along with its operations and keywords unnecessary.

Movement

Movement is the transferring of a unique pointer from one field (or variable or parameter) to another. This involves changing the type of the moving object[1]. The target type may be non-unique, *i.e.,* moving may mean losing uniqueness.

For simplicity, we choose destructive reads (described in Section 2.2.1) as the scheme for maintaining uniqueness. This means that read access of a unique field has the side-effect of nullifying the field.

A unique value, as opposed to a unique field, can be obtained through object creation, by destructively reading a unique field or variable, or as the

[1] Except in the case of the object moving from one field to the same field.

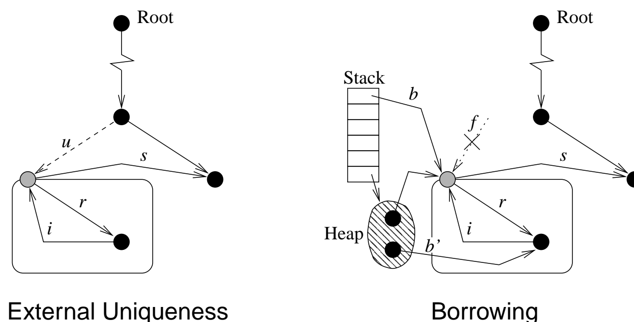


Fig. 4.2. Mediating between external uniqueness and borrowing — b is the original borrowed reference. b, b' are only valid during the borrowing. (Stack grows downwards.)

result of a method call. To simplify the formal account of the Joline language, destructive reads are made explicit. To a certain degreee, the explicit syntax also has the benefit of making the semantics clearer, thus aiding the programmer in dealing with slipperiness. In Joline, a programmer writes `x = y--` or `return y--`, instead of `x = y` or `return y`, respectively when `y` is unique. Ultimately, a unique value is consumed by assigning it to a field or variable, or by passing it as an argument to a method.

Since the type of a moving object changes, the key to sound movement is to make sure that no two aliases have conflicting views of the object simultaneously. In our system, the internal pointers have a different view of an encapsulating, externally unique object, but as we shall see, these pointers are never active at the same time as the unique. Thus, the problem is circumvented (see also Section 4.1.3). We achieve this by using a construct called borrowing.

If an object moved inwards, that is, was moved into itself, it would detach itself from the object graph and become its own owner which is unsound in our system. Luckily, this is not possible. To move an object, you need a unique reference to it. However, to be able to pass something into a unique object, its unique reference must first be converted to a non-unique reference by the borrowing construct (see below). Thus, moving an object into itself would require us to have both a unique and a non-unique reference to the same object simultaneously which is not possible. This means inwards movement is not possible.

Borrowing

Performing an operation on a unique reference other than moving it, *e.g.*, accessing a field or calling a method, requires that the reference first be bor-

rowed. This is done using a syntactic construct in the language called borrowing. The borrowing construct creates a non-unique reference to the borrowed entity for a limited lexical scope. Remembering the discussion in the paragraph just above, this non-unique reference has the same view of the object as any internal pointers to it. In the example below, $lval$ is the entity being borrowed, x is the variable to contain the non-unique reference to the borrowed object and s is the scope of x.

$\texttt{borrow}\ lval\ t\ \texttt{as}\ \langle\alpha\rangle\ x\ \texttt{\{}\ s\ \texttt{\}}$

The borrowing construct moves $lval$ (of type t) into x and gives it a new owner, α also defined for scope s. Since the name α is visible, other types using α as an owner parameter can be formed in s, enabling the creation of siblings to the borrowed entity, etc. The reason for including t in the statement is to simplify the formal account. Devising an elaboration function that instead fills in the necessary information prior to the compilation is easy. For brevity, we do not include the type information in our examples.

We now take a more concrete example: If we have a unique pointer to a `List` instance in the variable `unique:List list`, to call its `append` method we are required to do the following:

```
borrow list as <temp> borrowed { borrowed.append(...); }
```

Figure 4.2 shows what happens during the phases of a borrowing. The left-hand picture indicates the state of the program before the borrowing occurs. Initially, all access paths, including path originating from the stack, from the root to the grey object and its representation contain the unique reference u (`list` in our example). If the reference in `list` is in place, it is inactive; otherwise, a previous borrowing would have consumed it. Thus the object and its representation are also inactive. During the borrowing, depicted in the right-hand picture, the original reference is placed in variable b (`borrowed` in our example) which can be treated as an ordinary, non-unique variable. The type of `borrowed` is not unique, having owner `temp`, which is a "fresh owner", for the duration of the borrowing. While in scope of the borrowing (*i.e.,* between the `{` and `}` brackets, and in methods called within the borrowing block) we can access the fields and methods of the borrowed object, pass it to methods, or even store it in subsequent stack frames or on locally created heaps (*i.e.,* in objects owned by `temp` or objects inside such objects, see Section 4.2.2). At the end of the borrowing, `temp` is invalidated (goes out of scope), and thus there can be no valid or active references to objects with `temp` in its type even if the borrowed object was referenced from the heap. Only internal aliases remain, and the situation returns to the original inactive state.

The last thing to happen is that the value of `borrowed` is reinstated into `list`. Since it is the only variable who's contents is saved, whatever reference it contains will be the only external reference into the set of objects owned by the temporary owner (*i.e.,* it is an externally unique pointer to the borrowed aggregate). Since `borrowed` is a regular variable, it might have been updated during the execution of the borrowing block meaning that the reinstated pointer is not necessarily the same as the borrowed pointer.

Maintaining External Uniqueness

Borrowing exists primarily to facilitate type checking by providing a construct to convert between uniqueness and non-uniqueness. Ownership typing ensures that no reference escapes from the borrowing construct via some back door and that an object's internal references never escapes. Thus, when a unique reference is in place, all internal references are inactive, meaning that they cannot be used in a way that would violate uniqueness. When the unique reference is borrowed, it is replaced by a non-unique reference of the same kind as the internal references, and the object can be treated as a regular non-unique object. However, the externally unique reference must be handled correctly, and its uniqueness not be compromised.

We now consider how to maintain external uniqueness. There are essentially three approaches (the same as introduced earlier in Section 2.2.1):

do nothing — rather than invalidate the contents of the borrowed variable, we could simply weaken the definition of uniqueness, permitting both the reference in the borrowed variable and the borrowed references, and even allow movement of the borrowed value underfoot.

destructive — we could nullify the borrowed variable during borrowing, and then either:

- simply restore the original contents of the borrowed variable when the borrowing ceases;
- restore the final contents of the borrowing variable at the end of the borrowing (this was done in our previous example). Restoring the initial value is consistent with conventional uniqueness, whereas enabling a different reference into the same aggregate to be reinstated is consistent with external uniqueness; or
- rather than simply nullify the borrowed variable, we could record the state of its contents. There are three possible states: *available*, *null*, and *borrowed*, indicating that the variable contains something, nothing, or is disabled due to some currently active borrowing. In the presence of multiple threads, additional states could be added to indicate whether a different thread is borrowing the reference.

This solution would give the programmer full control, dynamically, of the possible ways to handle an attempt to borrow an already borrowed variable etc. If borrowing an empty variable, it might be desirable to perform an initialisation first; if trying to borrow an already borrowed variable, some extra cation or other fall-back action might be appropriate.

alias burying — the last possibility is to employ alias burying, as outlined when introducing uniqueness; instead of requiring uniques to be destructively read, we can allow multiple references to a unique object as long as all but one reference is buried, *e.g.,* out of scope or otherwise invalidated. This would ensure that when the variable is read, all its aliases are unusable (Boyland 2001a). Alias burying eliminates the need for destructive reads, but unfortunately is costly in other respects. As it is based on program analysis, its strength is sensitive to the underlying analysis. To achieve modular checking, interfaces must be further annotated to indicate which unique fields are read by what methods (Boyland 2001b). This may well reintroduce the abstraction problem, for example if a method's implementation is changed so that a previously borrowed variable is stored on the heap or if synchronisation is employed to prevent simultaneous access of a borrowed field. For fields, Alias Burying requires that borrowed fields be locked in order to prevent simultaneous accesses. This is similar to the temporary nullification of destructive reads, but the semantics and effects on practical programming are much nicer.

In case of destructive reads and alias burying, the equi-strong aliasing properties given by both schemes ensures that there is no active reference to the target object other than the reference extracted from the borrowed variable. As was stated earlier, we choose the destructive reads approach to maintain uniqueness. This keeps our formal system simple, while retaining a strong definition of uniqueness.

A drawback of destructive reads is that it precludes simultaneous non-conflicting operations on unique references, *e.g.,* allowing read-only methods on unique references even during a borrowing.

4.1.5 Movement Bounds

To enable subtyping in ownership types, subsumption is allowed to "forget" owner parameters. Classes `A` and `B` in Figure 4.3 illustrate this. When appearing as an instance of class `A`, an instance of class `B` may still hold aliases to external objects owned by owners that are no longer visible in the type (*e.g.,* in field `data2`). Due to dynamic binding, methods may be called that operate

```
class A<a>
{
  this:Something<a> data1;
}

class B<b outside owner, a inside b> extends A<a> // (**)
{
  this:Something<b> data2;
}

// o1 inside o2
unique:B<o1,o2> ok = new unique:B<o1,o2>();
o2:A<o2> notok = ok--; // (*)
```

Fig. 4.3. Violation of the external uniqueness invariant caused by movement in the presence of subtyping. After line `(*)`, the object in `notok` is moved outside of `o1` while still retaining a reference to it, breaking the owners-as-dominators property.

on `data2` but since `b` is not visible in the interface, its type cannot be named and thus, the contents of the variable cannot be returned. Thus, the containment invariant is preserved.

Movement introduces another problem since movement changes the types of objects. Moving what appears to be an instance of class `A` could lead to a violation of the external uniqueness invariant if the instance was really of some other type. Consider the lines of code at the bottom of Figure 4.3.

As the example shows, moving an object to a new location could result in residual aliasing of the internals of its original location, thus violating the invariants of deep ownership and external uniqueness. When moving an instance of class `B`, we are aware of the relation between the owner and the parameters `a` and `b`. In order to preserve the invariants which come as a consequence of the type of `data2`, `b` must be outside `this`. In the line marked `(*)`, an instance of `B` is moved to owner `o2` which is outside `o1`. This and breaks the strong encapsulation since `o1`-owned objects can still be referred to from inside of the moved object. Since the moved object is now owned by `o2`, an object external to `o1`, this breaks the owners-as-dominators property. An example of this is shown in Figure 4.4.
To avoid movement such as in Figure 4.4, all occurrences of `unique` have an associated *movement bound* which appears in the formal system as $\texttt{unique}_p$ (written `unique[p]` in code). The movement bound is a regular owner that bounds the outwards movement of a unique reference—a unique reference may only be moved to variables with owners inside its movement bound. In addition to limiting movement, the movement bound also ensures safety

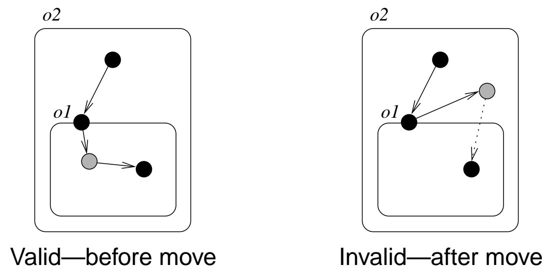


Fig. 4.4. Movement that violates deep ownership. In the leftmost figure, encapsulation is preserved. In the rightmost, after moving the gray object from o_1 to o_2, encapsulation is violated by the dotted reference crossing the boundary of o_1 from the outside to the inside.

when borrowing by ensuring that references do not escape the scope of the borrowing in a similar fashion.

The movement bound may be any visible owner inside the innermost owner used to form the type. In the example in Figure 4.3, `o1` is inside `o2` and thus the movement bound must be (inside) `o1`. The only possible type of `ok` is therefore `unique[o1]:B<o1,o2>`. To allow for some flexibility, the movement bound can also change, but only inwards, which always upholds deep ownership. Also, when losing uniqueness, the new owner is required to be inside the bound. These restrictions preclude any violating movement. Since `o1 inside o2` in the example, changing the owner of `ok` from `o1` to `o2` is a direct violation of these rules and our example would be caught as a type error during compilation.

Another minor restriction necessary to preserve our invariants in presence of movement is that `owner` may not appear as an owner parameter in the extends clause, *i.e.,* the rightmost `a` may not be replaced for `owner` on line `(**)` in the example in Figure 4.3. This is problematic since a supertype might end up with an owner that cannot be named (since the unique owner cannot be named externally, necessary to maintain the uniqueness), see 4.5. We choose to not permit `owner` in the problematic, non-owner, position.

```
class B<p outside owner> { ... }
class C extends B<owner> { ... }

unique:C c;
unique:B<??> b = c--; (**)
```

Fig. 4.5. Using `owner` in the extends-clause might make supertypes impossible to reference from outside. On line `(**)`, ?? indicates the lack of a name for the owner parameter to `B`.

Choosing a Movement Bound

In Appendix A.1, we present an elaboration function that gives a valid movement bound to all unique types. The default movement bound is equivalent to replacing all occurrences of `unique` with `unique[owner]`, except for code not inside classes (where `owner` is not defined), in which case `unique[world]` is used.

Another way is to put movement bounds under programmer control. The programmer may select any visible owner inside the lower bound with respect to the inside ordering of the owner parameters of the unique type as movement bound.

Choosing movement bound is a trade-off: an outer bound enables more movement, but limits what other objects the object can access (*i.e.,* what ownership parameters can appear in its type). An inner bound would enable less movement, but permit more access. Unique references with movement bound `world` can be moved anywhere in the system, but as for objects owned by world, can only reference other world-owned objects in addition to its representation.

4.2 Discussion

In this section, we discuss some of the issues and features around our proposal, including owner-polymorphic methods, generational ownership, how to deal with constructors and why shallow ownership does not suffice to enable external uniqueness.

4.2.1 Owner-Polymorphic Methods

Joline sports owner-polymorphic methods, which are methods that take owners as parameters. These methods can be given temporary permission to reference certain objects. The declaration of an owner-polymorphic method takes the following form:

```
<temp inside world> void method(temp:Vector p) { ... }
```

where the owner parameters appear in front of the rest of the method head (before the types that use them), inspired partly by GJ (Bracha, Odersky, Stoutamire, and Wadler 1998) and λ-calculus.

Below is an example of invoking an owner-polymorpic method when owner `o2` is not inside `o1`:

```
// o2 not inside o1
o1:Object r;
o2:Object a;
r.method<o2>(a);
```

In the code snippet above, `r` is owned by `o1` and since `o2` is not inside `o1`, regular methods in `r` are not permitted to reference `a`. However, we pass the method the temporary permission to reference objects owned by `o2`, and are thus allowed to pass `a` as an argument to `method`. Since the permission is temporary and cannot be stored in `r`, no references to `a` can be retained after the method has exited (other than references created in object inside `o2`).

4.2.2 Generational Ownership

The inclusion of scoped regions in our language enables an interesting feature that we call generational ownership[2]. Previously, the lowest owner accessible to a method was the current `this`. That meant that methods executed inside the receiver object in the ownership graph.

In generational ownership, a stack frame can be thought of as being nested inside all existing owners in the ownership dag. The introduction of blocks as owners allow creation of stack-owned heaps and gives an ownership structure such as the one depicted in Figure 4.6. The uniquely referenced subheaps are due to external uniquenss. They are subheaps with only one ingoing pointer, *i.e.,* the top object is referenced externally unique and no other objects in the heap are referenced from outside the heap.

A stack frame with a heap can be thought of as a “generation of objects”. Later stack frames are ordered inside earlier stack frames (which is the only sensible option since the owners of later generations does not exist when previous stack frames are still active); later generations can access all previous generations, but not vice versa. Thus, the top stack frame and objects in its heap are ordered inside all other objects in the system and the bottom frame (*i.e.,* the first frame created) corresponds to `world`. Since objects belonging to earlier generations cannot reference objects of later generations, when a stack frame is destroyed, the contents of its heap can no longer be referenced. Thus the entire generation of objects can be destroyed.

4.2.3 Constructors

Normally, instantiating an object from a class template results in a unique object. If a constructor is called on the instantiated object, it might create

[2] The name generational ownership was coined by John Potter.

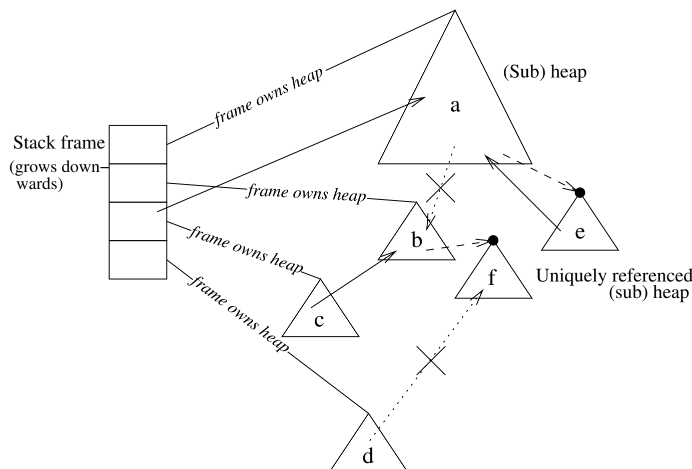


Fig. 4.6. Generational ownership. The lines without arrows indicate ownership, the triangles are heaps and the rectangles are stack frames in a downwards growing stack. Arrows are references, dashed arrows denote externally unique references and dotted arrows denote invalid references. Bullets are externally unique objects.

aliases to the object that renders it non-unique. Thus, object creation with constructors in the presence of (external) uniqueness is a bit tricky.

Constructors are effectively methods called only once and thus have the same restrictions as ordinary methods. In an ordinary method call, external uniqueness would be violated if the method created a path to `this` in a preexisting external object. In case of a constructor, this would mean that the result of an object creation would not be an externally unique object.

To be accessible in the method, the preexisting external object would have to have `owner` as its owner and be accessible from an object passed to the method. If the `owner` was merely acting as the movement bound, there would be no problem, because such an argument to the constructor would be unique and thus be consumed by the constructor into the new object meaning any aliases to `this` stored in the argument object would be internal. Thus, the fix is simple: *the parameters of a constructor cannot have* `owner` *in their type, except as a movement bound*. This is a minor restriction as constructors may still make static aliases to its arguments and create and receive any objects.

Having argued how to deal with unique object creation in the presence of constructors using only minor restrictions, we omit constructors from our language description for simplicity.

4.2.4 Shallow Ownership does not Suffice

The property of external uniqueness that owners are dominating edges is a refinement of the owners-as-dominators property of ownership types, and

also a strengthening of the property of traditional uniqueness. For traditional uniqueness, trivially a unique reference is a dominating edge since there can be no other references to the object. In external uniqueness, a dominating edge also dominates all objects in the referenced aggregate, not just the one uniquely referenced. This deep version is stronger and therefore more powerful than the shallow property given by traditional uniqueness. It should provide more reasoning power, and also allow movement of aggregates in contrast to movement of a single object possibly leaving its representation in place.

Dominating edges are possible only in systems where dominators can be enforced, *i.e.,* in systems that offer deep ownership. Shallow ownership systems, as offered by Boyland et al.'s Capabilities for sharing (2001), Aldrich et al.'s AliasJava (2002) and some systems in Clarke's dissertation (2001) lack the necessary strength to support external uniqueness since there is no way of distinguishing between inside and outside of an object, or because internal references may escape to the outside and compromise a supposedly externally unique reference.

Adding traditional uniqueness in systems with shallow ownership is easy, since the invariants underlying deep ownership need not be considered.

5

Formalising External Uniqueness

In this section, we formalise our proposal in a Java-like programming language called Joline. We show the static and dynamic semantics of our proposal and outline the most important proofs.

5.1 Introducing Joline

Although quite different, the Joline programming language carries on the tradition of Joe_1 (Clarke and Drossopolou 2002), but lacks effect annotations etc. It also reintroduces the nesting relations in the class header and uses a local state instead of let-statements. This section describes the syntax, static semantics and dynamic semantics of Joline along with its key properties—a structural property that captures both the owners-as-dominators property in the presence of owner polymorphic methods (generational ownership) and the owners-as-dominating edges property of external uniqueness, and regular soundness.

5.1.1 Syntax

Joline's syntax is shown in Table 5.1. It should be familiar to anyone with some experience in Java. A program is a collection of classes followed by a statement and a resulting expression. We could have followed Java's example and use static methods etc., but chose this way out for simplicity.

Classes are parameterised with owner parameters. Each owner parameter (except the implicit, first, `owner` parameter) must be related to either `owner` or some previous parameter of the same class. Classes contain fields and methods; fields must be initialised and methods may carry owner parameters (remember the discussion about owner polymorphic methods earlier).

The most interesting pieces of syntax are undoubtedly these: The destructive read suffix, `--`, required to destructively read a field or variable; Object

$c \in$ **ClassName** $\quad f \in$ **FieldName** $\quad md \in$ **MethodName**
$x, y \in$ **TermVar** $\quad \alpha \in$ **OwnerVar**

P	$::=$	$class_{i\in 1..n}\ s\ e$	(Program)
class	$::=$	$\texttt{class}\ c\langle \alpha_i\ \mathsf{R}_i\ p_{i\in 1..m}\rangle\ \texttt{extends}\ c'\langle p'_{i'\in 1..n}\rangle\ \{\ fd_{j\in 1..r}\ meth_{k\in 1..s}\ \}$	(Class)
fd	$::=$	$t\ f = e;$	(Field)
meth	$::=$	$\langle \alpha_i\ \mathsf{R}_i\ p_{i\in 1..m}\rangle\ t\ md(t_i\ x_{i\in 1..n})\ \{\ s\ \texttt{return}\ e\ \}$	(Method)
lval	$::=$		(L-value)
		x	(variable)
		$e.f$	(field)
e	$::=$		(Expression)
		`this`	(this)
		$lval$	(l-value)
		$lval$`--`	(destructive read)
		`new` t	(new)
		`null`	(null)
		$e.md\langle p_{j\in 1..m}\rangle(e_{i\in 1..n})$	(method call)
s	$::=$		(Statement)
		`skip;`	(skip)
		$t\ x = e;$	(variable delcaration)
		$e;$	(expression)
		$lval = e;$	(update of lvalue)
		$s_1;\ s_2$	(sequence)
		$\texttt{if}\ (e)\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}$	(if-statement)
		$(\alpha)\ \{\ s\ \}$	(scoped region)
		$\{\ s\ \}$	(block)
		$\texttt{borrow}\ lval\ t\ \texttt{as}\ \langle \alpha\rangle\ x\ \{\ s\ \}$	(borrow)
p, q	$::=$		(Owners)
		`this`	(this)
		α	(owner parameter)
		`world`	(world)
		`owner`	(owner)
		`unique`	(unique)
		$\texttt{unique}_p$	(unique in elaborated language)
t	$::=$		(Type)
		$p : c\langle p_{i\in 1..n}\rangle$	

Table 5.1. Syntax of Joline

Table 5.2. Helper function definitions.

Owners

$$\text{owners}(p_1 : c\langle p_{i\in 2..n}\rangle) \mathrel{\widehat{=}} \{p_{i\in 1..n}\}$$
$$\text{owners}(\texttt{unique}_b : c\langle p_{i\in 1..n}\rangle) \mathrel{\widehat{=}} \{b, p_{i\in 1..n}\}$$
$$\text{owners}(E) \mathrel{\widehat{=}} \text{The set of all owners defined in } E \text{ plus } \texttt{world}$$

creation which requires the owner parameters specified in the class header to be bound to owners in scope; Method call, that may require owner arguments if the invoked method is owner polymorphic; The scoped region construct that introduces a new owner, α, for a given block; and borrowing, as has been explained previously.

5.1.2 Joline's Type System

In this section, we present the static semantics for Joline. First, we introduce the static typing environment, 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 the types and subtypes, including unique types and rules for losing uniqueness or moving a unique type to into another bound. Last, we present the type rules for Joline's statements and expressions.

Type checking is defined for programs which have movement bounds in place (either added manually by the programmer using the `unique[owner]` syntax or inferred by the elaboration function in Appendix A.1).

Static Type Environment

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

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

where $\alpha \prec^* \bigsqcup\{p_{i\in 1..n}\}$ means α is inside all $p \in \{p_{i\in 1..n}\}$. We will use the test $\text{isunique}(t)$, that is true if and only if the type t is unique, *i.e.,* $\text{isunique}(t)$ iff $t = \texttt{unique}_b : c\langle\sigma\rangle$. 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 (see Table 5.2 for formal definitions).

Substitutions

Substitution of owners for owner parameters is denoted σ where σ is a map from owner variables to owners. If type $p\!:\!c\langle q\rangle$ is formed from the class definition

```
class c<data inside owner>
{
  ...
}
```

we sometimes write $p : c\langle\sigma\rangle$ for the same type where $\sigma = \{\texttt{data} \mapsto q\}$. We will also sometimes write σ^p to mean $\sigma \cup \{\texttt{owner} \mapsto p\}$ or σ_n to mean $\sigma \cup \{\texttt{this} \mapsto n\}$ or σ_n^p for the combination. Applying a substitution to a type is written $\sigma(t)$ or $\sigma(t \to t')$ and is defined thus:

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

Applying a substitution to an owner is written $\sigma(p)$ or $\sigma(\alpha\,\mathsf{R}\,p)$ and is defined thus:

$$\sigma(p) = q,\ \text{if } p \mapsto q \in \sigma$$
$$\sigma(p) = p,\ \text{if } p \mapsto q \notin \sigma$$
$$\sigma(p\,\mathsf{R}\,q) = \sigma(p)\,\mathsf{R}\,\sigma(q)$$

Sometimes, we compose several substitution maps. We write $\circ$ for composition of substitution maps:

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

Composition of substitution maps is equivalent to ordinary function composition and therefore is associative.

Field Lookup

For any class c, $\mathcal{F}_c$ is a map from the names of all field variables defined in c and all of its superclasses to their corresponding types, *i.e.,* for class

```
class List<data outside owner> extends Object
{
  this Link<data> first;
  this Link<data> last;
```

```
}
```

we have $\mathcal{F}_{\texttt{List}} = \{\ \texttt{first} \mapsto \texttt{this:Link}\langle\texttt{data}\rangle,\ \texttt{next} \mapsto \texttt{this:Link}\langle\texttt{data}\rangle\ \}$. Field lookup is formally defined thus:

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

The σ on the second line is a map from the owner names used in c' to those used in c to facilitate subclassing and $\bot$ means that the method is not defined for class c. P is the complete program, a global constant for all the rules except $\vdash P$ to reduce the syntactic overhead necessary to thread P from the top level to all the rules where it is required. When looking up a field variable for class c on the second 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 “translate” the types into using the names visible in c.

Method Lookup

Method lookup works 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. Thus, if the following methods were present in the `Link` class above

```
data Object get(world Int position) { ... }
world Boolean add(data Object obj) { ... }
```

we would have the following map for c: $\mathcal{M}_c = \{\texttt{get} \mapsto (\texttt{world:Int} \rightarrow \texttt{data:Object}), \texttt{add} \mapsto (\texttt{data:Object} \rightarrow \texttt{world:Boolean})\}$.

As for field lookup, the owner names used in the classes may vary even in the same hierarchy and therefore a translation is “built into” the method lookup function. Thus, when looking up a method in class c, the types returned will use owner names defined in c.

$$\mathcal{M}_c(md) = \begin{cases} \bot, & \text{if } c \equiv \texttt{Object} \\ (t \rightarrow t'), & \text{if } \texttt{class}\ c\ \cdots\ \{\ \cdots\ t'\ md(t\ x)\{\cdots\}\ \cdots\ \} \in P \\ \sigma(\mathcal{M}_{c'}(md)), & \text{if } \texttt{class}\ c\langle_\rangle\ \texttt{extends}\ c'\langle\sigma\rangle \cdots \in P \end{cases}$$

Table 5.3. Judgements used in the static semantics.

$E \vdash \diamond$	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
$E \vdash s\,; E'$	Statement s produces environment E'
$E \vdash meth$	Good method
$\vdash$ *Class*	Good class
$\vdash P$	Good program

Static Semantics

In this section, we present the static semantics of Joline. An overview of the different judgements used is found in Table 5.3.

Good environment

$$\text{(ENV-}\epsilon\text{)}\quad \frac{}{\epsilon \vdash \diamond} \qquad \text{(ENV-}x\text{)}\quad \frac{E \vdash t \qquad x \notin dom(E)}{E, x :: t \vdash \diamond}$$

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

The rules for good environment are pretty straight-forward. (ENV-ϵ) states that the empty environment, ϵ, is well-formed. (ENV-x) states that adding a variable name to type binding, $x :: t$ in 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. (ENV-$\prec^*$) states the same, but for the $\prec^*$ relation. It also permits ordering an owner inside several good owners, thus making the owners and their orderings a directed graph.

Good owner

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

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.

Owner Orderings

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

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

$P = \{p_{i \in 1..n}\}$ in (IN-ENV1) above. The inside and outside relations are derived from the owner orderings in E. The relations are transitive and reflexive and owners and their ordering form a dag (since an owner can be ordered inside several other owners 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`.

Program and Class

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

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

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.

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 mapping from owner names used in the class header of c' to the owner names $\alpha_{i \in 1..m}$ used in c). The assertion `owner` $\notin rng(\sigma)$ prohibits the owner from being used as an owner parameter in the non-owner position of any supertype of c. Allowing

this could cause unsoundness when moving uniques and we will discuss this later in more detail. 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. Note the application of the σ-substitution to $\mathcal{M}_{c'}$ to translate the names of the owner parameters into the corresponding names in c.

Good method

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

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 and variables declared in its header.

Types

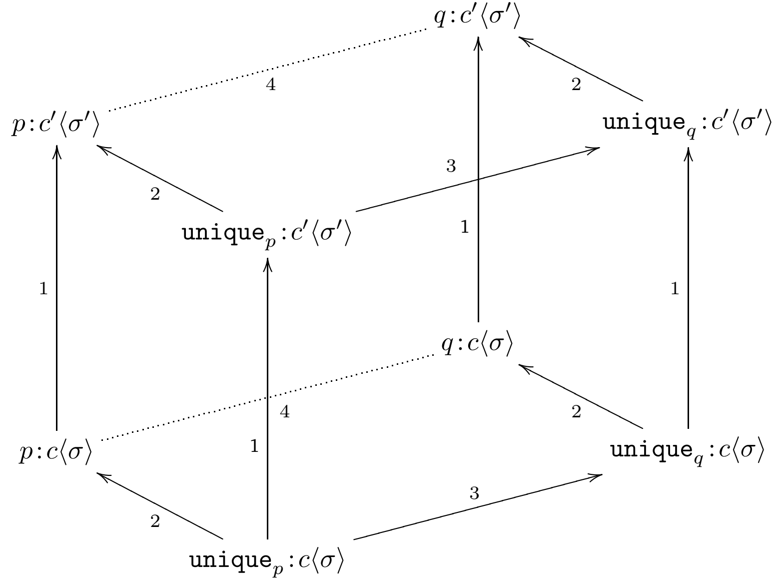
$$\text{(TYPE)}\quad \frac{\texttt{class}\ c\langle \alpha_i \, \mathsf{R}_i \, p_{i\in 1..n}\rangle \cdots \in P \qquad \sigma = \{\texttt{owner} \mapsto q, \alpha_i \mapsto q_{i\in 1..n}\} \quad E \vdash \sigma(\alpha_i \, \mathsf{R}_i \, p_i)_{i\in 1..n}}{E \vdash q : c\langle q_{i\in 1..n}\rangle}$$

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

A type is well-formed whenever the substituted owner arguments satisfy the ordering on parameters specified in the class header. A type can be either unique or not. In the case of unique types, the movement bound must satisfy the same constraint as it would satisfy if it was the actual owner. If it was possible to move a unique higher up in the ownership hierarchy than any of its owner parameters, it would break the owners-as-dominators property.

Subtyping

Subtyping in Joline involves not only traditional subtyping but also a unique losing its uniqueness and a unique moving (*i.e.,* changing its movement



Arrows are labelled 1-4 as follows: 1 (up) Regular subtyping; 2 (up left) Losing uniqueness ($\texttt{unique}_p$ becomes p, rest of type is unchanged.); and 3 (up right) Movement ($\texttt{unique}_p$ becomes $\texttt{unique}_q$, rest of type is unchanged); 4 (dotted lines) Invalid as non-unique types cannot move.

Fig. 5.1. Possible supertypes of $p : c\langle\sigma\rangle$ and $\texttt{unique}_p : c\langle\sigma\rangle$ when $c \leq c'$ and $q \prec^* p$.

bound). Figure 5.1 shows possible supertypes for types $p : c\langle\sigma\rangle$ and $\texttt{unique}_p : c\langle\sigma\rangle$ for some Γ s.t. $\Gamma \vdash q \prec^* p$ when c' is a superclass of c.

(SUB-CLASS)

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

(SUB-UNIQUE)

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

Subtyping is derived from subclassing, modulo the binding of superclass parameters. As this corresponds to the composition of two order-preserving functions, it is order-preserving, required to preserve deep ownership, see Clarke's dissertation (2001). In particular, subtyping preserves the owner, it is fixed for life for non-unique objects and may change with movement for unique objects. Letting the owner vary, as in Cyclone (Grossman, Morrisett, Jim, Hicks, Wang, and Cheney 2002), would be unsound in our system

(Clarke and Drossopolou 2002).

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

As expected, the subtype relation is reflexive and transitive. To simplify the formalism, we have no subsumption for uniques, *i.e.,* the programmer must explicitly convert a unique to a non-unique and explicitly tighten the bound of a unique.

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

$$\frac{E \vdash e :: \texttt{unique}_b : c\langle\sigma\rangle \qquad E \vdash p \prec^* b}{E \vdash (\texttt{unique}_p)\ e :: \texttt{unique}_p : c\langle\sigma\rangle} \text{ (MOVE-UNIQUE)}$$

The (LOSE-UNIQUE) and (MOVE-UNIQUE) rules are of special interest. Together, they allow movement of uniques from one place to another with and without the preservation of uniqueness. The (MOVE-UNIQUE) rule allows a unique type to be transformed if the target type's movement bound is inside the original bound, a possible strengthening of the movement restriction. This is crucial to preserve soundness as was discussed in Section 4.1.5. Losing uniqueness is possible by just dropping the `unique` and using the movement bound as the actual owner.

As stated above, losing uniqueness and tightening movement bounds are explicit operations. In difference with ordinary type casts, these operations actually modify their targets (*e.g.,* moving subheaps into other objects, destroying uniqueness wrappers). Thus, preventing them from occuring implicitly in our calculus significantly reduces the complexity.

Statements

$$\frac{x \notin dom(E) \quad E \vdash e :: t}{E \vdash t\ x\ = e;\ ; E, x :: t} \text{ (STAT-LOCAL)}$$

(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).

$$\frac{E \vdash \diamond}{E \vdash \texttt{skip;}\ ;E}\ (\text{STAT-SKIP}) \qquad \frac{E \vdash e :: t}{E \vdash e\texttt{;}\ ;E}\ (\text{STAT-EXPR}) \qquad \frac{E \vdash \mathit{lval} : t\ \textbf{ref} \quad E \vdash e :: t}{E \vdash \mathit{lval} = e\texttt{;}\ ;E}\ (\text{STAT-UPDATE})$$

skipp 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. Remember that lval is both x and $x.f$.

$$\frac{E \vdash s_1\ ;E'' \quad E'' \vdash s_2\ ;E'}{E \vdash s_1\ s_2\ ;E'}\ (\text{STAT-SEQUENCE}) \qquad \frac{E \vdash s\ ;E'}{E \vdash \{s\}\ ;E}\ (\text{STAT-BLOCK})$$

From (STAT-SEQUENCE), statements can be sequenced in an unsurprising fashion. From (STAT-BLOCK), statements in a block may produce new environments (*i.e.,* make new variables or owner parameters visible). However, these are "removed" at the exit of the block and only the initial environment when entering the block is left.

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

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

The rule (STAT-SCOPED-REGION) introduces a new owner variable for the duration of the given block. This is "blocks-as-owners" (the owner is only defined for the duration of the borrowing block). The bounds P, though unspecified in code, determine which objects may be accessed by objects created in this scope. Thus, a owner introduced by a scoped region is potentially internal to all objects in scope. The reason for requiring the type of the borrowed *lval* to be present is to make the dynamic semantics simpler. There is nothing to prevent the necessary information to be elaborated in.

The rule (STAT-BORROW) states that any uniquely typed l-value may be borrowed[1]. This is achieved by introducing a new owner variable which is restricted in scope 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 into the borrowed value do not escape this scope,

[1] For the dynamic system, we make a simplification and only allow borrowing from local variables. This does not make the system weaker.

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 it is reinstated with a correctly typed value when the borrowing ends.

Again, to simplify the formalism, we only permit borrowing from local variables. This does not make the language less powerful, since borrowing a field can be simulated by first moving its contents to a local variable, borrowing from the variable and then reinstating its contents into the field.

l-values

(LVAL-VAR)

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

(LVAL-FIELD)

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

The rules above give the types of l-values, which are variables (other than `this`) and fields. Their type is given exactly as declared, modulo substitution of parameters. l-values may be updated or destructively read. The condition $\texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this}$, which was called the *static visibility* in the original ownership types system (Clarke, Potter, and Noble 1998), ensures that types that contain `this` in them, *i.e.,* 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 representation objects in are private. This is not essential; we could have used *dynamic aliasing* as in Joe_1 (Clarke and Drossopolou 2002), but the resulting type system would have been too complex to present our ideas. Also, allowing dynamic aliases arguably opens up a hole in deep ownership types which might not be desirable. Note that subsumption *does not* apply to l-value types to ensure the validity of reinstatement at the end of borrowing or implicit conversion of a unique type to a non-unique type.

Expressions

(EXPR-LVAL)

$$\frac{E \vdash lval :: t\ \textbf{ref} \qquad \neg\mathsf{isunique}(t)}{E \vdash lval :: t}$$

(EXPR-DREAD)

$$\frac{E \vdash lval :: t\ \textbf{ref}}{E \vdash lval\texttt{--} :: t}$$

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

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

From (EXPR-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.

$$\frac{E \vdash p\!:\!c\langle\sigma\rangle}{E \vdash \texttt{new}\ p\!:\!c\langle\sigma\rangle :: \texttt{unique}_p\!:\!c\langle\sigma\rangle}\ (\text{EXPR-NEW})$$

The rule (EXPR-NEW) contains a subtle detail: instantiating an object creates an externally unique.

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

The rule for method call is a behemoth. First, it applies only to non-unique types (as does the rule for field access). The second line is the static visibility test, same 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 owner arguments of the target type and the owner arguments supplied to the method form two substitutions to transform the method's argument and return types into types terms of the owners in scope. The value supplied to each argument of the method must have the type expected by the method. The method may return a uniquely typed value. In the dynamic semantics, we make the simplification and allow only one owner parameter and method parameter. Subsumption does apply to expression typing if the type of the expression is not unique.

It would be possible to infer the binding σ' of owners to the parameters of the method, but that would introduce unnecessary complexity for our purposes here.

Remarks

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

5.1.3 Joline's Dynamic Semantics

Joline's dynamic semantics are formulated as a big-step operational semantics from configurations to configurations, $\langle C\,;\,S\,|\,s\rangle$ to $\langle C\,;\,S\,|\,v\rangle$ where C is the nested heap, S is the stack, s is a statement and v an optional value. The structure of heaps reflect the the nesting of objects imposed by ownership types and external uniqueness.

For brevity, the dynamic semantics we present here does not include generational ownership. Therefore, we remove the scoped region construct that allows the creation of a temporary owner on the stack. While it does not affect the structural properties of our proposal, generational ownership requires some additional complexity that would only obscure the formalism of the matter at hand. To this end, we save the presentation of generational ownership for the final PhD thesis.

We now begin by exploring the structure of the store and the store-type information.

Store Structure

The store consists of nested subheaps. We denote heaps H, stacks S, frames V and values v. We use C exclusively to denote a heap appearing at the top level, *i.e.,* a heap nested inside world.

Store

$$C, \mathrm{H} \quad ::= \quad \mathrm{H}, n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}] \quad | \quad \mathrm{H}, n \mapsto \mathsf{B}[\mathrm{H}] \quad | \quad \mathsf{nil}$$

The store consists of nested objects and *borrowing blocks*. Objects are written $n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]$ where n is the id of the object and C the name of the object's class. Borrowing blocks are written $n \mapsto \mathsf{B}[\mathrm{H}]$ where n is the id of the block. The borrowing blocks are not proper objects and cannot be referenced. They are used as place-holders in the semantics to keep track of a subheap belonging to a unique object while borrowed. H is a subheap component of an object or borrowing block, V a field-value mapping and M a method-implementation mapping.

Stack

$$S \quad ::= \quad S \bullet \mathrm{V} \quad | \quad \mathsf{nil}$$

The stack is the set of all frames in the program ordered from left (initial frame) to right (currently active stack frame).

Frame

$$\mathrm{V} \quad ::= \quad \mathrm{V}, x \mapsto v \quad | \quad \mathsf{nil}$$

A frame is a mapping from (field) variable names to values.

Value

$$v \quad ::= \quad \uparrow n \quad | \quad \mathsf{U}_n[v \mid \mathrm{H}] \quad | \quad null$$

There are three values in our semantics: pointers (written $\uparrow n$ for a pointer to object with id n), unique values (written $\mathsf{U}_n[v \mid \mathrm{H}]$ for a unique with id n, subheap H and pointer compartment v) and the special $null$ value. A unique is similar to a borrowing block, except that it can be stored in a field or variable.

All values (including uniques) can be stored in variables or fields and be passed to or returned from methods. A unique's subheap can be thought of as nested inside a field.

Store Typing

The store typing is a nested map from object ids to types. It contains the type information for all objects except uniques and objects nested inside uniques. Its structure is parallel with the structure of the store.

We write $\mathcal{H}$ for a set of types with other types nested inside them. We sometimes use Γ as an alias for $\mathcal{H}$ to denote the entire store-typing. Each id-type pair corresponds to an object and the id-type pairs nested inside the type are the typing of the object's subheap. There are three types, object types, $c\langle\sigma\rangle$, borrowing block types, $\mathfrak{B}$, and unique types, $\mathfrak{U}$.

$$\Gamma, \mathcal{H} ::= \mathcal{H}, n :: T[\mathcal{H}] \quad | \quad \mathsf{nil}$$
$$T ::= c\langle\sigma\rangle \quad | \quad \mathfrak{B} \quad | \quad \mathfrak{U}$$

We write $T[\mathcal{H}]$ for type T with types $\mathcal{H}$ nested inside it. When there are no types nested inside a type, we omit the braces and write simply $n :: \mathfrak{U}$ etc.

Types do not contain an owner component. Rather ownership is implicit in the structure of a heap and is computed as required. For example, from store type $\Gamma = p :: T[\mathcal{H}, n :: c\langle\sigma\rangle[\mathcal{H}']]$, we can compute that the type of n is $p : c\langle\sigma\rangle$. Lookup of types from pointers is defined as a recursive lookup, constantly pushing the owner of the current "level" (written as subscript):

$$\mathcal{H}(\uparrow m)_p = \begin{cases} p : c\langle\sigma\rangle & \text{if } \mathcal{H} = \mathcal{H}_1, m :: c\langle\sigma\rangle[\mathcal{H}_2], \mathcal{H}_3 \\ \mathcal{H}_2(\uparrow m)_n & \text{if } \mathcal{H} = \mathcal{H}_1, n :: T[\mathcal{H}_2], \mathcal{H}_3 \text{ and } m \in \mathsf{defs}(\mathcal{H}_2) \end{cases}$$

$\Gamma(\uparrow m)_p$ is equivalently defined. Since the owner of the top level is the constant `world`, we will generally omit the subscript and write $\Gamma(\uparrow m)$ to mean $\Gamma(\uparrow m)_{\texttt{world}}$.

Owner orderings

The helper function defs is defined for Γ. It returns a set of the objects typed by a store typing:

$$\begin{aligned} \mathsf{defs}(\mathcal{H}, n :: T[\mathcal{H}']) &= \mathsf{defs}(\mathcal{H}) \cup \{n\} \cup \mathsf{defs}(\mathcal{H}') \\ \mathsf{defs}(\mathsf{nil}) &= \emptyset \end{aligned}$$

Thus, $\mathsf{defs}(\Gamma)$ is the set of object ids type in Γ. Also, $q :: T[\mathcal{H}] \in \Gamma$ means that $q :: T[\mathcal{H}]$ is an element of Γ at some depth, *i.e.,*

$$q :: T[\mathcal{H}] \in \mathcal{H}' = \begin{cases} \text{if } \mathcal{H}' = \mathcal{H}_1, q :: T[\mathcal{H}], \mathcal{H}_2 \\ \text{if } \mathcal{H}' = \mathcal{H}_1, p :: T'[\mathcal{H}''], \mathcal{H}_2 \text{ and } q :: T[\mathcal{H}] \in \mathcal{H}'' \end{cases}$$

Given these helper functions, we can define ownership nesting relations using the nesting of the store typing.

$$\text{(GOOD-OWNER)}\quad \frac{\Gamma \vdash \diamond \qquad p \in \mathsf{defs}(\Gamma) \cup \{\texttt{world}\}}{\Gamma \vdash p}$$

$$\text{(IN-OWNER)}\quad \frac{\Gamma \vdash \diamond \qquad q :: T[\mathcal{H}] \in \Gamma \qquad p \in \mathsf{defs}(\mathcal{H})}{\Gamma \vdash p \prec^* q}$$

$$\text{(IN-REFL)}\quad \frac{\Gamma \vdash p}{\Gamma \vdash p \prec^* p} \qquad \text{(IN-OUTSIDE)}\quad \frac{\Gamma \vdash q \prec^* p}{\Gamma \vdash p \succ^* q}$$

By (GOOD-OWNER), any id that maps to a type in Γ can be used as an owner. By (IN-OWNER), p is inside q if the type of p is nested inside q. By (IN-REFL), inside is a reflexive relation.

Contexts

A context is a piece of the ordinary syntax with a hole, where the hole is denoted by $\langle\,\rangle$. Any store C or stack S can be factored in a number of ways into a context. A hole can be empty, contain a subheap, or a unique value. This is useful since it allows us to talk about a specific subcomponent of a larger structure.

We write $K_{\langle\,\rangle}$ for some stack-store pair with a hole. $K_{\langle\,\rangle}$ is defined thus:

$$\begin{aligned}
K_{\langle\,\rangle} &::= C_{\langle\,\rangle}; S \;\mid\; C; S_{\langle\,\rangle} \\
C_{\langle\,\rangle}, \mathrm{H}_{\langle\,\rangle} &::= \mathrm{H}_{\langle\,\rangle}, n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}] \;\mid\; \mathrm{H}_{\langle\,\rangle}, n \mapsto \mathsf{B}[\mathrm{H}] \;\mid \\
&\quad\;\; \mathrm{H}, n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}_{\langle\,\rangle}] \;\mid\; \mathrm{H}, n \mapsto \mathsf{C}[\mathrm{V}_{\langle\,\rangle} \mid \mathrm{M} \mid \mathrm{H}] \;\mid \\
&\quad\;\; \mathrm{H}, n \mapsto \mathsf{B}[\mathrm{H}_{\langle\,\rangle}] \;\mid\; \mathrm{H}, \langle\,\rangle \\
S_{\langle\,\rangle} &::= S_{\langle\,\rangle} \bullet \mathrm{V} \;\mid\; S \bullet \mathrm{V}_{\langle\,\rangle} \\
\mathrm{V}_{\langle\,\rangle} &::= \mathrm{V}_{\langle\,\rangle}, x \mapsto \langle\,\rangle \;\mid\; \mathrm{V}_{\langle\,\rangle}, x \mapsto v \;\mid\; \mathrm{V}, x \mapsto \mathsf{U}_n[\uparrow m \mid \mathrm{H}_{\langle\,\rangle}]
\end{aligned}$$

We write *e.g.,* $C = C\langle d\rangle$ to mean that C can be factored as some store $C_{\langle\,\rangle}$ with the hole containing d which can be an object, a borrowing block or a unique. We also use similar notation for holes in store-typing, but we will defer the introduction until later.

For convenience, we will sometimes write $C\langle \mathrm{H}\rangle_n$ to mean that H is part of a subheap of an object with id n instead of $C\langle n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}', \mathrm{H}]\rangle$.

Dynamic semantics

Joline's dynamic semantics is a big-step operational semantics. Starting configurations have the form $\langle C\,;\, S \,|\, s\rangle$ (remember expressions are also statements) and resulting configurations have the forms $\langle C\,;\, S \,|\, v\rangle$, where v is a value returned by an evaluated expression, or $\langle C\,;\, S\rangle$ for a statement that does not return a value.

Note that we have further simplified things by only allowing one argument and only local variables as receivers for method calls. Also, we only allow borrowing the contents local variables.

Statements

$$\textsc{(stat-local)}\quad \frac{\langle C\,;\, S \,|\, e\rangle \rightarrow \langle C'\,;\, S' \bullet \mathrm{V} \,|\, v\rangle}{\langle C\,;\, S \,|\, t\ x = e\rangle \rightarrow \langle C'\,;\, S' \bullet \mathrm{V}, x \mapsto v\rangle} \qquad \textsc{(stat-skip)}\quad \frac{}{\langle C\,;\, S \,|\, \texttt{skip}\rangle \rightarrow \langle C\,;\, S\rangle}$$

By (STAT-LOCAL), local variable declaration first evaluates the initial value and then adds a local variable to the top stack frame and stores the result of the evaluation in it. By (STAT-SKIP), the `skip` statement changes nothing.

Updating the value of a variable or field is written $\mathrm{V}[x \mapsto v]$. It is defined thus:

$$\mathrm{V}[x \mapsto v] = \begin{cases} \mathrm{V}', x \mapsto v, \mathrm{V}'', & \text{if } \mathrm{V} = \mathrm{V}', x \mapsto v', \mathrm{V}'' \\ \mathrm{V}, & \text{otherwise} \end{cases}$$

For stacks, we write $S[x \mapsto v]$ to mean $S' \bullet \mathrm{V}[x \mapsto v]$, where $S = S' \bullet \mathrm{V}$ (*i.e.,* we only update variables on the top frame).

$\mathrm{V}(x)$ is a variable lookup defined thus:

$$\mathrm{V}(x) = \begin{cases} v, & \text{if } \mathrm{V} = \mathrm{V}', x \mapsto v, \mathrm{V}'' \\ \bot, & \text{otherwise} \end{cases}$$

where $\bot$ means that the variable x is not defined in V. As for updating, we write $S(x)$ to mean $\mathrm{V}(x)$ where $S = S' \bullet \mathrm{V}$. We use x, y for local variables and f for field variables.

$$\textsc{(stat-expr)}\quad \frac{\langle C\,;\, S \mid e\rangle \rightarrow \langle C'\,;\, S' \mid v\rangle}{\langle C\,;\, S \mid e\rangle \rightarrow \langle C'\,;\, S'\rangle} \qquad \textsc{(stat-update-local)}\quad \frac{\langle C\,;\, S \mid e\rangle \rightarrow \langle C'\,;\, S' \mid v\rangle}{\langle C\,;\, S \mid x = e\rangle \rightarrow \langle C\,;\, S'[x \mapsto v]\rangle}$$

$$\textsc{(stat-update-field)}\quad \frac{\langle C\,;\, S \mid e\rangle \rightarrow \langle C_1\,;\, S_1 \mid v\rangle \qquad S_1(x) = \uparrow n \qquad C_1 = C_2\langle n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]\rangle}{\langle C\,;\, S \mid x.f = e\rangle \rightarrow \langle C_2\langle n \mapsto \mathsf{C}[\mathrm{V}[f \mapsto v] \mid \mathrm{M} \mid \mathrm{H}]\rangle\,;\, S_1\rangle}$$

By (STAT-EXPR), an expression can be treated as a statement by just forgetting its resulting value. Updating an lvalue is split into two different rules, (STAT-UPDATE-LOCAL) and (STAT-UPDATE-FIELD), matched in the type system by (STAT-UPDATE). They are straightforward, first the RHS expression is evaluated to obtain the value that will be stored in the variable or field. The value is then stored in the lvalue, replacing its old contents. Note the use of the hole in (STAT-UPDATE-FIELD). $C_1 = C_2\langle n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]\rangle$ is just factoring C_1 into the equivalent context $C_{2\langle\,\rangle}$ where the hole contains the object pointed to by x. We then update the value of f in the object's field component to the new value and the resulting store is the store in the resulting configuration.

$$\textsc{(stat-sequence)}\quad \frac{\langle C\,;\, S \mid s_1\rangle \rightarrow \langle C_1\,;\, S_1\rangle \qquad \langle C_1\,;\, S_1 \mid s_2\rangle \rightarrow \langle C_2\,;\, S_2\rangle}{\langle C\,;\, S \mid s_1 ; s_2\rangle \rightarrow \langle C_2\,;\, S_2\rangle}$$

By (STAT-SEQUENCE), sequences of statements are evaluated left to right and the resulting store-stack pair of one statement is the initial store-stack pair of the subsequent statement.

$$\textsc{(stat-block)}\quad \frac{\langle C\,;\, S \bullet \mathsf{nil} \mid s\rangle \rightarrow \langle C'\,;\, S' \bullet \mathrm{V}\rangle}{\langle C\,;\, S \mid \{\ s\ \}\rangle \rightarrow \langle C'\,;\, S'\rangle}$$

Block works as in Java or C++, except that shadowing local variables is not supported. Before we evaluate the statements in a block, we push an empty frame onto the stack. When the block exits, the contents of that frame are removed. Remember that frames in our terminology does not correspond to a specific method. Instead, a frame is tied to a block, which can be, but does not necessarily need to be, a method body.

$$\frac{\begin{array}{c}S_1 = S \bullet \mathrm{V}, x \mapsto \mathit{null}, y \mapsto v \\ \langle C\langle n \mapsto \mathsf{B}[\mathrm{H}[n/x]]\rangle_b \,;\, S_1 \mid s\rangle \to \langle C'\langle n \mapsto \mathsf{B}[\mathrm{H}']\rangle_b \,;\, S_2\rangle \\ S_2 = S' \bullet \mathrm{V}, x \mapsto v'', y \mapsto v' \qquad n \text{ is fresh}\end{array}}{\begin{array}{r}\langle C \,;\, S \bullet \mathrm{V}, x \mapsto \mathsf{U}_x[v \mid \mathrm{H}] \mid \texttt{borrow}\ x :: \texttt{unique}_b : c\langle\sigma\rangle\ \texttt{as}\ \langle\alpha\rangle\ y\ \texttt{in}\ \texttt{\{}\ s\ \texttt{\}}\rangle \to \\ \langle C' \,;\, S' \bullet \mathrm{V}, x \mapsto \mathsf{U}_x[v' \mid \mathrm{H}'[x/n]]\rangle\end{array}} \;(\textsc{stat-borrow})$$

Borrowing destroys the uniqueness wrapper and instead wraps its contents in a borrowing wrapper. As opposed to the uniqueness wrapper, there may be pointers to its contents from the stack, and such a pointer is created by moving the value of the pointer compartment of the unique into the borrowing variable on the stack. The variable originally containing the unique is updated with null. After the statements of the borrowing block have been executed, the borrowing wrapper is destroyed and replaced by a uniqueness wrapper and the resulting value is stored in the borrowed variable.

Expressions

$$\frac{S(x) = \mathsf{U}_x[\uparrow m \mid \mathrm{H}]}{\langle C \,;\, S \mid x\texttt{--}\rangle \to \langle C \,;\, S[x \mapsto \mathit{null}] \mid \mathsf{U}_{\mathrm{free}}[\uparrow m \mid \mathrm{H}[\mathrm{free}/x]]\rangle} \;(\textsc{expr-dread-local})$$

$$\frac{S(x) = \uparrow n \qquad C = C'\langle n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]\rangle \qquad \mathrm{V}(f) = \mathsf{U}_{n.f}[\uparrow m \mid \mathrm{H}']}{\langle C \,;\, S \mid x.f\texttt{--}\rangle \to \langle C'\langle n \mapsto \mathsf{C}[\mathrm{V}[f \mapsto \mathit{null}] \mid \mathrm{M} \mid \mathrm{H}]\rangle \,;\, S \mid \mathsf{U}_{\mathrm{free}}[\uparrow m \mid \mathrm{H}'[\mathrm{free}/n.f]]\rangle} \;(\textsc{expr-dread-field})$$

The rules for destructive reading of variables and fields are pretty straightforward. Whenever a unique is moved (given another owner), we apply a substitution that replaces all occurrences of the old owner, *i.e.,* the field or variable that stored it, x or $n.f$ above, with the owner free, to its entire contents. Since the result of a destructive read is a free value, there is no field or object that owns it. We denote this by giving it the special owner free.

To talk about a specific object in a store, we factor it into the object and its context. This is denoted $C = C'\langle n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]\rangle$ where C' is the context of the object $n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]$ somewhere in C.

Instantiation

$$\frac{\mathrm{V} = \{f \mapsto \mathit{null} \mid f \in \mathit{dom}(\mathcal{F}_c)\} \quad \mathrm{M} = \sigma_n^{\mathrm{free}}(\mathcal{M}_c) \quad \text{where } n \text{ is fresh}}{\langle C\,;\, S \mid \mathtt{new}\; p{:}c\langle\sigma\rangle\rangle \rightarrow \langle C\,;\, S \mid \mathsf{U}_{\mathrm{free}}[\uparrow n \mid n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathsf{nil}]]\rangle}\ (\textsc{expr-new})$$

In object instantiation, the object is given a fresh id and all its fields are initialised to $null$. The method component is the static definitions of the methods where the static owner names are substituted for the owners from object's type. The object is then wrapped inside a uniqueness wrapper with a pointer to the created object in its pointer compartment. Finally, the entire uniqueness wrapper is returned. Note that the owner of the object in the subheap is free. The owner specified by the programmer, p, becomes the bound of the unique (see the static semantics).

Owner cast

$$\frac{\langle C\,;\, S \mid e\rangle \rightarrow \langle C'\,;\, S' \mid \mathsf{U}_{\mathrm{free}}[\uparrow n \mid \mathrm{H}]\rangle \qquad C' = C''\langle p \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}']\rangle}{\langle C\,;\, S \mid (p)\ e\rangle \rightarrow \langle C''\langle p \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}', \mathrm{H}[p/\mathrm{free}]]\rangle\,;\, S' \mid \uparrow n\rangle}\ (\textsc{expr-lose-unique})$$

When losing uniqueness, the uniqueness wrapper is destroyed and its contents moved into the subheap of the new owner. A substitution is applied to change all occurrences of the owner name of the unique with the new owner. Finally, the value in the pointer compartment is returned.

Method call.

$$\frac{\begin{array}{c}\langle C\,;\, S \mid e\rangle \rightarrow \langle C_1\,;\, S_1 \mid v\rangle \qquad S_1(x) = \uparrow n \\ C_1 = C_2\langle n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]\rangle \qquad \mathrm{M}(md) = \lambda\, x :: t\, \lambda\, \alpha \prec^* q.\, s; \mathtt{return}\ e \\ \langle C_1\,;\, S_1 \bullet x \mapsto v \mid s[p/\alpha]\rangle \rightarrow \langle C_3\,;\, S_2 \bullet \mathrm{V}\rangle \\ \langle C_3\,;\, S_2 \bullet \mathrm{V} \mid e[p/\alpha]\rangle \rightarrow \langle C'\,;\, S' \bullet \mathrm{V}' \mid v'\rangle\end{array}}{\langle C\,;\, S \mid x.md\langle p\rangle(e)\rangle \rightarrow \langle C'\,;\, S' \mid v'\rangle}\ (\textsc{expr-call})$$

The owner polymorphic method is pretty straightforward. The static name of the owner parameter is substituted for the actual argument in the method body. The argument expression is evaluated, the value of the receiver variable is looked up and the method code fetched and executed with a new frame containing the argument value. That frame is popped on returning and the method's return value is the return value of the expression.

Table 5.4. Judgements for well-formed store typing.

$\mathcal{H} \vdash \diamond$	Good store type
$\mathcal{H} \vdash t$	Type t is well-formed under $\mathcal{H}$

5.1.4 Well-formed Store Typing

Recall the definition of the store typing on Page 71. Below, the rules for well-formed store typing are introduced and explained. Note that we assume that the ids of objects (and borrowing blocks and uniques) are unique in the store-typing (but do not clutter the formalism with the trivial constructs to enforce it).

Contexts

As in the stores, we use contexts also for the store typing. $\mathcal{H}_{\langle\ \rangle}$ denotes a piece of store typing with hole in it.

$$\mathcal{H}_{\langle\ \rangle} ::= \mathcal{H}, \langle\ \rangle \quad | \quad \mathcal{H}_{\langle\ \rangle}, n :: T[\mathcal{H}] \quad | \quad \mathcal{H}, n :: T[\mathcal{H}_{\langle\ \rangle}]$$

When necessary, we label holes with subscript object ids to indicate the position of the hole in $\mathcal{H}$. For example, $\mathcal{H}\langle\mathcal{H}'\rangle_p$ means that $\mathcal{H}'$ is nested *directly* inside the type of the object (or unique or borrowing block) with id p. We write $\mathcal{H} = \mathcal{H}'\langle\mathcal{H}''\rangle_p$ to mean that $\mathcal{H}$ can be factored as a context with $\mathcal{H}''$ in a hole in p. When $\mathcal{H}$ is defined, we write $\mathcal{H}\langle\mathcal{H}'\rangle_p$ to mean that $\mathcal{H}'$ is added to $\mathcal{H}$ inside object p.

A special case of the subscript notation exists for `world`. When the label is `world`, *e.g.,* $\mathcal{H}\langle\mathcal{H}'\rangle_{\texttt{world}}$, this means that $\mathcal{H}'$ is in a hole at the top-level in $\mathcal{H}$, *i.e.,* $\mathcal{H}_{\langle\ \rangle} = \mathcal{H}'', \langle\ \rangle$.

The well-formed store typing judgments are described in Table 5.4. Initially, the store typing is empty, denoted nil.

(STORE-TYPE-EMPTY)

$$\frac{}{\mathsf{nil} \vdash \diamond}$$

(STORE-TYPE)

$$\frac{\mathcal{H} \vdash T \ \vee \ (T = c\langle\sigma\rangle \wedge \mathcal{H} \vdash p : c\langle\sigma\rangle)}{\mathcal{H}\langle n :: T\rangle_p \vdash \diamond}$$

By (STORE-TYPE-EMPTY), the empty store typing is well-formed. By (STORE-TYPE), a store type is well-formed if all types it contains are well-formed; a id-type mapping can be added into an existing type in a well-formed store type if the type added is well-formed with respect to the store type. Remember, the LHS of the turn-stile means that $n :: T$ is a id-type map directly nested inside p in $\mathcal{H}$.

Table 5.5. Judgements for well-formed configurations.

$\Gamma; F \vdash \langle C\,;\, S\rangle$	$\langle C\,;\, S\rangle$ is a well-formed configuration
$\Gamma; F \vdash \langle C\,;\, S \mid e\rangle :: t$	$\langle C\,;\, S \mid v\rangle$ is a well-formed config. of type t
$\Gamma; F \vdash \langle C\,;\, S \mid s\rangle; E$	$\langle C\,;\, S \mid s\rangle$ is a well-formed configuration
$\Gamma; F \vdash \langle C\,;\, S \mid v\rangle :: t$	$\langle C\,;\, S \mid v\rangle :: t$ is a well-formed config. of type t
$\Gamma; F \vdash C; S$	$C; S$ is a well-formed store-stack pair
$\mathcal{H}; p \vdash \mathrm{H} \gg \mathcal{H}'$	H is a well-formed subheap/object/borrowing block, owned by p, typed by $\mathcal{H}'$
$\Gamma; F \vdash S$	S is a well-formed stack
$\Gamma; E \vdash \mathrm{V}$	V is a well-formed frame
$\Gamma \vdash v :: t$	v is a well-formed value of type t

The helper function $\mathcal{P}_c$ returns the set of owner parameters and their relations used in class c and is defined thus:

$$\mathcal{P}_c = \begin{cases} \{\alpha_i \, \mathsf{R}_i \, p_{i=1..n}\}, & \text{if } \texttt{class } c\langle \alpha_i \, \mathsf{R}_i \, p_{i=1..n}\rangle \cdots \in P \\ \bot, & \text{otherwise} \end{cases}$$

where $\bot$ means that the class c is not defined in P.

$$\text{(TYPE)}\quad \frac{\mathcal{H} \vdash \diamond \qquad \mathcal{H} \vdash \sigma^p(\alpha \, \mathsf{R} \, q) \qquad \text{for all } \alpha \, \mathsf{R} \, q \in \mathcal{P}_c}{\mathcal{H} \vdash p : c\langle \sigma\rangle} \qquad \text{(TYPE-WRAPPER)}\quad \frac{\mathcal{H} \vdash \diamond \qquad T \in \{\mathfrak{B}, \mathfrak{U}\}}{\mathcal{H} \vdash T}$$

By (TYPE), a type is well-formed if the owners used to form it satisfy nesting relations between the owner parameters specified in the class. By (TYPE-WRAPPER), the type of a unique or borrowing block is well-formed if the store typing is well-formed.

5.1.5 Well-formed Configuration

The rules for well-formed configuration might seem somewhat unorthodox at first. Basically, we pull a piece of typing information out of nowhere and use it to type the right hand side of the turns-tile (see for example $\mathcal{H}$ in (STORE)). However, we force this information to match the structure of the heap.

Some of the rules are on the form $\Gamma; p \vdash \mathrm{H} \gg \mathcal{H}$. The $\mathcal{H}$ to the right of the $\gg$ symbol is the store typing information for the visible parts of the subheap H. For $\Gamma; p \vdash \mathrm{H} \gg \mathcal{H}$ we say that "H is typed by $\mathcal{H}$".

As the previous sentence hints, visibility restrictions are built into the well-formed configuration rules. The store typing information on the LHS of the turns-tile is the information visible to the objects in the subheap on the RHS. When "validating" an object, its type information is added to the LHS when validating its subheap, making the object visible to its own representation.

Notably, typing information for uniques (and the types of objects nested inside uniques) is never made visible outside the uniques. This is a pivotal restriction that makes it unsound for an object outside of a unique to reference an object inside it.

At for store type information, we assume that objects ids are unique and do not clutter the formalism with the trivial rules to enforce it.

Configurations

$$\frac{\Gamma; F \vdash C; S}{\Gamma; F \vdash \langle C\,;\, S\rangle}\ (\textsc{config}) \qquad \frac{\Gamma; F \bullet E \vdash C; S \quad E \vdash e :: t}{\Gamma; F \bullet E \vdash \langle C\,;\, S \,|\, e\rangle :: t}\ (\textsc{config-expr}) \qquad \frac{\Gamma; F \bullet E \vdash C; S \quad E \vdash s; E'}{\Gamma; F \bullet E \vdash \langle C\,;\, S \,|\, s\rangle; E'}\ (\textsc{config-stat})$$

By (CONFIG), (CONFIG-EXPR) and (CONFIG-STAT), a configuration is well-formed if its store and stack are well-formed and its optional expression/statement is well-formed with respect to the static type information of the top-most frame.

$$\frac{\Gamma; F \bullet E \vdash C; S \qquad \begin{cases} \Gamma\langle \text{free} :: \mathfrak{U}\rangle_b \vdash v :: \texttt{unique}_{\text{free}} : c\langle\sigma\rangle, & \text{if } t = \texttt{unique}_b : c\langle\sigma\rangle \\ \Gamma \vdash v :: t, & \text{otherwise} \end{cases}}{\Gamma; F \vdash \langle C\,;\, S \,|\, v\rangle :: t}\ (\textsc{config-val})$$

(CONFIG-VAL) deals with resulting configurations, containing a value returned from an expression. If the value is free, we require that it is well-formed with respect to its bound, b (note that we are inserting the type information for the uniqueness wrapper, which will contain the types of all objects in the subheap of the free into the type of object b). Since the typing information of the free is not in Γ, clearly no references to the free can exist in $C; S$. If the value is not free, it must be well-formed with respect to the store typing.

Store-stack pair

$$\frac{\mathcal{H}; \texttt{world} \vdash C \gg \Gamma \qquad \Gamma; F \vdash S}{\Gamma; F \vdash C; S}\ (\textsc{store-stack})$$

By (STORE-STACK), a store-stack pair is well-formed if the store and stack components are well-formed with respect to the store typing. The $\mathcal{H}$ is “pulled out of thin air”, but is required to be a subset of Γ (see (SUBHEAP)). Remember that we write Γ for a store typing that types an entire store (and not $\mathcal{H}$).

Stores

$$\frac{\mathcal{H} \vdash n}{\mathcal{H}; n \vdash \mathsf{nil} \gg \mathsf{nil}} \quad (\text{SUBHEAP-EMPTY})$$

By (SUBHEAP-EMPTY), an empty subheap inside n in $\mathcal{H}$ is well-formed and is typed by an empty store type if n is a good owner under Γ.

The objdom helper function returns the set of objects defined in a subheap, not including borrowing blocks:

$$\mathsf{objdom}(\mathrm{H}) = \begin{cases} \mathsf{objdom}(\mathrm{H}') \cup \{n\}, & \text{if } \mathrm{H} = \mathrm{H}', n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}''] \\ \mathsf{objdom}(\mathrm{H}'), & \text{if } \mathrm{H} = \mathrm{H}', n \mapsto \mathsf{B}[\mathrm{H}''] \\ \emptyset, & \text{if } \mathrm{H} = \mathsf{nil} \end{cases}$$

$$\frac{\begin{array}{cc} \mathcal{H}; p \vdash \mathrm{H}_i \gg \mathcal{H}_i \text{ for all } i \in 1..n & \begin{cases} \mathcal{H} = \mathcal{H}', & \text{if } p = \texttt{world} \\ p :: T[\mathcal{H}'] \in \mathcal{H}, & \text{otherwise} \end{cases} \\ \mathsf{defs}(\mathcal{H}') = \mathsf{objdom}(\mathrm{H}_{i \in 1..n}) & \end{array}}{\mathcal{H}; p \vdash \mathrm{H}_{i \in 1..n} \gg \mathcal{H}_{i \in 1..n}} \quad (\text{SUBHEAP})$$

By (SUBHEAP), a subheap $\mathrm{H}_{i\in 1..n}$ inside p in Γ is well-formed and typed by $\mathcal{H}_{i\in 1..n}$ if all p objects and borrowing blocks H_i in the subheap are well formed and are typed by $\mathcal{H}_i$. Also, the type information in p must be equivalent to the type information of all objects (but not borrowing block) at top-level in $\mathrm{H}_{i\in 1..n}$.

The use of objdom in (SUBHEAP) is important in that it establishes the relation between the type information pulled out of thin air and the resulting type information. Consider (STORE-STACK) on the previous page. There, $\mathcal{H}$ appears magically on the LHS of the turnstile. However, in (SUBHEAP), we see that the contents of $\mathcal{H}$ must match the objects at top-level in C. The rules for well-formed object, unique and borrowing block will further require that the types in $\mathcal{H}$ match the types of the top-level objects. Thus, to be valid, the $\mathcal{H}$ must be a subset of Γ in (STORE-STACK).

Objects and borrowing blocks

$$\frac{\begin{array}{c} \Gamma(m) = p{:}c\langle\sigma\rangle \qquad \Gamma \vdash p{:}c\langle\sigma\rangle \qquad \Gamma' = \Gamma\langle\mathcal{H}\rangle_m \\ \sigma^p_m(\mathcal{M}_c) = \mathrm{M} \qquad \Gamma'; m \vdash \mathrm{H} \gg \mathcal{H}' \qquad t_i = \sigma^p_m(\mathcal{F}_c(f_i)) \\ \forall\, i \in 1..n \begin{cases} \Gamma'\langle m.f_i :: \mathfrak{U}\rangle_{b_i} \vdash v_i :: \texttt{unique}_{m.f_i}{:}c_i\langle\sigma_i\rangle, & \text{if } t_i = \texttt{unique}_{b_i}{:}c_i\langle\sigma_i\rangle \\ \Gamma' \vdash v_i :: \sigma^p_m(\mathcal{F}_c(f_i)), & \text{otherwise} \end{cases} \end{array}}{\Gamma; p \vdash m \mapsto \mathsf{C}[f_i \mapsto v_i\ {}_{i\in 1..n} \mid \mathrm{M} \mid \mathrm{H}] \gg m :: c\langle\sigma\rangle[\mathcal{H}']} \quad (\text{OBJECT})$$

By (OBJECT), an object $m \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]$ is well-formed under Γ at p and is typed by $m :: c\langle\sigma\rangle[\mathcal{H}']$ if it has the type $p : c\langle\sigma\rangle$ in Γ, its type is well-formed, the values in all its fields are well-formed, its subheap is well-formed and is typed by $\mathcal{H}'$, and its method component corresponds with the static method definitions but with the actual owners from the object's type. For a subtle reason, we cannot unify the validation of fields and (GOOD-FRAME); in the added type information for uniques values, the ids must correspond to the id of the object plus field-name, since this is the unique owner of the contents of the uniques.

$$\frac{\mathcal{H}\langle n :: \mathfrak{B}[\mathcal{H}']\rangle_p; n \vdash \mathrm{H} \gg \mathcal{H}''}{\mathcal{H}; p \vdash n \mapsto \mathsf{B}[\mathrm{H}] \gg n :: \mathfrak{B}[\mathcal{H}'']} \quad \text{(BORROWING-BLOCK)}$$

By (BORROWING-BLOCK), a borrowing block is well-formed under $\mathcal{H}$ and is typed by $n :: \mathfrak{B}[\mathcal{H}'']$ if its subheap is well-formed and typed by $\mathcal{H}''$ under the store type constructed by adding $n :: \mathfrak{B}[\mathcal{H}']$ to $\mathcal{H}$.

The contents of a borrowing block should not be visible outside itself (excluding the stack). In (BORROWING-BLOCK), the borrowed type is added to $\mathcal{H}$, meaning that it was not already in $\mathcal{H}$ and not visible to anything typed against it. Since only the stack and the contents of the borrowing block is typed against this extended $\mathcal{H}$, external references to objects inside the borrowing block from places other than the stack are impossible in out system.

Stacks and frames

$$\frac{\Gamma \vdash \diamond}{\Gamma; \emptyset \vdash \mathsf{nil}} \quad \text{(STACK/FRAME-EMPTY)} \qquad \frac{\Gamma; F \vdash S \qquad \Gamma; E \vdash \mathrm{V}}{\Gamma; F \bullet E \vdash S \bullet \mathrm{V}} \quad \text{(STACK)}$$

By (STACK/FRAME-EMPTY), an empty stack or empty frame is well-formed. The $\emptyset$ indicates that the piece of static type information associated with the stack/frame is empty, *i.e.,* it is not supposed to contain any frames or variables. By (STACK), a stack is well-formed if all its frames are well-formed.

$$\frac{\Gamma; E \vdash \mathrm{V} \qquad \begin{cases} \Gamma\langle x :: \mathfrak{U}\rangle_b \vdash v :: \mathtt{unique}_x : c\langle\sigma\rangle, & \text{if } t = \mathtt{unique}_b : c\langle\sigma\rangle \\ \Gamma \vdash v :: t, & \text{otherwise} \end{cases}}{\Gamma; E, x :: t \vdash \mathrm{V}, x \mapsto v} \quad \text{(FRAME)}$$

By (FRAME), a frame is well-formed if the value of all its variables are well-formed. Note the adding of $x :: \mathfrak{U}$ to Γ, similar to (CONFIG-VAL) above.

Values

$$\frac{\Gamma\langle\mathcal{H}\rangle_n \vdash v :: n\!:\!c\langle\sigma\rangle \qquad n \notin rng(\sigma) \qquad \Gamma\langle\mathcal{H}\rangle_n; n \vdash \mathrm{H} \gg \mathcal{H}'}{\Gamma \vdash \mathsf{U}_n[v \mid \mathrm{H}] :: \mathtt{unique}_n\!:\!c\langle\sigma\rangle} \text{(UNIQUE)}$$

By (UNIQUE), a unique is well-formed under $\mathcal{H}$ if its id corresponds to the bound of its type, the value of its pointer component is well-formed and its type is the bound, class and owner parameters from the unique's type, the id of the unique is not in the set of owner parameters, and its subheap is well-formed.

Note that the type information for a unique and its nested objects are never in Γ, meaning that external references to a unique or its contents are impossible.

$$\frac{\Gamma \vdash t}{\Gamma \vdash null :: t} \text{(VAL-NULL)} \qquad \frac{\Gamma(n) = p\!:\!c\langle\sigma\rangle}{\Gamma \vdash \uparrow\! n :: p\!:\!c\langle\sigma\rangle} \text{(VAL-PTR)} \qquad \frac{\Gamma \vdash v :: t \qquad \Gamma \vdash t \leq t'}{\Gamma \vdash v :: t'} \text{(VAL-SUBSUMPTION)}$$

By (VAL-NULL), $null$ can have any well-formed type. By (VAL-PTR), a pointer is well-formed and has type t if it typed t in Γ. By (VAL-SUBSUMPTION), subsumption applies to values.

5.2 Proof Statements

In this section, we state theorems and sketch the proofs of type soundness, along with the important structural properties that gives us owners-as-dominators and external-uniques-as-dominating-edges.

For convenience, we define the binary relation $\#$ on sets to mean that they are disjoint in all their elements, *i.e.,*

$$A \# B \iff A \cap B = \emptyset$$

where A and B are sets.

5.2.1 Dominance Properties

These structural properties capture both owners-as-dominators and external-uniqueness-as-dominating-edges. In a well-formed configuration, there are no external references to an object, except for from its owner. We model this using holes—in a well-formed configuration that can be factored as a configuration plus a hole containing an object, borrowing block or unique, there are no

Table 5.6. Definition of uses; the set of pointers used in a store/stack.

$$
\begin{aligned}
\mathsf{uses}(C;S) &= \mathsf{uses}(C) \cup \mathsf{uses}(S)\\
\mathsf{uses}(S \bullet \mathrm{V}) &= \mathsf{uses}(S) \cup \mathsf{uses}(\mathrm{V})\\
\mathsf{uses}(\mathrm{V}, x \mapsto v) &= \mathsf{uses}(\mathrm{V}) \cup \mathsf{uses}(v)\\
\mathsf{uses}(\mathrm{H}, n \mapsto \mathsf{B}[\mathrm{H}']) &= \mathsf{uses}(\mathrm{H}) \cup \mathsf{uses}(\mathrm{H}')\\
\mathsf{uses}(\mathrm{H}, n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}']) &= \mathsf{uses}(\mathrm{H}) \cup \mathsf{uses}(\mathrm{V}) \cup \mathsf{uses}(\mathrm{H}')\\
\mathsf{uses}(\mathsf{U}_n[\uparrow m \mid \mathrm{H}]) &= \{m\} \cup \mathsf{uses}(\mathrm{H})\\
\mathsf{uses}(\uparrow m) &= \{m\}\\
\mathsf{uses}(\mathsf{nil}) &= \emptyset
\end{aligned}
$$

Table 5.7. Definition of defs; the set of ids of all objects defined in a store/stack.

$$
\begin{aligned}
\mathsf{defs}(C;S) &= \mathsf{defs}(C) \cup \mathsf{defs}(S)\\
\mathsf{defs}(S \bullet \mathrm{V}) &= \mathsf{defs}(S) \cup \mathsf{defs}(\mathrm{V})\\
\mathsf{defs}(\mathrm{V}, x \mapsto v) &= \mathsf{defs}(\mathrm{V}) \cup \mathsf{defs}(v)\\
\mathsf{defs}(\mathrm{H}, n \mapsto \mathsf{B}[\mathrm{H}']) &= \mathsf{defs}(\mathrm{H}) \cup \mathsf{defs}(\mathrm{H}')\\
\mathsf{defs}(\mathrm{H}, n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}']) &= \mathsf{defs}(\mathrm{H}) \cup \mathsf{defs}(\mathrm{V}) \cup \mathsf{defs}(\mathrm{H}') \cup \{n\}\\
\mathsf{defs}(\mathsf{U}_n[\uparrow m \mid \mathrm{H}]) &= \mathsf{defs}(\mathrm{H})\\
\mathsf{defs}(\uparrow m) &= \emptyset\\
\mathsf{defs}(\mathsf{nil}) &= \emptyset
\end{aligned}
$$

references from the objects outside the hole to the *contents* of the object in the hole. If the contents of the hole is a unique value, there are also no references from the stack to the unique's contents. From now on, we make heavy use of the helper functions uses and defs, to capture the set of object ids pointed to respective the set of ids of all objects defined in some store/stack. (See Table 5.6 and Table 5.7 for the complete definitions.)

Theorem 5.1. (Dominance Properties)
Assume $\Gamma; F \vdash C; S$. *Then:*

1. *If* $C = C'\langle n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]\rangle$ *or* $C = C'\langle n \mapsto \mathsf{B}[\mathrm{H}]\rangle$, *then* $\mathsf{uses}(C') \# \mathsf{defs}(H)$.
2. *If* $S = S'\langle n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]\rangle$ *or* $S = S'\langle n \mapsto \mathsf{B}[\mathrm{H}]\rangle$, *then* $\mathsf{uses}(C) \# \mathsf{defs}(H)$.
3. *If* $C; S = K\langle \mathsf{U}_n[\uparrow m \mid \mathrm{H}]\rangle$, *then* $\mathsf{uses}(K) \# (\mathsf{defs}(\mathrm{H}) \cup \{n\})$.

Proof. (Outline) The first two cases deal with objects or borrowing blocks in the store and stack. $\mathsf{uses}(C')$ is the set of all pointer values stored in fields in all objects in C'. Since $\mathsf{uses}(C')$ is disjunct from the set of objects defined in

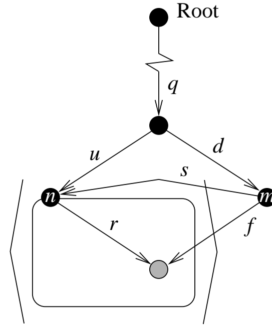


Fig. 5.2. Possible paths. The $\langle\ \rangle$ denotes a hole in the store with n and its subheap as its contents. n is an object or a uniqueness wrapper.

H in the hole, we see that there are no external pointers to the contents of an object. (Note that n is not included in $\mathsf{defs}(\mathrm{H})$.) The third case deals with unique values and imposes a stronger restriction—no object nested inside the unique or the unique is in the set of used pointer values. Thus, there are no references to the unique's content nor to the unique from any place else in the stack or the store.

Owners-as-Dominators

The owners-as-dominators-property (OAD) states that for any path to an object from the root of the object hierarchy, the path must contain the object's owner, *i.e.,* the owner is a dominating node for all objects nested inside it.

In other words, the OAD can be explained as *no objects belonging an to object n's representation are referenced from outside n*. If this is true, all paths to the object must pass through the owner at some point.

Thus, if for all $\Gamma; F \vdash K$ such that $K = C\langle n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]\rangle; S$ or $K = C; S\langle n \mapsto \mathsf{C}[\mathrm{V} \mid \mathrm{M} \mid \mathrm{H}]\rangle$ implies that $\mathsf{uses}(C) \# \mathsf{defs}(\mathrm{H})$, then that satisfy the OAD property.

In plain language, this means that the set of ids referred to by all fields in C and the set of ids of the objects defined in H are disjunct, *i.e.,* there are no pointer to the objects in H anywhere in C. Thus, the only possible external pointers to objects in H are from V, *i.e.,* from (inside) its owner, n.

As an example, consider the picture in Figure 5.2 above. There are three possible paths to the gray 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 gray object go via n, n is a dominating node for it, meaning that OAD is satisfied.

External-Uniques-as-Dominating-Edges

External-uniqueness-as-dominating-edges (EUADE) states that a uniquely referenced object is a dominating edge to the objects nested inside it. This means that for each path to an object inside a unique from the root, the path must contain the reference to the unique.

In a similar fashion as for OAD, Theorem 5.1 gives us EUADE. For any well-formed configuration, $\Gamma; F \vdash K\langle \mathsf{U}_n[v \mid \mathrm{H}]\rangle$, $\mathsf{uses}(K) \,\#(\mathsf{defs}(\mathrm{H}) \cup \{n\})$, *i.e.,* the set of all ids of all objects referenced by fields and variables in K (outside the unique, including the stack) is disjunct from the set of objects nested in the unique *and the id of the unique*. Thus, if u is the externally unique pointer to n, not only is the the path $q \to d \to f$ in the picture in Figure 5.2 invalid, but so is also $q \to d \to s \to r$ since m may not reference n. Now, since there can be only one pointer to a unique, the unique is clearly a dominating edge of the gray object.

□

Type soundness

Type soundness is proved as a subject reduction theorem, *i.e.,* types are preserved throughout the evaluation of a program. In combination with the dominance properties, subject reduction gives us the owners-as-dominators property and the external-uniques-as-dominating-edges property since it proves that configurations are well-formed between evaluation of statements.

Our subject reduction theorem states that if a well-formed configuration evaluates successfully, it produces another configuration that is well-formed with respect to some extended store-type environment. Movement does not produce any visible changes to the store-typing since unique types are not present in Γ.

Theorem 5.2. (Subject reduction)
If $\Gamma; F \bullet E \vdash \langle C\,;\, S \,|\, s\rangle; E'$ *and* $\langle C\,;\, S \,|\, s\rangle \to \langle C'\,;\, S'\rangle$*, then there exists* Γ' *such that* $\Gamma'; F \bullet E' \vdash \langle C'\,;\, S'\rangle$*, where* $\Gamma \subseteq \Gamma'$.

Proof. (Outline) The proof is by structural induction over the Joline's statements and expressions. The key to soundness is the hiding of the typing of unique variables in Γ since these types may change during evaluation.

If the store-typing for all uniques (and their contents) are not visible to external objects, then no external objects are affected by movement since they cannot witness the change of type. For all internal objects, the internal view of the moving object is unchanged by the movement operation and the only necessary operation to keep things sound is to replace any occurrence

of the old owner (if owner was used to form types inside the unique) with the new one, which is a trivial operation. For owner-polymorphic methods and borrowings, the store-typing is temporarily extended during evaluation. However, the ownership ordering and the dominance properties guarantees that values typed by these extensions never escape from the stack frames or borrowing blocks in point. Thus, we can simply remove the extension from the store-type as we pop the frame or remove the borrowing block from the stack since this contains all values that use the additional type information.

Modulo these concerns, the proof of subject reduction is straight forward. □

5.3 Concluding Remarks

This concludes the technical presentation of Joline. We save generational ownership and the full proofs for the final PhD thesis. Confident that our system is sound, we move on to describe some applications for external uniqueness.

6

Applications and Extensions

This chapter presents examples of applications for external uniqueness as well as possible extensions.

For the applications, we show how external uniqueness together with our proposed supporting constructs enables transfer of ownership (*e.g.,* moving objects between representations) and merging representations in the presence of ownership types and how we can use our borrowing construct, scoped regions and owner-polymorphic methods to simulate various notions of borrowing. We also show “movable aliased objects”, non-unique objects with all the benefits of unique objects that can be encoded using external uniqueness and how we overcome the intialisation problem and allow external initialisation of an object’s representation.

Many of these examples were previously not possible to encode in a system with ownership types.

6.1 Applications for External Uniqueness

External uniqueness has virtues other than overcoming the abstraction problem. As an extension to ownership types allowing transfer of ownership, we can encode object pools or merge several objects’ representations, which was not previously possible. Transfer of ownership is also useful in overcoming the initialisation problem pointed out by *e.g.,* Detlefs et al. (1998).

6.1.1 Transfer of Ownership

Transfer of ownership is an important design pattern in concurrent object-oriented programming (Lea 1998). Ownership of an object is transferred from one object to another and the first object must release all its references to the moving object.

```
class TokenRing
{
  owner:TokenRing next;  // sibling
  unique:Token token;

  void give()
  {
    next.receive(token--);
  }

  void receive(unique:Token tkn)
  {
    token = tkn--;
  }
}
```

Fig. 6.1. A Token ring implementation.

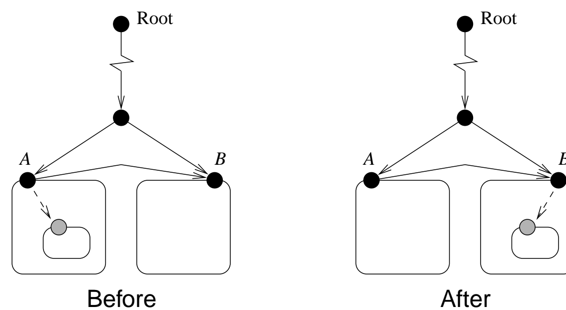


Fig. 6.2. Transfer ownership of the gray object from A (left) to the sibling B (right).

Figure 6.1 shows the implementation of a token ring. The token object is passed from one object to the next by calling the `give` method. Figure 6.2 illustrates the move of the grey `Token` from `TokenRing` element A to B. The movement bound of the moving token must be outside the target `TokenRing` element. The movement bound must also be outside A. Otherwise there could be residual aliasing from the grey token to A which would become unsound during a move. If the movement bound is outside A, any alias from the token or any object internal to the token are references to objects outside A which makes them valid even if moved to B. B can be the movement bound or some owner below the movement bound.

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

We now examine other examples where transfer of ownership is beneficial.

Object Pools

Using object pools is a well-known optimisation technique. In systems where many short-lived objects are needed to perform some task, instead of frequently creating objects, using them and then disposing of them, tasks which are expensive, a pool of possibly initialised objects is used. “New objects” are taken from the pool when needed, and later returned there instead of disposed.

Unless objects taken from object pools should not be encapsulated, previous ownership types systems have a severe limitation with regard to object pools since owners are fixed for life. To use the object pool for different objects with different owners would require one object pool per owner, which is likely to be suboptimal under most designs.

With external uniqueness, we can encode object pools where the objects in the pool are unique. An object can be taken from the pool and moved into the appropriate owner. When discarded, the object could be simply moved back into the object pool unless the object had lost its uniqueness or had its movement bound changed in a way that prevented it from being moved back into the pool.

Avoiding Unnecessary Synchronisation

As we have previously stated, uniqueness can be used to avoid unnecessary synchronisation. Making sure that an object is thread-local (or confined to a single thread) is easy with uniqueness since there is only one pointer to the object. It is also easy to realise that movement of an object between threads without risking residual aliasing is trivial. However, as we have continuously argued, as traditional uniqueness only applies to a single object, moving that from one thread to another might not be sufficient since references to its (conceptually) internal objects might still exist in the first thread. Thus, the aggregate is split between threads, which is likely to be the opposite of the intention. Being a shallow property, uniqueness cannot solve this problem, other than by forcing all references to internal objects in an aggregate to be unique. With this constraint, moving the bridge object will implicitly move the entire aggregate. However, it precludes internal sharing which might not be possible or desirable, and the abstraction problem persists.

External uniqueness overcomes this limitation (as is done also in Parameterised Race-Free Java (Boyapati and Rinard 2001; Boyapati, Lee, and Rinard 2002) but perpetuating the abstraction problem and neither allowing the existence of back pointers nor the existence of sibling objects). Since an externally

unique reference is a dominating edge, moving it from one thread to another implicitly moves all its internal objects along with it. Thus, if an externally unique object is moved to a thread, there is no need for synchronisation on any method, except if the receiver is an external object. This is a powerful consequence.

6.1.2 Merging Representations

External uniqueness enables the protected, self-referential internals of two or more data structures to be merged *without copying* into a new data structure. This offers the same degree of protection *without* any residual aliasing from the original data structures and *without* the increased performance penalties and complexity required for copying. While merging by copying might sometimes be possible, in the presence of internal sharing, it is likely to cause inconsistencies. At least, it will break the internal structure of the object since the original objects and not their copies will be referenced, unless all the objects in point are implemented with this in mind.

Merging representations was previously impossible in the presence of deep ownership since owners were fixed for life. Merging entire representations, or parts of representations is a common enough task that preventing it greatly impacts the usefulness of the proposals. In previous systems with deep ownership, merging the sets of links of two lists without resorting to copying required that the lists shared a common representation or that the links were not part of the lists' representations' and shared a common owner. The former is not allowed and the latter weakens encapsulation since external objects are able to access the lists' representations.

As an example of merging representations in Joline, Figure 6.3 shows the merging of one doubly-linked list into another. The `append` method of the first list is invoked with the second list as argument. The phases of the operation are (in order):

- `*` The first list borrows its own head link (*i.e.,* the entire list) using a temporary owner (here `ho`).
- `**` The head link of the other list is moved into a variable owned by `ho`, *i.e.,* the two lists now share a common owner.
- Remaining code: the merge is performed, in this case an append, and then the `head` variable is reinstated with the resulting value of `bh`. Note that `bh` may have been set in line `***`. This illustrates what we have earlier stated about borrowing: when a borrowing ceases, there is only one reference into the aggregate, not necessarily the same as the one originally borrowed. This is consistent with external uniqueness.

```
class Link<data>
{
  data:Object data;
  owner:Link<data> next, prev;
}

class List<data>
{
  unique:Link<data> head;

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

Fig. 6.3. Merging two doubly-linked lists in a JLL. `ho` is the temporary owner of the list head while borrowed.

Note that `other.head` is consumed in this operation, *i.e.,* after merge, the second list is empty.

David Holmes posed this example as a challenge when he saw the original ownership types proposal (Clarke, Potter, and Noble 1998). No existing combination of uniqueness and ownership types can handle it, but we finally do so in an elegant manner.

6.1.3 Simulating Borrowing and Orthogonality of Concepts

To recapitulate, borrowing was introduced in previous uniqueness proposals to tackle slipperiness. With borrowing, unique references could be operated upon without consuming their targets and or having to manually reinstate the

```
class Example extends Object
{
  <borrowed inside this> void method(borrowed:BlackBox bb)
  {
    ...
  }
}

void example(world Example e)
{
  unique:BlackBox<> bb = new unique BlackBox<>();
  borrow bb as <temp> b
  {
    e.method<temp>(b); // pass borrowed owner and reference
  }
}
```

Fig. 6.4. Passing a borrowed object as argument to a method. The method `method` in class `Example` is given a temporary permission to reference the owner `temp`, created in the method `example`. Thus, the reference to the borrowed black box in `b` can be passed as a parameter to `method`.

values after using them. Existing uniqueness proposals impose the restriction that a borrowed reference cannot be stored in the field of an object, making borrowed references a kind of second-class citizens which are neither orthogonal to unique references nor non-unique references. These proposals lack mechanisms in their type systems to treat borrowed references as usual non-unique references but maintain the uniqueness invariant between borrowings.

The borrowing we propose here is of a different kind. Rather than introduce borrowed references as a special kind of reference, we introduce a borrowing construct that allows an externally unique object to become a regular, non-unique object temporarily. While borrowed, an object is a regular non-unique object and thus suffers from no unnecessary restrictions. Any operations that can normally be carried out on a non-unique object can be carried out on a borrowed reference. Ownership types guarantees that any aliases created are temporary or appropriately contained. The result is a cleaner language which employs only orthogonal constructs (unique, non-unique) in a flexible manner; there isn't a borrowed keyword that needs to be propagated through the system, and no need for borrowing annotations, just the borrowing construct.

Pass a Borrowed Object as an Argument

Our borrowing takes a unique reference, converts it to a non-unique with the ability to convert it back, and places the non-unique reference on the stack.

```
void example(unique BlackBox bb)
{
  borrow bb as <temp> b
  { // 1st block
    temp:List<temp> list = new temp:List<temp>;
    list.add(bb);
    ...
    (scoped)
    { // 2nd block, nested in 1st
      scoped:List<temp> list = new scoped:List<temp>;
      list.add(bb);
      ...
    }
    ... // (*)
  }
  ... // (**)
}
```

Fig. 6.5. Storing a borrowed reference on the heap. The method creates a temporary owner `scoped`, and uses that as an owner for a list object in the innermost block.

Since the owner of the non-unique reference is local, no object in the system (except the borrowed object, and objects internal to it) has the necessary permission to create an alias to it.

To simulate the kind of borrowing found in traditional uniqueness systems we use the owner polymorphic methods introduced in a previous chapter (Section 3.2.2). Any owner can be borrowed, even owners not corresponding to a particular object. Thus, we can pass the permission to access a borrowed object to an owner polymorphic method along with the borrowed reference. Since the permission did not exist before the borrowing, no previously existing external objects' types are parameterised with the permission and thus cannot store a permanent reference to the object on the heap.

The owner polymorphic method can use its owner parameter to instantiate objects that may store references to the borrowed object on the heap in the same fashion as discussed immediately below. However, any such object will be invalidated before the borrowing exits, since the appropriate owners are out of scope and the necessary types cannot be named.

Store a Borrowed Object on the Heap

A borrowed reference can be stored on the heap in any object that has a permission to reference it. As was shown previously when the borrowing construct was introduced, a temporary owner of the the borrowed object is defined for a particular scope. This owner can then be used as a regular owner

to create objects. These will have permission to access the borrowed object and store it in a field. An example of this is shown in Figure 6.5. The example also shows a scoped region used to create an additional owner `scoped`, inside `temp`, defined for a block nested inside the borrowing block.

When the second, innermost, block exits, the `scoped` owner goes out of scope an cannot be named. Thus, in the code to follow, the lines marked `(*)` and `(**)`, the type cannot be formed and thus no variables can hold a reference to the object and the object cannot be accessed. When the first block exits, the `temp` owner goes out of scope and thus, after the borrowing, at the line marked `(**)`, the type of any reference to the borrowed object cannot be formed.

Thus, it is safe to reinstate the value in `b` to `bb`.

6.1.4 Movable Aliased Objects

The original idea that led to the discovery of the abstraction problem with uniqueness and the definition of external uniqueness was the realisation that one pointer to an object was an unnecessary restriction. Uniqueness is useful since it is a way to account for all pointers—if we could move all pointers to an object in one fell swoop from one thread to another or between objects, we could still have transfer of ownership; if we could update the types of all pointers to an object, we can change the object's type. The key behind all examples in this chapter is that all (active) pointers are accounted for since there is only one. We now show how external uniqueness can be used to implement several pointers into a data structure and still have all the benefits of uniqueness. We call these *movable aliased objects* (and sometimes omit "aliased" when it is obvious what we mean).

Movable aliased objects can be encoded into external uniqueness by use of externally unique proxy objects. The proxy can then hold multiple sibling pointers (remember, pointers to objects owned by `owner`) into an aggregate; the aggregate is aliased but movable. To move the object, we simply move the unique proxy object. This moves all external pointers to the movable aliased object since all such pointers are internal to the proxy. A movable aliased object is an object to which there are no external pointers, only sibling pointers. To access it, its proxy object must be borrowed. This creates an external name for the owner of the movable object and allows it to be referenced from the outside.

Figure 6.6 shows the object graph for a movable aliased object and its unique proxy. It also shows a virtual ownership bound for the unique object and its siblings. We say that bound is virtual, since it does not correspond to a

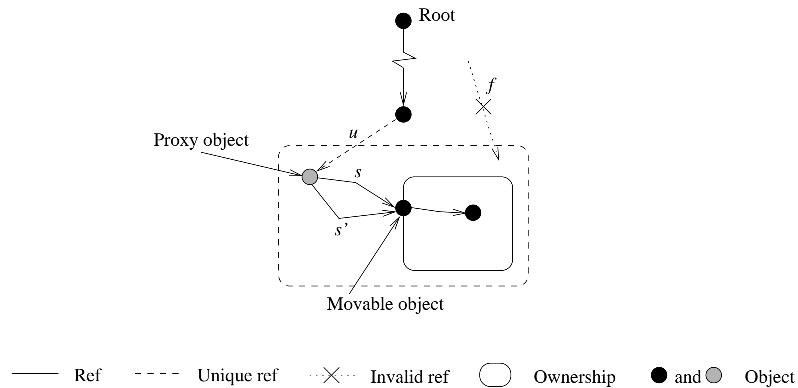


Fig. 6.6. Movable aliased objects—the dashed box denotes the "virtual boundary" of the unique proxy. s, s' are sibling references to the movable object. Owners-as-dominators applies as usual. There could of course be movable objects (or rather, all sibling objects are part of the movable aliased aggregate).

specific object. By the unique-owners-as-dominating-edges property, there can be only one pointer crossing the virtual boundary. Thus, the movable objects are protected from external aliasing and are thus implicitly moved by just moving the proxy.

Applications for Movable Aliased Objects

In this section, we show some applications for movable objects. These applications cannot be handled by traditional uniqueness since they require multiple external pointers.

List with Head and Tail Links

In the previous example in Figure 6.3 there was a single unique reference into the internal set of links of a doubly-linked list from the `List` object. External uniqueness enables doubly-linked lists, which is not possible in traditional uniqueness, by encapsulating all links in the same ownership bound and preventing the existence of more than one external pointer to that bound. The object graph for that example is shown in Figure 6.7. It is quite simple to modify this example to account for multiple pointers into that data structure to enable for example a tail pointer in addition to the pointer to the list head. This too cannot be handled by traditional uniqueness.

Figure 6.9 shows the implementation of a simple proxy object class called `HeadAndTail`. It is completely empty except for two fields, `head` and `tail`. Instead of having the two pointers directly in the `List` class, we encapsulate them in the proxy. The resulting object graph is shown in Figure 6.8. The

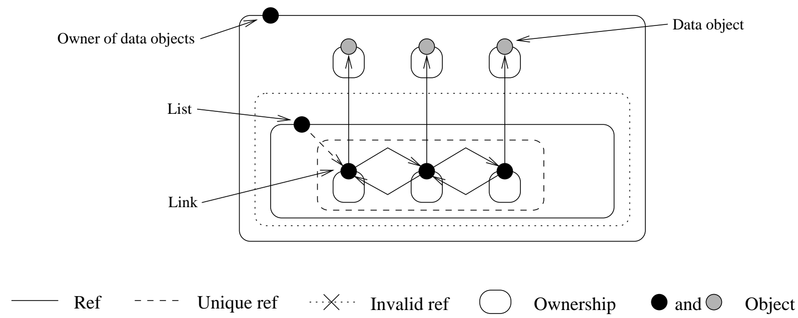


Fig. 6.7. The object graph for the doubly-linked list in Figure 6.3. The dotted box denotes the possibility of any number of owners between the owner of the data objects and the list object. The dashed box denotes the ownership bound of the externally unique set of links.

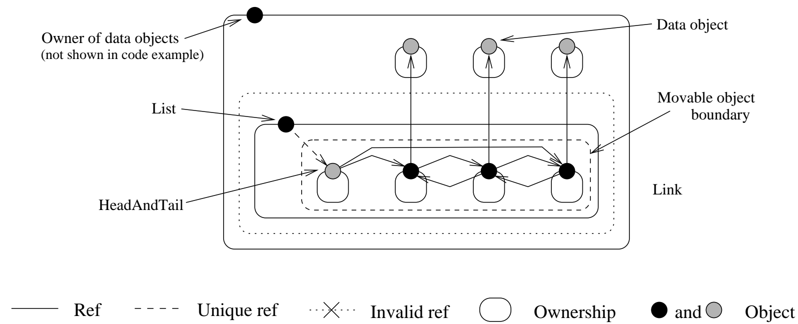


Fig. 6.8. Object graph for the doubly-linked list with head and tail pointers in Figure 6.8. The dashed box denotes the virtual bounds of the owners of the unique objects. The virtual bound corresponds to the owner of `handle`.

dashed box denotes the virtual ownership bound of the unique proxy object that encapsulates the proxy and the links. To show the absence of any magic, we show the modified `merge()` method from Figure 6.3 that moves and appends in the presence of multiple external pointers into the lists. The difference is notably quite small—we now operate on handles instead of directly on the head and the handle of the other list is moved into the representation of the target list and then consumed. Also, because of the presence of a tail pointer, there is no need to iterate through the list to get to the last list element. If our language had tuple types, as Haskell (Peyton Jones, Hughes, et al. 1999) does, movable aliased objects would be even easier to implement, since no special class would have to be written for the proxy object. However, it is now possible to define specific methods for the proxy class to manipulate the external references into the object.

```
class Link<data>
{
  data:Object data;
  owner:Link<data> next, prev;
}

class HeadAndTail<data>
{
  owner:Link<data> head, tail; // The external pointers
}                              // to the movable object

class List<data>
{
  unique:HeadAndTail<data> handle;

  void append(owner:List<data> other)
  {
    borrow handle as <ho> bh
    {
      ho:HeadAndTail<data> ohandle = other.handle--;

      if (bh.head == null)
      {
        bh = ohandle;
      }
      else if (ohandle.head != null)
      {
        bh.tail.next = ohandle.head;
        ohandle.head.prev = bh.tail;
        bh.tail = ohandle.tail;
      }
    }
  }
}
```

Fig. 6.9. Movable aliased objects enable head and tail pointers to unique set of links in a list.

```
class Lexer
{
  this:InputStream stream;     // internal

  Lexer(unique:InputStream s)
  {
    stream = s--;
  }
}

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

Fig. 6.10. Overcoming the initialisation problem

6.1.5 The Initialisation Problem

The initialisation problem is the inability to externally create and initialise objects that are part of some other object's representation. In a system with ownership types, an object's representation cannot be named outside it, meaning it is impossible to create or initialise a representation object outside its owner. This has been a limitation with many of the previous systems with deep ownership.

This is arguably a severe drawback since external initialisation increases the flexibility and facilitates code reuse. For example, plug-in architectures are not possible.

Overcoming the Initialisation Problem in Joline

In Joline, we can overcome the initialisation problem by using movement and transfer of ownership. The externally initialised objects are created as external objects and then moved into the target representation. The dominating edge property of external uniqueness guarantees that once the representation object is moved into their target, no aliases from objects external to the target exist.

Revisiting an early discussion in Section 2.2, the example in Figure 6.10 presents a lexer class that reads tokens from an externally initialised stream. The `lexerClient()` method creates and initialises the `InputStream` which is then moved into the representation of the lexer without leaving any external aliasing to the stream object. This enables the implementor of the lexer to disregard any external aliasing, which makes the implementation easier, voids the need for checks that no-one has *e.g.,* moved the file pointer externally

of the lexer class, and makes it easier to maintain and reason about class invariants.

In our example, the input stream also loses its uniqueness.

7

Discussion

We argue that our model of uniqueness is more appropriate for use in an object-oriented context than traditional uniqueness; external uniqueness views objects as aggregates; unique pointers to true black boxes is allowed; and it does not suffer from the abstraction problem.

In this chapter, we aim to substantiate these claims. In particular, we show how external uniqueness overcomes the abstraction problem and facilitates reasoning. We also discuss the problem with external uniqueness in the presence of multiple threads.

7.1 External Uniqueness for Object-Orientation

By using a strong notion of encapsulation, such as the one provided by ownership types, external uniqueness enables unique pointers to entire aggregates—one single entry point to a potentially large collection of objects. This is a more natural way of looking at a unique reference, since the aggregate is viewed as a whole, and not as a bunch of interconnected objects where the uniqueness only applies to one. A traditionally unique pointer cuts a slice in an aggregate and allows the slice to be manipulated, *e.g.,* moved to another object, without necessarily effecting its representation. For modern programs with large collections of objects cooperating together to form aggregate objects we need a uniqueness mechanism that supports *aggregate uniqueness* as well as *object uniqueness*. Tasks such as moving an aggregate without risking residual aliasing of the aggregate's internals or moving a mobile agent from one platform to another requires uniqueness to honour deep ownership.

In contrast to uniqueness, external uniqueness is a property of how the object is used externally, unbeknownst to the object—not a property of the object itself. An object's implementation does not control how it can be referenced nor does it need to consider its external pointers in its implementation.

There is no need for second-class citizen borrowed pointers with unnecessary restrictions. We allow entire aggregates to be moved, between threads or different objects' representations, without the risk of residual aliasing of either the bridge object, or its representation. Furthermore, we allow arbitrary aliasing in an aggregate, even back pointers to an object considered unique externally. All this makes external uniqueness more properly suited to object-oriented programming.

7.2 How External Uniqueness Overcomes the Abstraction Problem

As described earlier in Section 2.3.2, the abstraction problem is caused by the treatment of `this` internally, or more exactly, the possibility of subjective movement. Additionally, in the case of PRFJ, by the treatment of `this`'s owner which is intimately linked with the former. The key to overcoming the abstraction problem, the same key as to enable unique pointers to aggregate objects, is to prevent subjective movement. We do so by presenting different views of the uniquely referenced object (the bridge object) internally and externally. Internally, the object is non-unique and in the presence of deep ownership cannot escape to become visible outside. Thus, subjective treatment of `this` can disregard how the object is viewed externally; we can even allow the creation of dynamic and static aliases to `this` since these aliases are confined to the object.

The external view of the object is where uniqueness is important. As we have argued before, external uniqueness is effectively unique (Section 4.1.3). Since the internal references are not active or reachable while the object is viewed as unique externally, it suffices to preserve uniqueness externally and let the implementation of the class bother less with the external view of its instances. Ownership types allow us to make the distinction between an objects inside and outside. Since we know that references to representation will never escape, we can allow them, even if the object is viewed externally as unique. From a software engineering perspective, our proposal is better suited to software evolution than traditional uniqueness, since it does not break the principle of abstraction; is does not require interfaces to change when the internal implementation does, as illustrated by the upcoming example.

Subjective Treatment in the Presence of External Uniqueness

The key to overcoming the abstraction problem is to make all possible internal operations valid regardless of how the object is being referenced externally.

Subjective movement is problematic since it requires `this` to be unique; this requirement will necessarily propagate into the interface (if we want modular checking) since the receiver of a method using subjective movement must be unique. We choose to disallow it.

In external uniqueness, objects are never unique internally. If they do not create internal aliases to `this`, they are actually unique (in the traditional sense); if they do, they are effectively unique, or externally unique. The encapsulation of ownership types prevent any internal alias to an externally unique pointer from escaping and compromising the external uniqueness invariant; we can allow an object to treat itself non-uniquely, create aliases to itself etc., and as we have shown, external uniqueness is still effectively unique.

> *As the internal view of all objects is non-unique, the same set of operations are available (and valid) inside the object regardless of how it is referenced and thus, external uniqueness does not suffer from the abstraction problem.*

Since instances will never view themselves as unique, there is no need to reflect treatment of `this` in the interface (or track it using program analysis, other than for the purposes of preserving the owners-as-dominators property of ownership types). Also, since instantiating an owner with `unique` does not propagate through implementation as in PRFJ, there is no need for `where`-clauses or similar constructs to control which objects or owner parameters can be uniquely referenced respective instantiated with `unique`. This means that any change to the class' implementation cannot change its instances ability to be referenced uniquely, nor affect any external, unique references to itself. Any possible treatment of `this` or of the owner parameter is always valid, regardless of any possible external unique references.

Thus, the price to avoid the abstraction problem is the loss of subjective uniqueness—an object can no longer move itself. The gains are much greater.

A Concrete Example

Figure 7.1 shows the implementation in Joline of a `Server` class used in Chapter 2 in the description of the abstraction problem. The difference between this figure and the previous ones is the complete absence of any annotations concerning uniqueness or the treatment of `this`, which s consistent with external uniqueness. The `connect()` method will store a reference to the server in a client object, a consuming method in the case of method-level annotations, and an invalid method in a unique class in the case of class-level annotations. The only additional requirement which stems from ownership types is that the owner of the server must be accessible to the client object. At the

```
class Server extends Object
{
  int no_connections = 0;

  void connect(owner:Client client) // (*)
  {
    client.isManagedBy(this);       // (**)
  }

  int getConnections()
  {
    return this.no_connections;
  }
}
```

Fig. 7.1. The `Server` class example from Figures 2.4 and 2.5 encoded with external uniqueness.

line marked with `(*)`, the client parameter is declared as sharing the same owner as the server object, and it will thus have the necessary permissions to receive the `this` reference at line `(**)`. Possibly a little contrived in this particular example, this means that the client belongs to the same aggregate as the server and that there may be no external references to the client from outside that aggregate. If such references were possible, then the client objects could be used to access the server externally. This would be unsound if the server is externally unique and therefore it is not allowed.

Now, lets consider the effects of changing the code of the figure, in particular replacing the entire `getConnections()` method for the following code, making the method a consuming method.

```
int getConnection()
{
  this:BlackBox<owner> bb = new this:BlackBox<owner>();
  bb.xyzzy(this);
  return this.no_connections;
}
```

The method now creates a temporary black box object with permission to reference the receiver and then passes `this` to the black box's `xyzzy` method with unknown consequences.

In the case of external uniqueness, this change is perfectly legal without changing the method header, since the `getConnections()` method can only be invoked if there are no unique references to the receiver object, meaning that the receiver object can be treated any way we like. Thus, there is no need for the notion of borrowing or consuming methods, no need for annotations,

and the view and possible operations on `this` are the same regardless of the method's implementation.

In case of traditional uniqueness in the style of Minsky (1996), uniquely referencing instances of `Server` would have required some class annotation in the class header. The addition of the back pointer in the change to `getConnections()` would have forced this annotation to be removed—the internal use of `this` leaking out in interfaces.

In the style of *e.g.,* Hogg (1991) and Boyland (2001a), the same problem appears but with different symptoms: the `getConnections()` method is forced to be declared as *consuming* its receiver, and the first call to accept will steal the only reference to the server.

Again: the same set of actions is possible regardless of the object's external reference(s) meaning that the implementation cannot effect how the object is referenced and used externally (nor need it be reflected in the interface)—which is what the abstraction principle states.

7.3 Facilitating Reasoning about Objects

Uniqueness can be instrumental in determining whether a piece of code is sensible or not, for example in the issue of closing and reading two possibly aliased file variables. Uniqueness enables the movement of an object between threads without the risk of residual aliasing, which makes synchronisation unnecessary and therefore reduces the risks of data races and deadlocks in certain situations. Knowing that an object is unique means that we can track its state as is done in *e.g.,* Vault (DeLine and Fähndrich 2001) and Fugue (DeLine and Fähndrich 2003). This would not be possible in the presence of aliases, since it is generally impossible to determine that some method invocation does not affect the state of a non-unique object.

Similar properties are provided by deep ownership. In Joe_1, Clarke and Drossopoulou (2002) show how to determine the disjointness of the effects of two method invocations based on deep ownership types. Smith and Drossopoulou (2003) further this work and use ownership types to facilitate program verification in the context of a Hoare logic with a type and effects ownership system. Also, preserving non-trivial class invariants is a lot easier in presence of strong encapsulation, since modifications of an object's representation are either performed internal to the object, or by the object itself, not by external objects.

By bringing together uniqueness and the strong encapsulation of ownership types, we aim to achieve the benefits of both systems. Indeed, we extend

deep ownership with uniqueness suggesting that all the benefits of deep ownership apply to our system as well.

External Uniqueness and Software Protocol Checking

We observe a weakness in our system that stems from the use of internal pointers. Even though external uniqueness is effectively unique, in a system such as Fugue that tracks type-state (see Strom and Yemeni (1986) for an enlightening read on type-states), coordinating all pointers to an object to make sure that any possible internal back pointer views its referenced object as having the same state as an external pointer is at least a hard problem to solve. A field of an internal object that has a type which allows it to hold a back pointer need not necessarily contain one. Making this work will at least require some additional machinery.

However, external uniqueness gives more powerful guarantees than traditional uniqueness. In particular, the owners-as-dominating-edges property applies to all objects in a transitive closure, not just to a single one. Moving a unique object between threads will move the object along with its representation, even in the presence of internal sharing between the representation objects.

7.4 External Uniqueness in the Presence of Multiple Threads

A feature of uniqueness important in the context of concurrency is that an object referred to uniquely can only be entered by one thread.

However, if threads are created within an object while it is being borrowed, then their mere existence threatens the possibility of retaining a strong notion of uniqueness. If the thread internal to the borrowing outlives the borrowing, active internal references exist simultaneous with the external reference which breaks the effective uniqueness of external uniqueness. We outline three ways to deal with this below.

Make sure borrowing outlives subthreads. An immediate solution is to force the borrowing thread to outlive all subthreads. Any thread internal to the object created during the borrowing must have finished its execution before the borrowing ceases. If any threads are still running at the end of the borrowing, the borrowing thread must wait until they have finished.
This is an unsatisfactory solution since it is not possible to tell when a borrowing will end, if ever.

Prevent simultaneous borrowings. A second solution to the problem is to allow any borrowed threads to continue executing after the borrowing exits. The

new requirement is that these threads must not be alive when the external reference is borrowed again. Thus, a possible delay is deferred to the start of the next borrowing block.

This approach suffers from the same problem as the first solution. Both solutions violate the definition of borrowing since the behaviour of any internal threads might prevent the borrowed value from being reinstated.

Weaken uniqueness. The third solution views internal threads as part of an aggregate's implementation and should therefore be permitted to exist simultaneously with externally unique pointers. Where sound, movement is allowed even in the presence of internal threads and the threads move with the object. Simultaneous borrowings are allowed and localised reentrancy is void.

We feel that creating threads inside borrowed objects should be avoided. Creating a thread linked to a unique object is far superior, since this moves the unique object into the thread together with all its representation and thus voids the need for synchronization.

8

Related Work

In this chapter, we discuss related work; mainly other forms of alias management techniques and variations on uniqueness. As most of the detailed discussions of the latter are included in the relevant sections, this chapter presents only a brief overview of them.

We look at other systems that provide uniqueness and/or aggregates or strong encapsulation and compare these to external uniqueness and point to where these are discussed elsewhere in the thesis. We then look briefly at region-based memory management which is related in some aspects, and uniqueness and linearity in functional programming. We finally look closer on the originality of our proposal.

8.1 Alias Encapsulation: Containment, Ownership, etc.

Alias encapsulation schemes (a large body of which are ownership types systems) have been employed for reasoning about programs, *e.g.,* Clarke and Drossopoulou (2002) and Müller and Poetzsch-Heffter (1999); for alias management, *e.g.,* Clarke et al. (1998), Noble et al. (1998); and in program understanding in the presence of aliasing (Aldrich, Kostadinov, and Chambers 2002). Boyapati and Rinard (2001) and Boyapati et al. (2002) use ownership types as the basis for a system to eliminate data-races respective deadlocks from concurrent programs. Boyapati, Liskov and Shrira (2002) 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 (Hogg 1991), Confined Types (Bokowski and Vitek 1999), Universes (Müller and Poetzsch-Heffter 1999) and Ownership Types (Clarke, Potter, and Noble 1998). Most other approaches are either reminiscent of these, or just weakened versions.

Islands was defined for Smalltalk (Goldberg and Robson 1983) 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. We defer the discussion of Islands to Section 8.4 where we discuss the originality of our proposal.

Confined types uses Java packages (Gosling, Joy, and Steele 1996) 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. Recent studies by Grothoff, Palsberg and Vitek (2001) 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 practice, except that confined types may be quite compatible with existing ways of constructing object-oriented programs.

Universes (Müller and Poetzsch-Heffter 1999) is basically an extended subset of ownership types (see below) 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 (Hogg 1991; Müller and Poetzsch-Heffter 1999), or as stand-alone alias control schemes (Kniesel and Theisen 2001; Skoglund and Wrigstad 2002). In the last case it is however unclear what practical gains stand from using them.

Ownership Types was proposed by Clarke, Noble and Potter (1998). 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.

Ownership Types was described in more detail in Chapter 3. Even more detailed descriptions are available in the literature, *e.g.,* in Clarke's dissertation (Clarke 2001) where a full-blown description of the theory of ownership types is presented in Abadi's and Cardelli's Object Calculus (Abadi and Cardelli 1996), or Clarke and Drossopoulou's (2002) Joe_1 for a more recent implementation of ownership types in a class-based Java-like setting. The last paper is also a good example of how ownership information can be used to assist reasoning about programs by forming the basis of a disjointness theorem showing that two statements do not interfere with each other. See also Smith and Drossopoulou (2003) for further developments of the Joe_1 platform.

8.1.1 Comparison

Table 8.1 on page 112 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 of deep ownership and orthogonal borrowing. The italicised letters in the table are keys that are explained on page 114.

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

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* (Baker 1995) and *unsharable* (Minsky 1996).

	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 8.1. Comparison of related work. (a-k see text.)

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.

As was shown in Figure 3.3 on page 33, 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 (Noble, Vitek, and Potter 1998).

Full alias encapsulation, a term coined by Noble, Vitek and Potter (1998) 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 encap-

sulation 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 (2003).

In addition to the ones considered above, other forms of encapsulation exist, such as the package level confinement found in Confined Types (Grothoff, Palsberg, and Vitek 2001). These are however too coarse-grained to enable external uniqueness and are therefore not further discussed.

Kim, Bertino and Garza (1989) 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.

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 (Chan, Boyland, and Scherlis 1998), temporary (Kniesel 1996), lent (Bacon, Strom, and Tarafdar 2000), unconsumable (Minsky 1996) and unique (Hogg 1991). 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* (Minsky 1996), AliasJava (Aldrich, Kostadinov, and Chambers 2002), Balloon Types (Almeida 1998), Pivot Uniqueness (Leino, Poetzsch-Heffter, and Zhou 2002) and Capabilities for sharing (Boyland, Noble, and Retert 2001).

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 (Boyapati and Rinard 2001), Vault (DeLine and Fähndrich 2001), and in Alias Burying (Boyland 2001a).

Checking the constraints underlying alias burying modularly leads to an interdependence between uniqueness and read effects identified by John Boy-

land (2001b). Guava (Bacon, Strom, and Tarafdar 2000) also uses lent parameters to avoid capturing of objects in a system for avoiding data races in Java.

Comments to the table

a) Parameterised Race-Free Java (Boyapati and Rinard 2001) permits object graphs which violates deep ownership, but it uses an effects system to prevent access through the offending references. The result is *effectively deep ownership*. In addition, to increase flexibility, PRFJ allows `unique` to be used even as a non-owner parameter. For a more detailed discussion of PRFJ, see page 119.

b) Flexible Alias Protection (Noble, Vitek, and Potter 1998) 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. For a more detailed discussion, see page 120.

c) The Vault system (DeLine and Fähndrich 2001; Fähndrich and DeLine 2002) 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 (Aldrich, Kostadinov, and Chambers 2002) 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. AliasJava was shown in the introduction to shallow ownership, Section 3.2.1. For a more detailed description, see Aldrich's dissertation (2003).

e) Pivot Uniqueness (Leino, Poetzsch-Heffter, and Zhou 2002) 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 (Boyland, Noble, and Retert 2001) 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 (Hogg 1991) allow borrowing through read-only references. For a more detailed discussion, see page 117.

h) Balloon Types (Almeida 1997; Almeida 1998) is discussed in more detail on page 118.

i) OOFX/Alias Burying (Greenhouse and Boyland 1999; Boyland 2001a) 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* (Minsky 1996) is an early system bringing traditional uniqueness into object-oriented programming using method-level annotations to deal with subjective treatment of `this`. Eiffel* is covered in the introduction to uniqueness, Section 2.2.

k) Virginity (Leino and Stata 1999) is basically free values obtained by object creation whose freeness is lost once assigned to a field or variable.

8.2 Region-based Memory Management

Our scoped region construct is similar to the lexically scoped `letregion` construct used in region-based memory management (Talpin and Jouvelot 1992; Tofte and Talpin 1997). There are a number of differences. Firstly, our construct is under programmer control, as in Cyclone (Grossman, Morrisett, Jim, Hicks, Wang, and Cheney 2002), 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 (2001)), whereas the regions calculus uses effects to determine that references into a deleted region are never dereferenced.

Both Cyclone (Grossman, Morrisett, Jim, Hicks, Wang, and Cheney 2002) and Gay and Aiken's RC (2001) 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.,* (Yates 1999; Christiansen and Velschrow 1998)), 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 (Boyapati, Salcianu, Beebee, and Rinard 2003). (Other styles of effects system also exist for Java (Greenhouse and Boyland 1999; Clarke and Drossopolou 2002; Boyapati and Rinard 2001; Boyapati, Lee, and Rinard 2002).) 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 (Bollella, Gosling, Brosgol, Dibble, Furr, and Turnbull 2000) 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 (Walker and Watkins 2001; Crary, Walker, and Morrisett 1999), using linearity to track the use of regions to avoid the lexical scoping or region allocation and deallocation in the regions calculus.

8.3 Uniqueness and Linearity

Girard's linear logic (Girard 1987) 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 (Wadler 1990; Baker 1995). Wadler's `let!` construct, quasi-linear types (Kobayashi 1999), and Vault's adoption and focus (Fähndrich and DeLine 2002), 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.

8.4 Originality

Our system is not the first to offer uniqueness and strong encapsulation combined. Some of the systems mentioned previously in this thesis, and in this chapter, namely Hogg's Islands (1991), Almeida's Balloon Types (1997), and Boyapati et al.'s Parameterised Race-Free Java (Boyapati and Rinard 2001; Boyapati, Lee, and Rinard 2002) all offer similar features. Noteworthy, Flexible Alias Protection (Noble, Vitek, and Potter 1998), which later evolved into Ownership Types, provides single entry points to aggregates for free values. We now look at these proposals in a little more detail to show how we extend them.

8.4.1 Hogg's Islands

Hogg's seminal Islands paper used uniqueness as a way of achieving protection from aliasing in object-oriented languages. The protection domain is called an *island* and is basically an aggregate object with a single entry point in the same fashion as is enabled by external uniqueness. Encapsulation is guaranteed statically by the use of side-effect free functions and unique pointers. The main constraint in play to achieve islands is that all references passed in and out of a bridge object must be unique. That way, no references to objects internal to the island can escape and break the protection boundary. Instead of escaping, the entire internal object, which must itself be an island, is moved out of the island.

In Islands, there is no way of distinguishing between references to internal objects and references to external objects. All references count as internal and thus, no state reachable from a bridge object can be referenced from outside an island. This was one of the most severe obstacles overcome in ownership types.

Islands use method-level annotations to achieve and maintain uniqueness. As we have previously shown, this breaks the principle of abstraction, and furthermore requires that all references are treated as internal. Since all references passed in and out of a bridge object are required to be unique, sharing of internal objects as well as external objects passed as arguments to methods in the bridge object is precluded. This makes Islands an unsuitable strategy

for many object-oriented programs. For a comparison of the encapsulation offered in Islands and ownership types, see Clarke's dissertation (2001).

External Uniqueness could be used to implement (a sort of) Island, with a simple requirement on bridge objects that all non-representation parameters and return values of an object are unique.

8.4.2 Almeida's Balloon Types

Balloon types enable the same kind of construct as Islands—a *balloon* is a transitive closure of objects with a single entry point. References cannot cross a balloon boundary in any direction. Instead on relying on extensive program annotations (indeed, the syntactic additions is comparable to uniqueness which cannot be said for *e.g.,* Islands or the deep ownership in Joline), balloon types is achieved using an intricate program analysis, described in Almeida's dissertation (Almeida 1998). The overly constraining restriction of Islands is thus also found in balloons, objects internal to a balloon cannot reference objects external to the balloon, but only for static aliases. Dynamic aliasing is not covered by balloons[1] and thus, the encapsulation offered by balloons does not apply to dynamic references.

We believe that the inability of statically referencing external objects internally is too severe to make balloon types a feasible system for enforcing encapsulation in object-oriented programming. Furthermore, because of this inability, if such a reference is required due to changes to the implementation, the containing structure can no longer be a balloon, forcing the removal of the `balloon` keyword. Thus, the implementation of a class determines its ability of being a balloon, a clear violation of the abstraction principle. However, because of the dependency on extensive static analysis rather than program annotations, this causes no further propagating changes to the program. The cost is the need for whole program analysis to preserve the balloon invariants.

In conclusion, ballons are overly restrictive and lacks even a destructive read to move balloons around (relying instead on deep-copying, which might not be a desirable nor working alternative). In our proposal, the pointers to aggregates enabled by external uniqueness can be moved around (as opposed to deep-copied), and the bridge object may be aliased internally. Also, the underlying model of encapsulation is more flexible, since it allows static references to objects outside the aggregate from objects inside the aggregate.

[1] To this end, Almeida proposes *Opaque balloons*, an extension to plain balloons that places the same restrictions on dynamic references as on static references.

8.4.3 Parameterised Race-free Java

Intended to be used for preventing data-races and deadlocks in object-oriented programming, Boyapati's and Rinard's ownership types system is slightly more powerful than the ownership types system we rely on to enable external uniqueness. It provides deep ownership types effectively—its type system permit references which violates deep ownership, but its effect system will prevent accesses through these references, in a sense similar to the read-only references found in Müller's and Poetzsch-Heffter's Universes (1999).

Also, the PRFJ system allows the `unique` keyword to be used as an owner parameter even in the non-owner position. When parameterised with `unique`, all fields etc. using that parameter as an owner will contain unique references. Naturally the class must have been written with that in mind for it to work. PRFJ uses `where`-clauses such as this[2]:

```
class List<data> where owner != unique
{
  ...
}
```

to prevent owners from being instantiated with `unique` in cases where that would be invalid or simply undesirable.

In PRFJ, an object is declared unique by instantiating its owner parameter with `unique`, exactly as in our system. The key difference is that inside the class, any occurrence of the owning (first) parameter will be replaced by `unique` which means that any potential back-pointer will instead be a unique pointer. This is PRFJ's way of preventing possible internal aliasing of the unique, precisely the opposite of our proposal. This design causes problems with abstraction in addition to the ones described earlier: if changes to the implementation requires the owning parameter to be used in a position where it cannot be unique (*e.g.,* prohibited by a `where`-clause in some other class or if an internal back pointer is necessary), a `where`-clause preventing this must be inserted and thus, any usage of unique pointers to instances of the class at any other location in the program must be removed. In PRFJ, not only how the object treats `this` internally, but how it uses its `owner` parameter causes the abstraction problem.

There should be no problem changing the uniqueness implementation of PRFJ to external uniqueness. The cost is the possibility to parameterise a class with uniqueness annotations to achieve unique pointers inside the class.

[2] Readers familiar with PRFJ may note that the syntax has been changed slightly to match the one we use for external uniqueness for convenience.

8.4.4 Flexible Alias Protection

Flexible Alias Protection (Noble, Vitek, and Potter 1998), takes, as the name hints, a more flexible approach to alias protection than Islands and Balloons. References from within the protection domain to objects outside it are allowed, but the objects in the protection domain may not depend on the external objects' mutable state. We mention Flexible Alias Protection here mostly because of its close relation to ownership types. Flexible Alias Protection enables a strong notion of aggregate objects, with the flexibility of ownership types (enabling references to external objects), even though these references may not be used to read or modify any mutable state.

Flexible Alias Protection introduced a *free mode*, basically a uniqueness annotation, but only for results of object creation or deep copying, not for static aliasing. A reference returned by an expression of free mode is not aliased and is thus by the encapsulation provided by the proposal's other constructs a single entry point to an aggregate, same as for an externally unique reference. Exactly how it is determined that a value is not aliased is not described in enough detail to determine whether or not Flexible Alias Protection suffers from the abstraction problem.

Flexible Alias Protection evolved into ownership types, which, in some senses even more flexible, lacked a lot of the features outlined in the original proposal, such as the *arg mode* that prevented internal objects to be dependent on mutable state of external objects, and the *free mode* described above. In a sense, external uniqueness is an additional step in the direction of finally including all the features of Flexible Alias Protection in ownership types.

❖ 9

Conclusions and Future Work

In this chapter, we summarise and critique our results and briefly discuss future work, in particular the Joline compiler.

9.1 Summary

In this dissertation, we have introduced *external uniqueness* and shown a type system and dynamic semantics for Joline, a language that implements our proposal and outlined its soundness proof along with the important dominance properties of *generational ownership* and the *unique-owners-as-dominating-edges property*. We have also shown applications for external uniqueness that were not possible to implement in previous uniqueness or ownership types systems.

We believe that external uniqueness it better suited to object-oriented programming than traditional uniqueness. It avoids the abstraction problem, and allows unique pointers to true, black boxes whose contents are properly encapsulated. Since internal pointers of an externally unique object are never active at the same time as the external, unique reference, external uniqueness is effectively unique.

9.2 Critique

Introducing external uniqueness in a language that already has deep ownership types, such as Joe_1 (Clarke and Drossopolou 2002) or PRFJ (Boyapati and Rinard 2001; Boyapati, Lee, and Rinard 2002), is virtually free. However, if ownership types is not present, external uniqueness requires a significantly large machinery to function correctly. On a positive side, introducing such a machinery will enable deep ownership even for non-unique objects in the entire system.

We now look at some of the weaknesses of our proposal.

9.2.1 Weaknesses Inherited from Ownership Types

The main weaknesses of external uniqueness are those inherited from deep ownership. Ownership types is simple and powerful in smaller examples and smaller programs, however, in combination with uniqueness, it requires the programmer to program with a single entry point in mind. Although we have nothing to back this claim up with, we suspect that programming with deep ownership in larger programs with more complex structure might be less smooth. For example, in programs that involve cross-boundary aliasing (*i.e.,* references crossing conceptual boundaries, such as references from the view layer to the model layer in a model-view architecture (Gamma, Helm, Johnson, and Vlissides 1994)), the programmer might be forced to choose between the desired level of encapsulation and the desired level of flexibility. Naturally, such trade-offs exist to some extent regardless of the support of a compiler or formal system to maintain it. The formal system just make things less flexible.

The observer pattern illustrates a problem that might arise in larger program structures. To allow objects from one conceptual layer, *e.g.,* the model layer, to be able to subscribe to events generated in the view layer, the event handling objects in the second layer need to be able to access objects of the first layer. While it may be possible to use unintuitive designs to achieve the appropriate ownership hierarchy (*e.g.,* letting the view objects be part of the model objects' representation), this will most likely lead to a flattening of the hierarchy if model objects are shared or interact heavily, or worse, if the objects in the view layer subscribe to events from model objects. The resulting object structure will most likely require many objects to be owned by a common owner to enable interaction, most likely `world`, unless measures are taken to propagate the common owners through the program.

While having little to do with our proposal directly, this illustrates a weakness of the system underlying external uniqueness. Shallow ownership overcome these weaknesses (Aldrich, Kostadinov, and Chambers 2002), by the use of proxy objects etc., but is not strong enough to enable external uniqueness. Recent attempts have been made by Clarke and Drossopoulou (2002) to overcome this while maintaining deep ownership, but so far only for dynamic references. More practical experience of programming with ownership types is necessary to determine if our suspicion is correct.

Boyapati and Rinard (2002) use inner classes to allow multiple access paths and cross-boundary aliasing by allowing objects of inner classes to access the representation of an object while having an external owner. The solution is rather elegant, with only a minor addition, namely the possibility of

an inner class to use `this` of the enclosing class as owner. While this has been shown to overcome the iterators problem, it would be unsound in our system, since a back-pointer could be exported during a borrowing and kept outside even after the borrowing ceases if the owner of the object of an inner class would be external to the boundary of the externally unique pointer. A possible restriction could be to only allow `owner` to be the owner of such “inner class objects”. While much less flexible, this restriction would preserve soundness of our system since any object possibly containing a back pointer will be part of the aggregate since the out-most possible owner of an object of an inner class is the owner of the unique. Thus, it would not be possible to reference while an externally unique reference to the aggregate is in place.

Nevertheless, we feel that this is an inadequate solution since it requires changes to a class to enable sharing capabilities that was not programmed into it in the first place.

9.2.2 The Joline System

The Joline system used to present our proposal is a fairly complex system with several concepts not necessary for external uniqueness. One virtue of the complex system is that it enables us to study the interaction of our proposed, orthogonal features, scoped regions, owner polymorphic methods etc.. This made it possible to produce the large number of examples of applications for external uniqueness which we feel is more intuitive than arguing its uses from a purely theoretical perspective.

On the downside, the complex system might have obscured our results and also slowed the process of proving soundness. A leaner system with a minimal core calculus would undoubtedly have been better in this respect.

Development of a core calculus for external uniqueness would be instructive.

9.2.3 The Price of External Uniqueness

Because of the lack of practical experience from programming with ownership types, there is no telling whether the price of external uniqueness (at least when depending on ownership types) is too high. While the studies of Noble and Potanin (2002) suggest quite some level of object encapsulation is present in existing programs, there are to the best of our knowledge no practical studies of programming in the presence of deep ownership. The proposed implementation of external uniqueness that we have presented here is no doubt costly if the encapsulation given by deep ownership types is considered unnecessary. To enable external uniqueness in an equally strong manner,

less costly for the programmer or less restrictive to the object graph (where possible), we might have to resort to program analysis or dynamic checks, techniques which have drawbacks of their own, possibly in combination with shallow ownership.

9.2.4 Lack of Practical Results

We would have liked to include some practical results in this study, in particular, a practical evaluation of the use of deep ownership and external uniqueness in real programming to address the price of external uniqueness issue above. This will instead be addressed in future work.

9.3 Future Work

We identify a lack of practical experience in using deep ownership with object-oriented programming. Also, there is not much experience in programming with unique pointers as first-class concepts, even though it seems that most objects in most programs are uniquely referenced; the studies of Noble and Potanin (2002) suggest that as much as up to 85% of all objects in an object-oriented program are normally uniquely referenced. This suggests that external uniqueness would be not only useful, but work well with existing ways of buiding software. However, as we have stated, the effects of bringing in the deep ownership necessary to enable external uniqueness are not fully known.

As a future direction of this research, we intend to perform a practical evaluation of our proposed concepts as well as ownership types and how well they fit with normal object-oriented programming. To this end we are in the process of implementing a Joline compiler to apply our ideas to memory management, concurrency, mobility, programming patterns, and so forth. With more insight into the effect on real programming of ownership types and external uniqueness, we also believe that we can better understand the benefits of possibly weaker forms of external uniqueness, alternative borrowing variants etc. We hope to have useful results along those lines to present in future papers.

Movable aliased objects can be used to merge lists' representation without copying. It is possible to move an externally unique set of links from one list to another. However, it is not possible to detach a single list node and move it to another list unless the node is externally unique too. While not directly related to uniqueness, it is related to transfer of ownership, which we believe is a necessary feature of any ownership types system to be used to program real-world applications.

References

[1996] Abadi, M. and L. Cardelli (1996). *A Theory of Objects*. Springer-Verlag.

[2003] Aldrich, J. (2003, August). *Using Types to Enforce Architectural Structure*. Ph. D. thesis, University of Washington.

[2002] Aldrich, J., V. Kostadinov, and C. Chambers (2002, November). Alias annotations for program understanding. In *OOPSLA Proceedings*.

[1997] Almeida, P. S. (1997, June). Balloon Types: Controlling sharing of state in data types. In *ECOOP Proceedings*.

[1998] Almeida, P. S. (1998, June). *Control of Object Sharing in Programming Languages*. Ph. D. thesis, Department of Computing, Imperial College of Science, Technology, and Medicine, University of London.

[2004] Austin, C. (2004, February). J2se 1.5 in a nutshell. Article at Sun. `http://java.sun.com/developer/technicalArticles/releases/j2se15/`.

[2000] Bacon, D. F., R. E. Strom, and A. Tarafdar (2000). Guava: a dialect of Java without data races. In *OOPSLA Proceedings*, pp. 382–400.

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

[2002] Banerjee, A. and D. A. Naumann (2002, January). Representation independence, confinement, and access control. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, Portland, Oregon.

[1999] Bokowski, B. and J. Vitek (1999). Confined Types. In *OOPSLA Proceedings*.

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

[2004] Boyapati, C. (2004, February). *SafeJava: A Unified Type System for Safe Programming*. Ph. D. thesis, Electrical Engineering and Computer Science, MIT.

[2002] Boyapati, C., R. Lee, and M. Rinard (2002, November). Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA Proceedings*.

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

[2003] Boyapati, C., B. Liskov, and L. Shrira (2003, January). Ownership types for object encapsulation. In *28th ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, pp. 213 – 223.

[2001] Boyapati, C. and M. Rinard (2001). A parameterized type system for race-free Java programs. In *OOPSLA Proceedings*.

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

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

[2001b] Boyland, J. (2001b, June). The interdependence of effects and uniqueness. In *3rd Workshop on Formal Techniques for Java Programs*.

[2001] Boyland, J., J. Noble, and W. Retert (2001, June). Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *ECOOP Proceedings*.

[1998] Bracha, G., M. Odersky, D. Stoutamire, and P. Wadler (1998). Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA Proceedings*.

[2003] Cardelli, L., P. Gardner, and G. Ghelli (2003). Querying trees with pointers. Unpublished note.

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

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

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

[2002] Clarke, D. and S. Drossopolou (2002, November). Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA Proceedings*.

[1998] Clarke, D., J. Potter, and J. Noble (1998). Ownership types for flexible alias protection. In *OOPSLA Proceedings*.

[2003a] Clarke, D. and T. Wrigstad (2003a, January). External uniqueness. In *10th Workshop on Foundations of Object-Oriented Languages (FOOL)*, New Orleans, LA.

[2003b] Clarke, D. and T. Wrigstad (2003b, July). External uniqueness is unique enough. In L. Cardelli (Ed.), *ECOOP Proceedings*, Volume 2473 of *Lecture Notes In Computer Science*, Darmstadt, Germany, pp. 176–200. Springer-Verlag.

[1999] Crary, K., D. Walker, and G. Morrisett (1999). Typed memory management in a calculus of capabilities. In *1999 Symposium on Principles of Programming Languages*.

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

[2003] DeLine, R. and M. Fähndrich (2003). The fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research.

[1998] Detlefs, D. L., K. R. M. Leino, and G. Nelson (1998, July). Wrestling with rep exposure. Technical Report SRC-RR-98-156, Compaq Systems Research Center.

[2001] Drossopoulou, S., F. Damiani, M. Dezani, and P. Giannini (2001, June). Fickle: Object re-classification. In *ECOOP Proceedings*, Budapest, Hungary, pp. 130–149. Springer Verlag.

[1990] Ellis, M. and B. Stroustrup (1990). *The Annotated C++ Reference Manual*. Addison-Wesley.

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

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

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

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

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

[1996] Gosling, J., B. Joy, and G. Steele (1996). *The Java Language Specification*. Addison-Wesley.

[1999] Greenhouse, A. and J. Boyland (1999). An object-oriented effects system. In *ECOOP'99*.

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

[2001] Grothoff, C., J. Palsberg, and J. Vitek (2001). Encapsulating objects with confined types. In *OOPSLA Proceedings*.

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

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

[1992] Hogg, J., D. Lea, A. Wills, D. de Champeaux, and R. Holt (1992, April). The Geneva convention on the treatment of object aliasing. *OOPS Messenger 3*(2), 11–16.

[1999] Joyner, I. (1999, July). *Object Unencapsulated, Java, Eiffel and C++??* Object and Component Technology Series. Prentice Hall PTR.

[1989] Kim, W., E. Bertino, and J. F. Garza (1989). Composite objects revisited. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, pp. 337–347.

[1996] Kniesel, G. (1996, November). Encapsulation = visibility + accessibility. Technical Report IAI-TR-96-12, Universiät Bonn. Revised March 1998.

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

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

[1992] Landi, W. (1992, December). Undecidability of static analysis. *ACM Letters on Programming Languages and Systems 1*(4), 323–337.

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

[1999] Leavens, G. T. and O. Antropova (1999, February). ACL — eliminating parameter aliasing with dynamic dispatch. Technical

Report 98-08a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.

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

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

[1986] Liskov, B. and J. Guttag (1986). *Abstraction and Specification in Program Development*. The MIT Press.

[1992] Meyer, B. (1992). *Eiffel: The Language*. Prentice Hall.

[1996] Minsky, N. (1996, July). Towards alias-free pointers. In *ECOOP Proceedings*.

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

[2002] Noble, J. and A. Potanin (2002, June). Checking ownership and confinement properties. In *4th Workshop on Formal Techniques for Java Programs*, Malaga, Spain.

[2003] Noble, J., R. B. E. Tempero, A. Potanin, and D. Clarke (2003, July). 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.

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

[1999] Peyton Jones, S., J. Hughes, et al. (1999, February). Haskell 98 — A non-strict, purely functional language. Available from `http://haskell.org`.

[1998] Potter, J., J. Noble, and D. Clarke (1998, November). The ins and outs of objects. In *Australian Software Engineering Conference*, Adelaide, Australia. IEEE Press.

[2003] Skoglund, M. (2003). Investigating object-oriented encapsulation in theory and practice. Lic. Thesis, Department of Computer and Systems Sciences, Stockholm University, Kista, Sweden.

[2002] Skoglund, M. and T. Wrigstad (2002, March). Alias control with read-only references. In *Sixth Conference on Computer Science and Informatics*.

[2003] Smith, M. and S. Drossopoulou (2003, July). Cheaper reasoning with ownership types. In D. Clarke (Ed.), *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming*, UU-CS-2003-030, pp. 15 – 28. Utrecht University.

[1986] Strom, R. E. and S. Yemeni (1986, January). Typestate: A progamming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering SE-12*(1), 157–170.

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

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

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

[2001] Walker, D. and K. Watkins (2001). On regions and linear types. In *International Conference on Functional Programming*, pp. 181–192.

[1999] Yates, B. N. (1999, August). 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.

A

Appendix

A.1 Elaborating Movement Bounds

The elaboration function below fills in the movement bound of a unique type automatically. Basically, it defaults to `owner`, except for in the initial expressions (that is not inside any object) where `world` is used.

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

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

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

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

$$\mathsf{E}_p(x) = x$$
$$\mathsf{E}_p(e.f) = \mathsf{E}_p(e).f$$
$$\mathsf{E}_p(\texttt{this}) = \texttt{this}$$
$$\mathsf{E}_p(lval\texttt{--}) = \mathsf{E}_p(lval)\texttt{--}$$
$$\mathsf{E}_p(\texttt{new}\ t) = \texttt{new}\ \mathsf{E}_p(t)$$
$$\mathsf{E}_p(null) = null$$
$$\mathsf{E}_p(e.m\langle p_{j\in 1..m}\rangle(e_{i\in 1..n})) = \mathsf{E}_p(e).m\langle p_{j\in 1..m}\rangle(\mathsf{E}_p(e_i)_{i\in 1..n})$$
$$\mathsf{E}_p(\texttt{skip;}) = \texttt{skip;}$$
$$\mathsf{E}_p(t\ x\ = e\texttt{;}) = \mathsf{E}_p(t)\ x\ = \mathsf{E}_p(e)\texttt{;}$$
$$\mathsf{E}_p(e\texttt{;}) = \mathsf{E}_p(e)\texttt{;}$$

$$
\begin{aligned}
\mathsf{E}_p(lval = e\texttt{;}) &= \mathsf{E}_p(lval) = \mathsf{E}_p(e)\texttt{;} \\
\mathsf{E}_p(s\ s') &= \mathsf{E}_p(s)\ \mathsf{E}_p(s') \\
\mathsf{E}_p((\alpha)\ \texttt{\{}\ s\ \texttt{\}}) &= (\alpha)\ \texttt{\{}\ \mathsf{E}_\alpha(s)\ \texttt{\}} \\
\mathsf{E}_p(\texttt{\{}\ s\ \texttt{\}}) &= \texttt{\{}\ \mathsf{E}_p(s)\ \texttt{\}} \\
\mathsf{E}_p(\texttt{borrow}\ lval\ \texttt{as}\ \langle\alpha\rangle\ x\ \texttt{\{}\ s\ \texttt{\}}) &= \texttt{borrow}\ \mathsf{E}_p(lval)\ \texttt{as}\ \langle\alpha\rangle\ x\ \texttt{\{}\ \mathsf{E}_\alpha(s)\ \texttt{\}} \\
\mathsf{E}_p(\alpha) &= \alpha \\
\mathsf{E}_p(\texttt{owner}) &= \texttt{owner} \\
\mathsf{E}_p(\texttt{world}) &= \texttt{world} \\
\mathsf{E}_p(\texttt{unique}) &= \texttt{unique}_p \\
\mathsf{E}_p(p_0\ c\langle p_{i\in 1..n}\rangle) &= \mathsf{E}_p(p_0)\ c\langle \mathsf{E}_p(p_i)_{i\in 1..n}\rangle
\end{aligned}
$$