Ownership-Based Alias Management

TOBIAS WRIGSTAD

KTH Information and Communication Technology

Doctoral Thesis in
Computer and Systems Sciences
Stockholm, Sweden 2006

DSV Report series No. 06-009
ISBN 91-7178-325-3
ISSN 1101-8526
ISRN SU-KTH/DSV/R–06/9–SE

Abstract Object-oriented programming relies on sharing and the mutable states of objects for common data structures, patterns and programming idioms. Sharing and mutable state is a powerful but dangerous combination. Uncontrolled aliasing risks causing representation exposure, where an object's state is exposed and modifiable out of the control of its conceptually owning object. This breaks encapsulation, and hence, in extension, abstraction.

Contemporary object-oriented programming languages' support for alias encapsulation is mediocre and easily circumvented. To this end, several proposals have been put forward that strengthen encapsulation to enable construction of more reliable systems and formally reasoning about properties of programs. These systems are vastly superior to the constructs found in for example C++, Java or C#, but have yet to gain acceptance outside the research community.

In this thesis, we present three constructs for alias management on top of a deep ownership types system in the context of the Joline programming language. Our constructs are fully statically checkable and impose little run-time overhead. We show the formal semantics and soundness proof for our constructs as well as their formal and informal aliasing properties. We show applications and extensions and perform a practical evaluation of our system with our implemented Joline compiler. The evaluation suggests that our constructs are compatible with real-world programming, makes use of some of our own proposed patterns, and encourages further practical studies of programming with ownership-based constructs for alias management.

Contents

List of Figures

List of Tables

Acknowledgements

Nuclear Girl, Prove Me Sound

The Jay Lowlifes, "Avanessa"

First and foremost, I wish to thank my supervisors, present and past, Dave, Love, Louise and Terttu, for all their support. I've been blessed by great supervisors, whose only unifying themes have been a great sense of humour and, curiously, leather pants. The biggest thank you goes, of course, to Dave, for encouragement, for forcing me to talk to the person next to me, one or two beers, telegraphing, and for the 3728 emails in the Dave inbox I've saved over the years.

Second, but only in this respect, Emma deserves a big thank you. For understanding my stress and press during the final write-up, and for supporting me through this entire process—from beginning to end.

Third, I wish to express my gratitude to my family, for support and for making me to think of other things at times. And for letting me sleep under the Christmas tree for a couple of nights every year.

Fourth, I wish to express my gratitude to colleagues and students who I have worked with over the years. To Mats for our endless discussions during our first years in the PhD programme. To Beatrice for letting me rant on about type systems for ever and ever, and for only hitting me occasionally. To Henrik for keeping me from doing even more teaching. To Gustaf and Sebastian for coping without a functioning compiler, and finally to Johan for helping me provide one.

Last, I want to thank Jaakko, Jan, Joc, Jocke, Filip, Fredrik, Linus, Markus, Martin, Olle and Thorbiörn (and who else I forgot) for not letting me slip out into the void. I know I have a tendency to stop calling when work is overwhelming. It is good that you don't. I also want to thank my trusty friends LaTeX and Emacs for carrying me this far.

Tobias Wrigstad, Stockholm 2006-02-03

Chapter 1

Introduction

This thesis makes a contribution to the field of alias management. It presents language constructs for dealing with aliasing in object-oriented programming languages and a formalisation, soundness proof and practical evaluation of these constructs in the Joline programming language.

Encapsulation, the practise of hiding an object's implementation from outsiders, is sometimes claimed to be the most distinguishing feature of object-orientation [51]. Proper use of encapsulation is necessary to achieve modular software with stable abstractions that can change independently of each other. Originally, object encapsulation meant the the enclosing of state and behaviour together in an object, hiding irrelevant details about the implementation to external entities. In the evolving field of alias management, the notion of object encapsulation has been extended to involve hiding references to an aggregate's subobjects from its clients. Not being able to access an object's state directly, but being forced to use the defined protocol is a property that distinguishes objects from mere records. This allows the formulation of invariants on an object's state, such as "salary is never negative", which can be enforced by placing appropriate checks in all methods that are involved in updating the salary field.

Aliasing, or object sharing, is powerful. It allows several objects to share a single data source and the modelling of situations with naturally occuring sharing. However, aliasing is also problematic in that it makes programming more complex. Contemporary programming languages outside the research community lack constructs to ex-

press that an object is aliased, or should not be, or restrict the flow of references from one part of the program to another. In languages where references are the only available way of passing and storing objects (for example Java [62], Smalltalk [61], Ruby [124] and Python [126]), it is difficult to determine the effects of a change made to an object. This is because there is no direct way of knowing if parts of the object's *representation*, the sub-objects of which the larger object is built, is aliased and the effect thus visible to other objects in other parts of the system. Uncontrolled sharing can lead to an object becoming dependent on the inner workings of some other object, a clear breach of encapsulation, and in extension, the principle of abstraction.

Aliasing is ubiquitous, or as some researchers [99] phrase it, is "endemic", in object-oriented systems. Aliasing permits several objects to be built from shared subobjects, which in combination with the mutable state of objects is a mixed blessing: it allows powerful propagating changes, but may cause errors if the sharing is unintentional or the updated object is no longer compatible with all its sharers. Thus, aliasing necessitates good mechanisms for enforcing proper encapsulation to achieve modularity and to protect abstractions and invariants of objects.

In this thesis, we present three language constructs for dealing with aliasing in the context of the Joline programming language.

1.1 Living with Aliasing

Aliasing is an important concern in well-structured software [75]. Aliasing and reference semantics have many benefits—they avoid time-consuming and memory-consuming copying, allow in-place updates and are vital in commonly used data structures and design patterns [57]. Modelling real-world concepts, a major selling point of object-orientation, would be considerably more cumbersome without it, as sharing is a common phenomenon in the real world.

In their seminal paper, "The Geneva Convention on the Treatment of Object Aliasing" [74], Hogg, Wills, Lea, de Champeaux and Holt observe that aliasing can be a source of error and frustration, not only for someone trying to formally prove a property of a program, but also in practical programming situations. As an illustration consider this simple example. If x and y can be aliases, the following, seemingly trivial Hoare formula [72] can be very hard to prove:

$$\{x = \mathit{true}\}\, y := \mathit{false};\, \{x = \mathit{true}\}$$

As a more practical example, an aliased object may be subject to *role confusion* [99] if different objects treat a shared object in different ways. If the different treatments are compatible, problems should not arise, but if they are not (like freeing the shared object before the other object is done with it), the object or its sharers may be put in an inconsistent state.

If aliasing can allow a representation object to be visible outside its owning object, it is possible to modify the representation object in ways not permissible by the protocol of its owner (such as setting salary to a negative number). Such an operation might well invalidate the owner's invariants. If the invariants are hidden inside the owner, an external client with a reference to a representation object cannot tell what they are, which makes them hard to respect. As it is generally not possible to tell from an object what its owner is, it might not even be clear where to look for how an object may be modified with respect to its owner.

Not being clear of ownership of objects can lead to other problems, such as dangling pointers: if two objects independently believe they are the owner of a third, the third object, or parts of its state, may be deleted before all of its clients were finished using it. In a worst case scenario, one might have to consider every change to an object as having a global effect. This is especially hard for formal models that suffer from the combinatorial explosion of assertions whether or not an object is aliased. As has been pointed out elsewhere [99], immutable objects are immune to the first two problems, but most objects are, however, mutable.

Sadly, the support for managing aliases is minimal in contemporary programming languages (including Java [62] and C# [71]). A programmer cannot express that an object belongs to a specific set of other objects and should not be exported outside this set, neither can she ban an object from being aliased or move references from one place to another without possible residual aliasing. It is also not possible to express that an object is immutable and therefore immune to most of the dangerous effects of aliasing. Today, such behaviour has to be encoded manually, which is tedious, error-prone and possibly hard to verify. The encapsulation mechanism commonly found in most object-oriented programming languages, name-based encapsulation, does not

suffice as the protection applies only to the field. As soon as the field's value is read and manipulated independently of the field, the value can be made visible in a non-protected field or simply returned [38, 131]. Better models for encapsulation that allows us to express constraints such as the ones just mentioned are needed. Several have been suggested over the years [73, 7, 99, 18, 3], but they have yet to find widespread use outside of the research community.

1.1.1 Alias Encapsulation and Pointer Restrictions

As pointed out in the Geneva convention [74], as well as in Clarke's [38] dissertation, it is hard to find the right model for alias encapsulation. One wants a model that prevents the bad programs but does not restrict the underlying language in ways that renders it useless for writing real-world programs [38]. The right model should also be easy to understand, by both programmers and people trying to reason about their code or prove a certain property of a program; the kind of "protection" from unwanted aliasing provided by a certain proposal must be understood to be useful. Entire papers [97] have been devoted to understand what the properties of different encapsulation schemes are and make these properties in some sense comparable.

Whereas the strategy of alias encapsulation is to allow aliases but bound their scope, *pointer restriction* schemes manage aliasing by limiting the number of pointers to an object. Over the years, several such proposals have been put forward [73, 92, 30, 127, 47], most of which deal with some notion of *unique object*.

A unique object is an object to which only one reference exists. Thus, a unique object is never aliased. Deleting a unique object is safe, they are not subject to things like role confusion, and the effects of changes to uniques are local and easy to reason about [30, 53]. The concept is easy to understand, but it has some hidden problems in maintaining the uniqueness of a reference. On the positive side, the semantics of unique pointers are very simple and some commercial compilers like Visual C++ [115] even include a *unique* interface attribute to specify an alias-free pointer.

On the downside, the alias management offered by uniqueness scales poorly. Only the unique object itself is effectively encapsulated but its subobjects may be arbitrarily shared. The impact of uniqueness is small, unless all objects are unique, which, besides introducing additional complexity to maintain pointer uniqueness, simply is not possible since sharing is often desirable and necessary in object-oriented systems.

One notable technique for alias encapsulation is *ownership types*, proposed by Clarke et al. [42] to realise Noble et al.'s Flexible Alias Protection proposal [99]. Ownership types was developed fully in Clarke's dissertation [38]. The *deep*, strong form of ownership types has a clear model of encapsulation, the *owners-as-dominators* property: Any path from the root object to an object's representation must go through the object itself. One shortcoming of ownership types is the lack of a mechanism for transfer of ownership—the encapsulation is strong, but in certain cases inflexible. For example, an object's representation cannot be initialised outside of the object itself nor can the representation of two lists be merged into one. There is also a lack of practical experience working with ownership types. This is less true of some of the weaker alias encapsulation schemes [2, 68, 43].

To conclude, aliasing is necessary in object-oriented systems, but necessitates mechanisms for alias control and encapsulation. We now move on to describe our contributions, three constructs for aliasing control presented in the context of the Joline programming language.

1.2 Contributions of This Dissertation

In this thesis, we propose three constructs that facilitate better alias control[1]:

- Owner-polymorphic methods can express temporary permissions to reference an object. This allows short-lived dynamic aliases that are guaranteed to be destroyed after a certain point.
- Scoped regions can express statically stack-local objects and temporary objects that are disjoint from representation objects. This extension also allows short-lived dynamic aliases with a constrained life-time.
- Externally unique pointers is an extended variant of the unique pointer described earlier that we believe fits better with object-oriented principles. Externally unique pointers have more lax restrictions without weakened invariants. They also consider aggregates as opposed to the uninteresting single objects. Externally unique pointers are implemented as a very simple extension to ownership types that also makes ownership types more usable.

[1] The idea of external uniqueness and the initial Joline system was published together with Dave Clarke [44, 45].

The constructs are orthogonal in nature, working together using a combination of pointer restrictions and alias prevention to facilitate alias management. Relying on Clarke's ownership types to do the encapsulation legwork, we present a novel form of *generational ownership*, where stack-frames can be granted a temporary permission to reference a set of objects for a defined duration. We also provide constructs to tie the lifetime of objects to that of a stack frame, even though the objects live on the heap and have static references to temporarily accessible objects, all without any dynamic checks and with little syntactic baggage over what is already in ownership types.

We provide a formalisation of deep ownership types reminiscent of Joe$_1$ [39] for a class-based Java-like programming language that maps closer to real-world programming languages than the formalisation based on Abadi and Cardelli's object calculus [1] in Clarke's dissertation [38]. We also believe that the statement of the owners-as-dominators property in our formalisation is easier to understand than its original formulation [111, 38].

Last, we provide an implementation of our constructs in an ownership types setting in the form of the programming language Joline and attempt a practical evaluation of the language in non-trivial programs.

In conclusion, the contributions of this dissertation are as follows:

Language Constructs for Living with Aliasing Owner-polymorphic methods and scoped regions enable preservation of separation between temporary objects and representation objects. Owner-polymorphic methods also enable borrowing and a more lax programming model where dynamic aliasing is allowed to cross ownership boundaries provided they are explicitly allowed by someone with the proper permissions. They also facilitate reuse of a piece of code with objects with different owners.

Scoped regions gives generational ownership, which is a nice and easily graspable model of encapsulation, and short-lived objects, tied to the life-time of a stack frame.

External uniqueness is a form of uniqueness that enables multiple internal pointers to an object without weakening the uniqueness invariant. We reveal a problem with abstraction in existing uniqueness proposals and show how external uniqueness overcomes it. External uniqueness also was the first construct to enable ownership transfer in ownership types systems.

Combined, these constructs allow a more powerful form of borrowing than in previous proposals. As a side-effect, our constructs remove some previous constraints in deep ownership types, making deep ownership more suitable for writing real-world programs.

Our proposal contains a complete formalisation of the static and dynamic semantics of our constructs, a soundness proof and proofs of the two important structural properties. We also present an alternative formulation of the owners-as-dominators property for deep ownership types that we believe is easier to understand that previous formulations.

Joline Programming Language An artefact of this thesis is a compiler for the Joline programming language implemented together with Johan Östlund. This language embodies all our proposed constructs in a full-blown programming language. The Joline compiler allows for experimentation with ownership types and the proposed constructs in practise and is the first compiler encompassing both deep ownership and external uniqueness.

Empirical Validation A presentation of our practical experiences from programming with ownership types in four case studies, two of which were carried out together with masters students at Stockholm University. These are the first real case-studies of programming with ownership types and external uniqueness. Even though the programs studied are too small to draw any hard conclusions, they suggest that our constructs are compatible with real-world programming and encourages us to develop the Joline compiler further to facilitate the studies of larger programs and longitudinal studies.

1.3 OUTLINE

The upcoming chapter gives a detailed background on aliasing, encapsulation and representation exposure. It surveys related proposals for alias encapsulation and control, to prepare the reader for the remainder of the thesis.

Chapter 3 introduces the Joline programming language, our vehicle for the presentation of our constructs for dealing with aliasing. This chapter presents the language, its syntax and formal semantics and gives a proof of soundness and the owners-as-dominators property. To avoid presenting a huge proof at the end of the thesis we

prove as much as possible for the "vanilla flavoured" Joline system in this chapter and subsequently extend the formalisms as new constructs are introduced.

Chapter 4 introduces owner-polymorphic methods, shows how these can be used to enable preservation of separation between arguments and representation, describes a form of borrowing and the Hide Owner pattern. We give a formal description and soundness proof in the context of the Joline language.

Chapter 5 introduces the scoped regions construct, and shows how it can be used to enable stack-based confinement in a way that preserves separation of temporary objects from arguments and representation. We give a formal description and proof in the context of the Joline language.

Chapter 6 recaps uniqueness and shows an inherent problem with abstraction when introducing the unique pointer concept into an object-oriented setting. We show how external uniqueness not only avoids this problem, but how it enables more lax restrictions while maintaining effective uniqueness. Last, we give its formal description and proof in the context of the Joline language.

Chapter 7 discusses the impact of our proposed constructs, in particular the introduction of generational ownership and how our constructs avoid some of the downsides of previous proposals.

Chapter 8 show cases a few practical applications for our constructs, such as transfer of ownership, merging of representation, borrowing and initialisation. Most of these were impossible in an ownership setting prior to our original proposal.

Chapter 9 briefly mentions three possible extensions and future directions for Joline that would be possible to implement with a minimal effort—unique borrowing, a proposal for downcasting without the need for a run-time representation of ownership and a way to enable iterator-like constructs in a system with deep ownership while maintaining deep encapsulation.

Chapter 10 presents our Joline compiler and our experiences from putting it to work in a few case studies of non-trivial programs of varying sizes.

Finally, Chapter 11 concludes with a summary, critique and directions for future work.

Chapter 2

Background and Related Work

2.1 Aliasing

Aliasing is ubiquitous in object-oriented programming. Commonly used design patterns such as the *Observer* and *Flyweight* patterns [57] are built on the concept of object sharing, as are doubly-linked and circular linked lists. Without aliasing, single-linked lists and hash-tables would require that their data objects are *moved* into the them and become inaccessible to the outside world. This would, for example, prevent data structures that share the same data set but are ordered or sorted differently for fast access using different access methods. An example would be a FIFO queue that describes some order of tasks and a hash-table that allows fast access to the tasks based on their process ids. The ubiquitousness of aliasing in combination with the mutable states of objects that are standard in imperative object-oriented languages makes object-oriented systems particularly sensitive to aliasing.

Aliasing occurs when two or more references refer to the same object. Aliases can be *static* or *dynamic*. Static aliases exist on the heap as part of object structures (fields in an object) or as global variables. Dynamic aliases exist only on the stack frame and are generally considered less harmful than static ones because of their volatile nature [74]. Although ubiquitous and indispensable in object-oriented programming, aliases may break abstraction [12], make programs more complex to understand [73] and make pretty much anything that relies on reasoning about state much, harder than in an alias-free system.

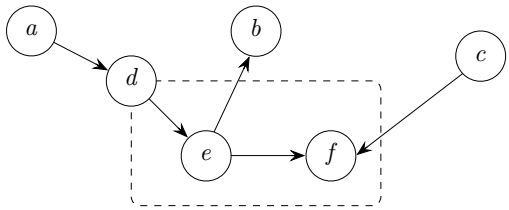

Figure 2.1: Alias types. Object *d* is an aggregate (denoted by the dashed box), aggregating objects *e* and *f*. The reference *a*→*d* is an innocuous, *incoming* alias. The references *d*→*e* and *e*→*f* are *internal* (within the aggregate). The reference *e*→*b* is an *external* alias. The reference *c*→*f* is an *incoming* alias that directly accesses part of *d*'s representation without its knowledge, causing *representation exposure* (see next page).

2.1.1 Different Types of Aliases

In his dissertation, Clarke [38] distinguishes between *internal*, *external*, *outgoing* and *incoming* aliases, a terminology we adopt here as well. Internal aliases exist only within an aggregate, a group of objects working together to form a whole; external aliases are aliases from outside the aggregate to the aggregate itself; outgoing aliases are aliases from within the aggregate to objects outside it; and incoming aliases are the converse. Figure 2.1 shows a graphic presentation of the different types of aliases.

According to Clarke [38], internal references are benign as they do not cross the boundary of the aggregate. The problem is however making sure internally aliased objects are not subject to incoming aliasing or dependent on outgoing aliases.

External references do not affect the implementation of the aggregate. As we shall see later, this happens not to be the case for unique pointers, due to an inherent *abstraction problem* in existing proposals.

Outgoing aliases are interesting, as they denote a dependency on external objects that are not under the control of the aggregate itself. Thus, the aggregate can only rely on the external object's invariants. If these are unknown, the aggregate must conservatively assume that the referenced value can have any form its type allows. To this end Noble, Vitek and Potter [99] introduce the notion of an *arg* pointer, where the aggregate may only depend on the parts of the external object that are immutable. This is powerful, but far from a complete solution.

Finally, incoming pointers are the most dangerous ones [74, 38] as they allow direct manipulation of the aggregate's inners without going trough an interface or using

the aggregate's protocol, possibly violating invariants or witnessing the object in an inconsistent state. This is called *representation exposure*.

2.1.2 Representation Exposure

Representation exposure or *rep exposure* for short, occurs when an alias to a mutable part of the representation of an object is exported outside the representation [48, 40, 38]. In other words, it occurs when an internal alias is leaked to an external object or when an internal reference is created from an incoming alias that continues to exist outside the object. Corresponding code examples can be found in Figure 2.2, see *getName( )* and *setName( )* respectively. We now illustrate the problems caused by rep exposure with an example based on the code in Figure 2.2.

It is not uncommon for an aggregate to use a representation object in a way such that the object assumes only a subset of its possible values or states during a program. For example, a *Person* object might not allow its name, represented as a *mutable* string, to be empty even though the empty string is a perfectly valid *String*. The constructor and the method in the *Person* class that updates the name field perform a simple check to prevent strings that are not valid names from being used (could perhaps be expressed as a precondition). Should the name string be aliased outside the *Person* object, it may be used however a string may be used (see the lines below the class declaration in Figure 2.2). It might, for example, be updated to become the empty string, putting the *Person* object in an invalid state, violating the abstraction of *Person* [12].

The second invariant of the *Person* class says that the value of *len* should be equal to *name.length( )*. This invariant is protected by the implementation and internal use of *setName( )*. As *name* is updateable without going through *setName( )*, this invariant can be broken, though the name stays valid.

In most contemporary programming languages, it is impossible to declare that an object belongs to the representation of another and should not be exposed outside of it. Languages like C++ [122] and Eiffel [90] are notable exceptions. They allow programming with value types (called *expanded types* in Eiffel), that is, objects that have value semantics and are passed around by copy as opposed to by reference. A field of such type will not hold a reference but an entire object. Thus, the object is physically nested inside the memory of its enclosing object achieving some notion of representation.

```
class Person extends Object
{
    // Invariant 1: name must never be an empty string
    private String name = null;

    // Invariant 2: len == name.length( ) at all times
    private int len = -1;

    public Person( String name )
    {
        // If resulting instance is invalid, throw an exception
        if (setName( name ) == false) { throw new CreationException( ); }
    }

    public boolean setName( String name )
    {
        // Protects the both invariants
        if (name.isEmpty( )) { return false; }

        this.name = name;
        this.len = name.length( );
        return true;
    }

    public String getName( )
    {
        return this.name; // Creates an outgoing alias to name
    }
}

Person p = new Person( "Jane Fonda" );
String s = "Barbarella";

p.setName( s );        // creates an outgoing alias in p.name to s
s.update( "Barbara" );     // not p.len == s.length( )—violates the 2nd invariant

String q = p.getName( );  // creates an outgoing alias in q to p.name
q.update( "" );        // p.name is now empty—violates the 1st invariant
```

Figure 2.2: Representation Exposure. The two invariants of the *Person* class are broken by seemingly valid external uses of aliases. Note that *String* is mutable. We ignore the fact that *name* can be *null* for simplicity.

```
class Aggregate extends Object
{
    private Object internal;
    private Object outgoing;

    public Object getInternal( )
    {
        return internal;
    }
}
```

Figure 2.3: Contemporary programming languages cannot express aliasing properties of fields, that a certain field should only be invokable from representation objects, or that a field contains a reference to a representation object.

However, constructs such as these have other problems due to copying such as keeping copies synchronised and the large overhead when passing objects around. Clearly, value-semantics is also not compatible with many data structures, design patterns and situations where sharing is necessary. Furthermore, in C++, which allows pointers and pointer arithmetic, it is possible to create a pointer to the contents of a field, and subsequently share a value that is supposedly copied when passed around. See also the language Tako [81] for a Java derivate with value semantics for simplifying reasoning about programs.

A naive solution to this problem would have to do two things: prevent representation objects from being returned from methods (all fields being private) and prevent argument objects from being stored in member fields. The last statement might not be obvious, but consider the use of *setName( )* in Figure 2.2 and the subsequent external modification to *s*. These restrictions are overly strong as many aggregates need internal sharing—parts of the rep should be exported to other parts of the rep. Thus, facilities for reading rep must be kept and, again, for most contemporary languages, there is no way of allowing such methods to only be invoked from inside the rep.

An illustration of this lack of expressiveness is shown in lines 3–4 of Figure 2.3. By simply looking at the code, there is no way of telling that the first field should contain an internal alias and the second one an outgoing alias (nor any of the other categories for that matter). Subsequently, it is also not possible to prevent the *getInternal( )* method from being invoked on an external alias without removing the method

altogether. Apart for name-based encapsulation, we cannot make the programming language work for us, but must ensure that no harmful aliasing occurs by manually inspecting the code. This can be a daunting task and it is not fail-safe. The "infamous HotJava bug" [116] is an illustration of the short-comings of name-based aliasing. There, aliasing caused a serious security bug. An accessor method leaked references to a private array of privileged signers thereby allowing external objects to add themselves to the array. A recent study of the use of aliasing in "real programs" by Hackett and Aiken [69] also reveals bugs due to unintentional aliasing in PostgreSQL.

Aliasing is useful but comes at a price: When managed correctly and with care, sharing enables natural modelling, in-place updates and memory efficient passing of objects, and so forth. When misused or insufficiently controlled, aliasing may invalidate invariants, violate abstraction, create dangling pointers and cause role confusion. We will now discuss encapsulation, the implementation-hiding principle at the heart of object-orientation, survey the current encapsulation constructs and research on strengthening encapsulation to enable safe sharing and control the effects of aliasing.

2.2 Encapsulation

Abstraction, *information hiding*, and *encapsulation* are different, but closely interrelated concepts [14]. Abstraction makes programming easier and facilitates changes to a class' implementation or replacing a whole class for another that has a compatible interface. It also allows reasoning about a class or component separately from the rest of the system. Information hiding is the practise of hiding the implementation details of a class or module to its clients [90]. This raises the level of abstraction classes provide and makes the software less complex [35]. Encapsulation, often used synonymously with information hiding [20, 64, 114], is a technique for collecting things together and minimising interdepenencies between separate modules by providing a strict external interface [121]. Blair et al. [16] state that encapsulation "implies the provision of mechanisms to support both modularity and information hiding". Wirfs-Brock et al. [130] state that "the concept of encapsulation [...] refers to building a capsule, in the case [of] a conceptual barrier, around some collection of things". In the field of alias control, encapsulation has additionally come to mean the grouping of an object with its representation objects and preventing these from escaping. In the C++ world encap-

sulation is equated with *data hiding* [75]. Whatever the use of the word, in the end, it is all about facilitating abstraction.

Encapsulation has been heralded as one of the distinguishing features of object-oriented programming [51] and has been a major selling point of object-orientation [74]. Classes encapsulate state descriptions and methods, just as the methods encapsulate their implementation [14].

We now examine the predominant forms of encapsulation in contemporary programming languages and follow with a survey of related proposals for dealing with aliasing from the research community.

2.2.1 Name-based Encapsulation and Selective Export

Name-based encapsulation schemes for object-oriented programming languages have been around for a long time. Virtually every object-oriented programming language has some reminiscence of such a scheme. In Simula, keywords for hiding attributes to external object as well as to subclasses was proposed by Jacob Palme in 1972 [104].

The idea of name-based encapsulation is simple; a field or method can be annotated to reflect its *visibility* to the rest of the program. A *private* field or method is only accessible from within the object itself whereas a *public* field is accessible from everywhere. Different languages have different interpretations of these keywords and offer additional visibility modifiers at various levels of granularity for dealing with things as modules and inheritance. In short, many variants exist. In some languages, such as Smalltalk, all variables are private and must be accessed through methods.

Sadly, as only the name of the field or method is protected and not its contents, circumventing name-based encapsulation is easy. As a matter of fact, we have already snuck such an example under the radar in Figure 2.3—the method *getInternal( )*. The method is public, yet it returns the contents of the private field *internal*. Similarly, we could define a public method that writes to *internal* and keep an alias to the argument (see for example *setName( )* in Figure 2.2). We could even declare a public field with a compatible type that aliases the contents of the private field. These examples illustrate that name-based encapsulation is not strong enough to prevent aliasing except in rather trivial cases.

Selective export is a mechanism found in Eiffel [90] that allows a more fine-grained control over visibility of *features* (Eiffel's terminology for field or method). While also

```
class AGGREGATE
creation
    ...                     -- code omitted

feature {A,B,C}
    m: OBJECT
    n (arg: C) is
        local
            temp: C
        do
            !!temp.make         -- equal to temp = new C( ); in Java
            temp.escape( rep )  -- store reference to rep in C-instance
            Result := temp      -- equal to return temp; in Java
        end

feature {NONE}
    rep: LIST

end
```

Figure 2.4: Selective Export in Eiffel. The features *m* and *n* are accessible to instances of classes *A*, *B* and *C* whereas the feature *rep* is only accessible to the current instance. Note that feature *n* breaks encapsulation of *rep* as it allows an aliasing to it to escape in the object returned.

being a name-based scheme, it is much more powerful than the more common method described above. In selective export, each feature, can be annotated with an *exports clause*. The exports clause describes which classes may use the feature. An example of selective export is found in Figure 2.4 that expresses that feature *m* is only accessible from classes *A*, *B* and *C*, and their subclasses wheres feature *rep* is only accessible to the current instance.

Eiffel's method is powerful, but still much too coarse-grained and suffers from the same problems as with *private* and *protected* variables. For example, instances of a class might be used on both the inside and outside of an aggregate and nothing prevents a class with rights to manipulate an aggregate's internal objects from being exported out of the aggregate. Indeed, this corresponds to the implementation of feature *n*.

In conclusion, the standard mechanisms for providing encapsulation in object-oriented programming languages are solving a problem orthogonal to those of pre-

venting unwanted aliasing and overcoming rep exposure. Hiding the names is not a good enough solution for the latter problems, as the programming languages' knowledge of the protection is lost as soon as the reference is read and manipulated independently of the variable. Hiding the actual reference seems like a more promising approach.

We now continue with a survey of related work on encapsulation.

2.3 Related Work on Encapsulation

In this section, we survey recent work that addresses the confinement of references in an object-oriented setting. We begin with uniqueness in the section below, and then work our way more or less chronologically through various proposals giving short descriptions, relations and pointing out strengths and weaknesses.

2.3.1 Unique Pointers

Unique pointers were originally proposed for functional languages to enable in-place updates [60]. A unique pointer is *alias-free*, meaning that no other pointers to the same object exists in the entire system. Subsequently, an object pointed to by a unique pointer is effectively encapsulated in its enclosing object since it cannot be referenced from elsewhere. The unique object can be *moved* to outside the object and will then no longer belong to the first object but be encapsulated in the receiving object, or even on the stack.

Hogg [73] uses unique pointers to create Islands, an alias management system described below, where variables, method parameters and method returns were annotated with a *unique* or *free* keyword to denote alias-free references. Later systems for alias managements have also used uniqueness but depended less on it to achieve encapsulation [6, 27, 21, 45]. Minsky [92] proposes a similar uniqueness extension to Eiffel [90]. Boyland, Noble and Retert [32] propose a capabilities system that can express uniqueness as a combination of capabilities and exclusive capabilities. Some proposals [99, 88] have a notion of a free value, one that is not yet captured in a variable, a weaker variant of uniqueness. Uniqueness has also been used in various ownership types systems [4, 27] which will be described shortly and by Leino et al. to check side effects in a modular fashion [87].

```
class Aggregate extends Object
{
    unique Integer rep = null;
    unique Aggregate setRep( unique Integer rep )
    {
        this.rep = rep--;
        return this;--;
    }
    unique Integer addFrom( unique Integer temp )
    {
        rep.addToValue( temp.value );
        return temp--;
    }
}
Aggregate a = new Aggregate( );
unique Integer i = new Integer( 4711 );
a = a.setRep( i-- );        // i and a are nullified, reinstating a, i is lost
i = new Integer( 42 );
i = a.addFrom( i-- );       // i and a are nullified, reinstating i, a is lost
```

Figure 2.5: Programming with Unique Pointers. The operator -- denotes destructive reading. It returns the value of the read variable and subsequently updates it with *null*. The **unique** keyword denotes uniqueness.

Programming with unique pointers requires use of special techniques to maintain uniqueness. One such technique is *destructive read* [73, 92], where a variable is nullified when read, a natural and conceptually easy restriction. On the downside, destructive reads make unique values "slippery", in the sense that they are destroyed as soon as they are read (the term is due to John Boyland), and other constructs such as *borrowing* [30] are required to allow passing uniques as arguments without having to manually reinstate them. Borrowed pointers are confined to the stack and cannot flow to the heap meaning that borrowed aliases will be destroyed after a method exits. An example of programming with destructive reads without borrowing in a Java-like language is found in Figure 2.5.

Boyland [30] proposes an alternative technique to destructive reads called *alias burying*. It is essentially a live-variable analysis that can determine that, for example, a stack-based alias will be destroyed when a method exists and thus will not invali-

```
class Aggregate extends Object
{
    unique Integer rep = null;

    void setRep( unique Integer rep ) anonymous
    {
        Aggregate temp = this;
        this.rep = rep;                    // No references to this are kept
    }

    void addFrom( borrowed Integer temp ) anonymous
    {
        rep.addToValue( temp.value );  // No references to this or temp are kept
    }
}

Aggregate a = new Aggregate( );
unique Integer i = new Integer( 4711 );
a.setRep( i );              // i is now null, a is not modified
i = new Integer( 42 );
a.addFrom( i );             // no need to destroy or reinstate i or a
```

Figure 2.6: Programming with borrowing and alias burying. The **borrowed** annotation denotes that the no static alias to the reference exists after the method exits. The **anonymous** annotation means that **this** is borrowed in the method. Note that *setRep( )* creates a reference to **this**, but alias burying can determine that this reference is dead when the method call exits. Also note that destructive reads are implicit.

date uniqueness. Other alternatives to destructive reads are copying [66], or swapping [70, 81]. An example of programming with alias burying and borrowing is found in Figure 2.6.

Outside the object-oriented setting, Girard's linear logic [60] 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, since it imposes heavy restrictions on data types (linear values may not be stored in non-linear values), which requires excessive passing of unique objects [127, 10]. Wadler's let! construct, quasi-linear types [79], and Vault's adoption and focus [53], for example, introduce means for alleviating this pain.

Smith et al. [119] propose alias types, a type system for low-level typed languages like compiler intermediate languages. The alias types type system can express both

that a location is uniquely pointed to and how updates to that location can change its type. In a style similar to borrowing and Fähndrich and DeLine's adoption and focus, they allow linear types to be temporarily treated as non-linear. Memory described by non-linear constraints may not be deallocated or used at different types (and thus, linear types stored in non-linears are protected). This work has also been extended to deal with recursive data structures, albeit not for an object-oriented setting [128].

2.3.2 Systems offering Strong Encapsulation

One of the earliest proposals for strengthening encapsulation for object-oriented programming languages was Islands, proposed for Smalltalk by John Hogg [73].

An Island is an aggregate consisting of the transitive closure of objects reachable from a bridge object. Objects are moved into an island using unique pointers and destructive reads. As every object is an island, they can have no outgoing aliasing that would create a dependency on external objects if moved into some other island. The bridge object is a single entry point to the island and every access to objects in the aggregate must go via it. This is called *full encapsulation*. Thus, Islands enables a strong notion of aggregate.

Just as "no man is an island" [50], neither are objects. Requiring objects to be self-contained entities is a too strong restriction, and the resulting programming model has proven to be too inflexible [2]. The Islands proposal was also never formalised nor is it statically checkable.

Similar work with equally strong protection done by Almeida in Balloons [7] had considerably less syntactic baggage. Islands relied heavily on annotations whereas Balloons relied on a brittle program analysis [38] and copying to pass references between Balloons. Copying is expensive and potentially harder to program with as copies must be discarded correctly or kept in sync.

2.3.3 Systems offering Lightweight Encapsulation

Systems offering more lightweight encapsulation have been proposed. These systems are less constraining and offers weaker encapsulation than full encapsulation systems. Confined Types were proposed by Bokowski and Vitek [18], and formalised by Zhao, Palsberg and Vitek [134]. Confined types prevent instances of package-scoped classes from being accessed or manipulated from outside instances of classes defined outside

the package. For the implementation of a list we could define a *list* package and make the link class package scoped. This prevents aliases to links from outside the package and if only classes belonging to the list implementation, such as *List* and *Link*, are contained in the package, link instances are effectively confined to being referenced from list objects. As the unit of confinement is packages, confined types cannot express that one object belongs to the representation of another (which require per-object confinement), which is the basis of the ownership-based systems that we describe below. For our list example, this means that a link can be shared between lists, which can be seen as both a feature and a flaw.

For a class to be confined to a package it must not be public; all inherited methods must be anonymous (*this* will not be stored); it cannot appear in the type of a non-private field it method of a non-confined type; and all its subclasses must be confined. Additionally, a confined type can only be widened to types with the same level of confinement.

The confinement inference tool Kacheck/J [68] has been used to analyse programs in the Purdue Benchmark Suite, around 100,000 classes. The results show that around 6.5% of all classes are effectively confined. Another interesting result is that 48% of all methods are anonymous, a result relevant even for systems with unique pointers as this suggests half of all method calls on unique receivers will require the receiver to be nullified.

Confined types have also been successfully applied to Enterprise Java Beans to enable static confinement checking [43]. This system, by Clarke, Richmond and Noble, has a slightly different set of rules than described above. Another important difference is the use objects (beans) as the unit of confinement.

Related to confined types, Real-Time Scoped Java [133] also have some interesting light-weight aliasing guarantees with its *scoped classes* and *portal classes*. Instances of scoped classes are only accessible to instances of classes defined in the same package or sub packages. Instances of portal classes are accessible to instances of classes defined in the immediate parent package. This achieves a nesting structure reminiscent of deep ownership (see below).

All in all, recent research shows that practical results stands to gain from using more lightweight systems.

2.3.4 Ownership-Based Systems

A good number of proposals for alias encapsulation are based on the notion of *object ownership*, conceived by Kent and Maung [77]. In ownership-based systems, every object is owned by another object, or some special "root owner". Depending on the system's containment invariant—the restrictions on references imposed on representation objects by their owners—different degrees of encapsulation are achieved.

Müller and Poetzsch-Heffter [95] propose Universes, a system for controlling representation exposure in Java. Universes associates a *universe*, or *context*, with each component. A component's universe is the partition of the heap where its representation is stored. As its representation objects can have representation themselves, universes form a nested structure. Two universes are either enclosing each other or are disjoint.

In a universe system, intra-universe aliasing is allowed. For inter-universe aliasing, references may only flow from objects in a universe to objects in a directly nested universe (the former are said to be the *owners* of the latter). The only exception to this is read-only references which are unconstrained. This protects the rep of a component from write-accesses, except from other objects in the component's rep or from the component's owning objects.

Objects that provide interfaces to the component are not inside the universe, but are the owning objects of the objects in the universe. Incoming aliases of a component's rep (that is, into its universe) may only originate from the universe's owning objects. All other inter-universe aliasing is banned, with the notable exception of read-only references. This prevents representation exposure and deals, albeit in a crude fashion, with the problem argument independence, when an object becomes dependent on an argument object over which it has no control. Thus, universes systems do not restrict the existance of aliases, but to where aliases through which updates may be performed may flow.

The universe invariant is stated thus [95, 94, 49]:

> *If object* o_1 *holds a direct reference to object* o_2*, then, either* o_1 *is the owner of* o_2 *(that is,* o_2 *belongs to the rep of* o_1*),* o_1 *and* o_2 *have the same owner, or the reference is read-only.*

The read-only references allows references to an object's representation to be exported outside the object. This breaks encapsulation, but in a fairly controlled way

as the references cannot be used to update an object. In particular, they can only be used to invoke *pure methods*. A method is pure if it is side-effect free, does not update any fields and only calls other pure methods, a pretty strict limitation. Read-only-ness is transitive, and reading fields via a read-only reference always yields another read-only reference. Pureness is tracked by annotation, and overriding must preserve pureness. The containment invariant offered by the universes system is sometimes called *owners-as-modifier*, as the owner can control modifications of owned objects, but not read-only access [49].

More recently, Leino and Müller have used universes to describe and check object invariants [85] and class invariants [86]. The encapsulation offered by these systems is weaker than the original universes proposals. Notably, in "Modular verification of static class invariants", encapsulation is not per-object, but per-class. This make sense for verification of class invariants as they depend on other class invariants for classes of representation objects and not on invariants of specific objects. See also work by Barnett et al. [13].

A lightweight version of the Universes system [49] has been implemented in the compiler for JML, the Java Modelling Language [83], to enable ownership-based verification techniques to programs specified in JML.

Flexible Alias Protection [99] was the first system to present a flexible alias encapsulation (in relation to Islands) that overcomes representation exposure. It sports a *rep* annotation to state that an object belongs to another's representation an thus may not be exported to outside objects. The rep annotation is used on types of fields, parameters, returns and on type parameters of a parametrically polymorphic class. Additionally, Flexible Alias Protection deals with dependence on outgoing aliases through a special reference type, *arg references*, which only allow access to immutable parts of the referenced object. Thus, an outgoing arg alias is safe to depend on as the only accessible parts are immutable and therefore will not change under foot. Flexible aliasing also includes *value objects*, which are immutable and thus safe to share, and a *free annotation* that captures uniqueness for values uniquely referenced from the stack.

Ownership Types [111, 41, 38] stem from Flexible Alias Protection. The containment invariant of ownership types is powerful and easily stated:

> *If an object* o_1 *references another object* o_2*, then either* o_1 *is the owner of* o_2 *or is internal to the owner of* o_2.

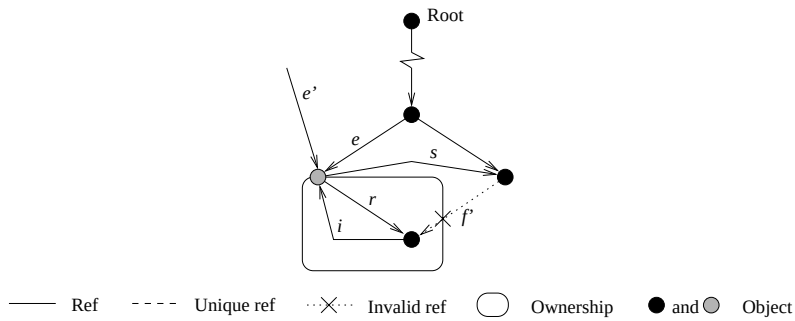

Figure 2.7: Deep ownership. *Reference kinds*: f′ – invalid (breaks deep ownership). s – sibling. e, e′ – external to grey object. i – internal. r – representation.

A nice theorem [111, 38] states that if that condition holds, then an object's owner will be on all paths from the root of the graph to that object, which is to say that an object's owner is its *dominator*. This property is called *owners-as-dominators*, and systems that have it provide *deep encapsulation*. Thus, external aliasing is only permitted from inside the owning object and incoming aliasing is banned. Outgoing aliasing is possible and there is no protection against argument dependence as in Flexible Alias Protection. However, for every reference it is clearly visible in the program whether it points to an external object or not, suggesting that programming with outgoing aliasing is made easier.

Deep ownership is illustrated in the second picture in Figure 2.7. 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 graphical explanation of the owners-as-dominators property is that *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 [38], and Clarke and Drossopoulou [39], this could have some positive side effects on garbage collection, somewhat similar to region-based memory management.

No presently available ownership types system addresses the problem of *argument dependence*. While external arguments cannot be mistaken for representation objects, they can still be changed "under foot" in ways incompatible with the object holding the outgoing aliases.

The deep encapsulation of ownership types has been argued to be too restrictive [21, 3]. For example, it is impossible to create an iterator external to a list, as outgoing aliasing is banned. Also, ownership is fixed for each object's lifetime, preventing external initialisation and the merging of the representation of two objects.

Clarke and Drossopoulou propose Joe$_1$ [39], an ownership types system with effects annotations that can be used to prove that two pointers are not aliases and that two operations will not interfere.

Clarke's dissertation [38] provides an extensive treatment of the theory of ownership types as well as a developed background an comparison of earlier work on alias encapsulation. As our proposed constructs rely on ownership types to work, a more detailed presentation of ownership types will follow in a later chapter.

Boyapati's SafeJava [21] and its predecessors [27, 22] build on Clarke's ownership and work by Flanagan and Abadi [54] on types for safe locking. SafeJava allows a crucial weakening of the containment invariant: incoming aliases are allowed to instances of inner classes that share the rep of their enclosing objects. With this notable exception, Boyapati's system enables deep encapsulation. Even though the systems permit references which violate deep ownership, its effect system will prevent access through these.

SafeJava sports unique pointers that allows ownership transfer, immutable objects reminiscent of Flexible Alias Protection's value objects. Relying on deep ownership, it has been used to prevent data races and deadlocks in multi-threaded settings [22, 27], to enable safe region-based memory management [28] and safe upgrades in persistent object stores [25].

Inspired by Confined Featherweight Java [134] and phantom types concept [55], Potanin et al. are using parametric polymorphism to encode deep ownership in Ownership Generic Java [109].

Aldrich's AliasJava [6] and ArchJava [4, 2] builds a system for enforcing architectural structure, which necessitates alias control, on top of an ownership types system. The encapsulation offered by the system is *shallow* and there is no clear containment invariant. This makes the system more flexible than systems with deep ownership, and studies suggest that it can be incorporated in existing code with small changes [2]. The *downside* and major difference between the ownership in Aldrich's systems and those that offer deep ownership [38, 21] is that the former lacks nesting relations between the *ownership domains* (called contexts in [38] and universes in [95]). Thus, it is not

possible to express that an object with permission to reference object o's rep may not be exported outside o. This makes it possible for owned objects to escape by proxy: if some object p is created internal to o and given the right to reference o's rep, nothing prevents p from being aliased outside o, *causing (indirect) representation exposure* [131].

Later work by Aldrich [3] and Krishnaswami and Aldrich [80], further develop ownership domains. Access rights are specified as inter-domain relations: the right to reference but not create objects in a domain, the right to reference and create objects in a domain, and no right to refer to or create objects in a domain. The system can express access rights policies that overcome the problem with escaping proxy objects pointed out above. Thus, a stronger protection can be achieved, but in contrast to a deep ownership system, protection of representation is not automatically preserved, must be encoded explicitly, and requires additional annotations to the system. Krishnaswami and Aldrich also propose methods that are owner-polymorphic, and allow extended temporary permissions (for example, creating an object in a temporarily accessible context), similar to other work [38, 34, 45].

In conclusion, a wide variety of systems exist that offer different levels of encapsulation. For a graphical comparison between the ownership systems, as well as uniqueness, Islands and Balloons, see Noble et al. [97].

2.4 Related Alias Management Techniques

2.4.1 Effects and Read-Only

A lightweight technique that does not encapsulate aliases but instead aims to control their effects is the concept of *read-only references*, generally transitive versions of C++'s *const* construct [122]. Such references are used in Islands [73] as well as Universes [95] to overcome some of the aliasing restrictions and alleviate the pain of programming under restrictive encapsulation schemes. Quite a few proposals for read-only references have been been put forward [73, 78, 32, 117, 118, 15]. They mostly have only minor differences. Notably, Skoglund and Wrigstad [117, 118] have a more lax definition of a read-only method than most other schemes; it is only guaranteed not to modify its receiver when invoked on a read-only reference. This is flexible, but as Birka and Ernst [15] point out, somewhat ad hoc.

Leino [84] proposes data groups which is basically an effects system that can express what modifications a method may do to groups of variables. The partition of variables into groups is clever and enables smooth treatment of inheritance and method overriding. A similar system is proposed by Boyland and Greenhouse [65]. While these systems neither encapsulate aliases nor control their effects, they somewhat facilitate programming with aliasing and the possible effects of for example a method call is visible from the an object's interface.

Clarke and Drossopoulou's previously mentioned Joe_1 system [39] also makes use of an effects system based on ownership.

2.4.2 Separation Logic and Representation Independence

Reynolds' Separation Logic [113] is a logic for reasoning about shared mutable data structures. The *spatial conjunction* operator, $*$, can express that different formulas hold for disjoint parts of the heap and that parts of the heap are disjoint. For example, $P * Q$ states that a heap can be split into disjoint subheaps, one satisfying P and one satisfying Q and the extended Hoare formula $\{P\}\, c\, \{Q\}$ specifies that after evaluating of a command c, in a heap satisfying P, the resulting heap satisfies Q.

The fact that the formula refers to specific heaps enables the *frame rule* [101]. The frame rule allows extension of the pre- and postconditions with arbitrary disjoint heaps that wont be modified by the command.

$$\frac{\{P\}\, c\, \{Q\}}{\{P * R\}\, c\, \{Q * R\}}$$

For example, if $\{x \mapsto 2\}\, x = x + 1\, \{x \mapsto 3\}$, then $\{H * x \mapsto 2\}\, x = x + 1\, \{H * x \mapsto 3\}$, for any H, where $x \notin \mathit{dom}(H)$.

Using separation logic, it is possible to prove that parts of a program will not affect certain parts of the heap. This facilitates reasoning about pointers and programs that alter data structures as a proof can concentrate on only the parts of the heap that a (sub)program accesses.

O'Hearn et al. [102] use a hypothetical frame rule to reason about static modularity but cannot express abstract data types or classes. Parkinson and Bierman [105], overcome this limitation, but note a downside in that the proofs become less compact. On a side note, Drossopoulou and Smith [120] extend a Hoare logic with frame properties relying on the aliasing restrictions of a deep ownership types system.

The goal of separation logic is verification, even though there are ideas of using as a typing discipline.

A good example of the benefits of abstraction is that one class' implementation can be replaced by another class with a compatible interface. If representation exposure can occur, this is no longer possible as external objects may well depend on implementation details, such as which classes are used in the internal representation. Thus, it is no longer possible to rely on compatible interfaces for replacement, suggesting that abstraction is broken. Banerjee and Naumann's [11] work on *representation independence* tackles this problem. They formulate representation independence for classes in an object-oriented programming languages with pointers, subclassing and dynamic dispatch. Classes which satisfy a certain confinement condition are proven to have "representation independence". Their results are achieved using a static analysis rather than relying on annotating code with qualifiers to control alias encapsulation.

2.5 Concluding Remarks

In this and the previous chapter, we have introduced the alias problem and discussed why aliasing in indispensable to object-oriented programming. We have discussed static and dynamic aliasing, the different kinds of aliases, and described specifically how incoming aliasing can lead to representation exposure and violate abstraction. We have also surveyed recent work on encapsulation and alias management.

In the next chapter, we present Joline, a programming language built on ownership types with deep encapsulation serving as our vehicle for the rest of the thesis. We show the language's syntax, its static and dynamic semantics, and parts of its soundness proof and proof of the owners-as-dominators property. The language is in itself interesting as it incorporates deep ownership in a Java-like language with subtyping and dynamic binding. Subsequent chapters will extend Joline with our proposed alias control constructs and show their effects on the language as well as their aliasing properties.

Chapter 3

The Joline Programming Language

In the previous chapter, we gave a detailed introduction to the alias problem in an object-oriented setting, and to why alias control is hard to achieve, and we surveyed previous work on this topic. This chapter introduces the syntax and semantics of the Joline programming language, our vehicle for the proposed alias control constructs presented in subsequent chapters.

Joline is a class-based object-oriented Java-like programming language with deep encapsulation due to ownership types. It is based on Clarke and Drossopoulou's Joe$_1$ [39], but lacks some of its constructs, such as the effect annotations which are not needed for our purposes here, and add others, most notably ownership nesting in the class headers. Joline supports inheritance, overriding, subsumption and dynamic binding. *While not the main contribution of this thesis, Joline is the only class-based language with deep ownership to have all these properties and to be formally proven sound in one place.*

We refrain from discussing trivial and well-understood constructs like conditionals and loops as they have the standard semantics, even in the presence of ownership types.

This chapter proceeds as follows. First, we describe ownership types and how ownership is implemented in Joline. We then discuss the syntax piece by piece, and conclude with the the language's static and dynamic semantics and soundness proof.

3.1 Ownership Types in Joline

Joline offers deep ownership. Deep ownership types [42] enforces the conceptual structural property that an object's representation is *inside* its *enclosing* object and cannot be exported outside it.

Ownership types introduces the notion of an *owner* and representation objects are *owned* by their enclosing objects. Classes are parameterised by ownership information and types are formed by instantiating these parameters with actual owners similar to Clarke's thesis [38].

Deep ownership enable constraining of the object graph by capturing the nesting of objects in the types in a simple and elegant manner. Representation objects are ordered *inside* their enclosing objects, and references to representation are not allowed to flow to the outside world. As the nesting is captured in the class declarations, the nesting information is propagated through the program, giving control over the global structure of the object graph. By prohibiting references owned by some owner x to flow to objects outside x, a strong, but flexible, containment invariant is achieved that cannot be circumvented as in shallow ownership, causing indirect representation exposure [38].

The existence of nesting relations between owners in a system with deep ownership types 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, ownership nesting allows us to distinguish between the outside and inside of an object.

3.1.1 Implementing Deep Ownership

Joline offers deep encapsulation through deep ownership types. Deep ownership types enable a strong notion of aggregate. Every object has an owner that is fixed for life and every object is an owner and can be the owner of other objects. An object owned by some object o is part of o's representation. Representations are either nested or disjoint, meaning that an object cannot belong to two different objects' representations. Ownership information is captured in types and ownership nesting is captured in class headers.

An owner can be seen as the permission to reference a group of objects. Types are formed from classes and *owner parameters*, which serve as placeholders to give

permissions to reference external objects. Thus, a type does not denote a set of possible instances of a class, but a set of possible instances of a class *with a particular set of permissions to reference other objects*. Types with different owner parameters are not compatible and references of types with different owners cannot be aliases [39].

We now describe ownership nesting, relations between owners and how types are formed from classes and owner parameters.

Classes with Ownership Information

To be able to statically determine ownership nesting, class declarations are extended with assumptions which describe the relations between owner parameters to thread nesting information through the program. As an example, the class *Example* below takes two owner parameters where the first parameter is nested inside the second.

class *Example*< *owner1* **outside** *owner, owner2* **outside** *owner1* > *{ . . . }*

Ownership parameters of a class must always be outside *owner*, the owner of the instance. This is key to avoiding the problem of indirect representation exposure in shallow ownership—an object belonging to some owner x cannot be allowed to reference objects owned by owner y if y is inside x.

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

The owner of a type is written before the class name and needs no explicit declaration in the class header, just like receivers in method declarations in most object-oriented programming languages.

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

class *Super*< *some* **outside** *owner* > **extends** *Object { . . . }*

class *Example*< *owner1* **outside** *owner, owner2* **outside** *owner1* >
extends *Super*< *owner2* > *{ . . . }*

The owner must be preserved through subtyping as it acts as the permission governing access to the object. Preserving it by subsumption is a key to achieving a sound system [38]. In the example above, *Example*'s second owner parameter will be mapped to *some* when viewed as its super class. This is valid if *owner2* is outside *owner*, a requirement derived from *Super*'s class header. That the requirement is fulfilled can be derived from the class header of *Example* as nesting is a transitive relation (*owner2* is outside *owner1* and *owner1* is outside *owner* implies *owner2* is outside *owner*).

Forming Types

Types have following the syntax:

$$owner : ClassName < owner_1, \ldots, owner_n >$$

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

When forming types from a class, the nesting requirements of the class' header must be satisfied by the owners in scope. The object graph is well-constructed with respect to the nesting requirements specified in the classes. A more formal syntax description follows shortly.

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

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

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

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

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

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

Last, *super* shows a concrete example of subsumption. *owner : Super< owner2 >* is a super type of *owner : Example< owner1, owner2 >* and we can therefore assign from *outgoing* to *super*. Note the remapping and hiding of owner parameters as discussed on the previous page.

Remarks

The ownership relation forms a tree, where an object is inside the object which owns it. Owners other than objects are possible. For example, Boyapati et al. [22] have introduced threads as owners to enable thread-local objects.

The inside relation is transitive, meaning that as *this* is inside *owner* and *owner* is inside *owner1* in the previous example, *this* is also inside *owner1*. Ownership is however *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.

This section has introduced the owner concept, shown class declarations with ownership nesting and shown how types are formed. Now, we move on to a more elaborate example to show the flexible encapsulation model of ownership types, and discuss the containment invariant of deep ownership.

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

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

Figure 3.1: A linked list and links using ownership types

3.1.2 Encapsulation Example

The classic example of ownership types is a list class. It has been used many times [42, 95, 38, 44, 45, 131, 21, 110] to introduce ownership and ownership nesting. As opposed to for example Islands [73], ownership types allows the data in the list links to be external to the list and shared with other external objects that have the right permissions. As an example, this allows two lists sorted in different ways to share the same data objects.

The code in Figure 3.1 shows the class declarations of a linked list and its links, omitting methods for brevity. The list class is parameterised by an owner for the data objects that will be contained in the list. This gives the list permission to reference the data objects, a permission which 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. Subsequently, the links cannot escape the list boundary (as shown in the picture of Figure 3.2), 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 (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 the data objects have that same owner.

In ownership types, owners are fixed for life. Thus, is it not possible to merge the set of links for two lists as they will belong to different representations. Ownership

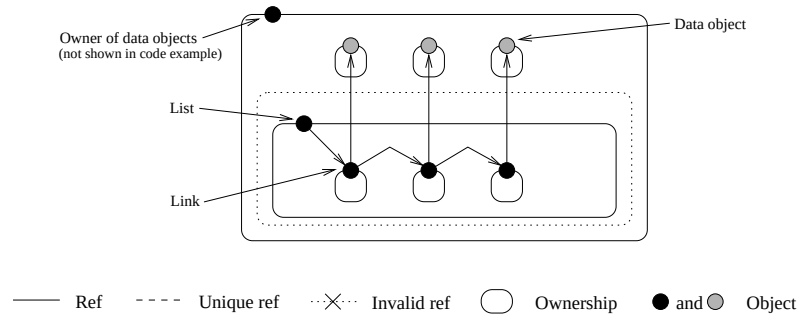

Figure 3.2: Object graph for the linked list example Figure in 3.1. The dotted line indicates the presence of zero or more boundaries between the list object and the data objects in the list. To satisfy the containment invariant, the list object is nested inside the owner of the data objects.

types also preclude the returning of an iterator with access to the links to external objects as the necessary type would not have the owner inside all its owner parameters (its permissions to reference non-rep objects) and would thus be invalid.

We have now shown how deep ownership types is implemented in Joline. Confident that the reader understands the concepts of owners and nesting, we move on to describe the rest of Joline.

3.2 JOLINE, STATICALLY

We now describe the syntax and static semantics of Joline. The dynamic semantics follow in Section 3.3. Note that the formal description of Joline in this chapter is only partially complete and will be extended in subsequent chapters when our proposed constructs for alias control are introduced. To avoid explaining things thoroughly twice, with the second explanation invalidating the first, some aspects of the formalism will be glossed over in this initial description. For the same reason, parts of the proofs are left out as proofs of more general constructs will fill in the gaps later.

Table 3.1: Syntax of Joline

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

P	::=	$class_{i\in 1..n}$ s e	*Program*
class	::=	`class` $c\langle \alpha_i\ R_i\ p_{i\in 1..m}\rangle$ `extends` $c'\langle p'_{i'\in 1..n}\rangle$ `{` $fd_{j\in 1..r}$ $meth_{k\in 1..s}$ `}`	*Class*
fd	::=	$t\ f = e$;	*Field*
meth	::=	$t\ md(t_i\ x_{i\in 1..n})$ `{` s `return` e `}`	*Method*
lval	::=		*l-value*
		x	*variable*
		$e.f$	*field*
e	::=		*Expression*
		`this`	*this*
		lval	*l-value*
		`new` t	*new*
		`null`	*null*
		$e.md(e_{i\in 1..n})$	*method call*
s	::=		*Statement*
		`skip;`	*skip*
		$t\ x = e$;	*variable declaration*
		e;	*expression*
		$lval = e$;	*update of lvalue*
		s_1; s_2	*sequence*
		`if` (e) `{` s_1 `}` `else` `{` s_2 `}`	*if-statement*
p, q	::=		*Owners*
		`this`	*this*
		α	*owner parameter*
		`world`	*world*
		`owner`	*owner*
t	::=		*Type*
		$p{:}c\langle p_{i\in 1..n}\rangle$	

3.2.1 Joline's Syntax

The syntax of Joline is displayed in Table 3.1. It is basically a subset of Java extended with ownership types and should be familiar to anyone with some experience of Java.

A program is a collection of classes followed by a statement and a resulting expression that are the equivalent of Java's main method. We could have followed Java's example and use a static main method etc., but chose this way out for simplicity.

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

3.2.2 Joline's Type System

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

Owner Substitutions

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

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

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

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

$$\begin{aligned} \sigma(p\!:\!c\langle p_{i\in 1..n}\rangle) &= \sigma(p)\!:\!c\langle \sigma(p_i)_{\ i\in 1..n}\rangle \\ \sigma(t \to t') &= \sigma(t) \to \sigma(t') \end{aligned}$$

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

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

As an illustration, if type $p\!:\!\mathtt{List}\langle q\rangle$ is formed from the class definition

class *List*< *data* **outside** *owner* > { . . . }

we sometimes write $p\!:\!\mathtt{List}\langle\sigma\rangle$ for the same type where $\sigma = \{\mathtt{data} \mapsto q\}$.

Field Look-up

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

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

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

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

Field look-up is formally defined as:

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

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

The σ on the third line is a map from the superclass' parameters to the parameters used in the subclass. When looking up a field variable for class c on the third line, the types in $\mathcal{F}_{\mathtt{c}'}$ use owner names in the class definition of class $\mathtt{c}'$. Thus we apply the substitution $\sigma(\mathcal{F}_{\mathtt{c}'}(\mathtt{f}))$ to bind the owner parameters of $\mathtt{c}'$ to the owners in c. This gives us a type for f in c.

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

Method Look-up

Method look-up is similar to the field look-up mechanism described above. For any class c, $\mathcal{M}_{\mathtt{c}}$ 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. Again, we give an example.

```
class Super< some outside owner > extends Object
{
    some : Object get(owner:Int pos) { ... }
}

class Example< data outside owner, other outside data >
                                        extends Super< other >
{
    data : Int add(data:Object obj) { ... }
}
```

Given these class definitions, we have the following map for *Example*:

$$\mathcal{M}_{Example} = \{get \mapsto (owner{:}Int \rightarrow other{:}Object), add \mapsto (data{:}Object \rightarrow data{:}Int)\}$$

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

Method look-up is formalised as:

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

(For the last case, we implicitly assum that *md* is not in the body of class c.) In coming chapters, this definition is extended to accept other kinds of parameters.

3.2.3 Well-formedness Rules

In this section, we present the static semantics of Joline. An overview of the different judgements used is found in Table 3.2. In the type rules, P is the complete program and a global constant for all 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.

Static Type Environment

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

$$E \quad ::= \quad \epsilon \quad | \quad E, x :: t \quad | \quad E, \alpha \succ^* p \quad | \quad E, \alpha \prec^* p$$

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

Good environment

$$\frac{}{\epsilon \vdash \diamond}\ (\textsc{env-}\epsilon) \qquad \frac{E \vdash t \qquad x \notin dom(E)}{E, x :: t \vdash \diamond}\ (\textsc{env-x})$$

Table 3.2: Judgements used in the static semantics.

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

$$\frac{\mathsf{E} \vdash \mathsf{p} \qquad \alpha \notin \mathit{dom}(\mathsf{E})}{\mathsf{E}, \alpha \succ^* \mathsf{p} \vdash \diamond}\ (\textsc{env-}\alpha \succ^*) \qquad \frac{\mathsf{E} \vdash \mathsf{p} \qquad \alpha \notin \mathit{dom}(\mathsf{E})}{\mathsf{E}, \alpha \prec^* \mathsf{p} \vdash \diamond}\ (\textsc{env-}\alpha \prec^*)$$

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

Good owner

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

The rules for good owners state that an owner is well-formed if it is defined in the static environment. Also, if present in the environment, the special variable `this` is also a good owner. The owner `world` is globally defined, and thus always valid.

Owner Orderings

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

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

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

Program and Class

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

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

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 map from owner names used in the class header of c' to

the owner names $\alpha_{i\in 1..m}$ used in c). Shadowing fields is not permitted as the names of the fields declared in c must not be in set of fields declared by any superclass to c. Any expression initialising a field must be valid under the class' environment E, constructed from the owner class' header, adding *owner* and the *this* variable. Finally, all methods declared in the class must be well-formed under E and, notably, for all overridden methods (a method with the same name as one defined in any superclass to c), the types must be invariant modulo σ-substitution which binds the names of the owner parameters of c' into the corresponding names in c.

Good method

$$\frac{\mathsf{E}, x_1 :: t_1, \ldots, x_m :: t_m \vdash s;\ \mathsf{E}' \qquad \mathsf{E}' \vdash e :: t_0}{\mathsf{E} \vdash t_0\ md(t_j\ x_{j\in 1..m})\{\ s\ \texttt{return}\ e;\ \}}\ (\textsc{method})$$

A method is well-formed under environment E if the statements and return expression of its body are well-formed with respect to E extended with the parameter variables declared in its header. Again, observe that (method) will be extended later in Section 4.3 when the method construct is extended with owner parameters.

Types

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

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

Subtyping

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

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

Subtyping is derived from subclassing, modulo names of the owner parameters. As this corresponds to the composition of two order-preserving functions, it is order-preserving. This is required to preserve deep ownership, see Clarke's dissertation [38]. In particular, subtyping preserves the owner that is fixed for life. Letting the owner vary, as in Cyclone [67], would be unsound in our system, as observed by Clarke and Drossopoulou [39].

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

As expected, the subtype relation is reflexive and transitive.

Statements

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

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

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

From (STAT-SKIP), `skip` is a valid statement under any valid environment. From (STAT-EXPR), a well-formed expression can be treated as a statement. The rule (STAT-UPDATE) simply enforces that updates can be performed to l-values only if the types match,

modulo subtyping of e. Recall that *lval* is either x or $x.f$.

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

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

l-values

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

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

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

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

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

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

The condition $\texttt{this} \in \mathsf{owners}(t) \Rightarrow e \equiv \texttt{this}$, which was called the *static visibility* in the original ownership types system [42], ensures that types that contain `this` in

them, that is types of representation objects, can only be accessed internally to the object. It amounts to saying that fields (and methods) which yield, return or require that representation objects are private. This is not essential; we could have used *dynamic aliasing* as in Joe$_1$ [39], but the resulting type system would have been too complex to present our ideas.

Expressions

$$\frac{}{\mathtt{this} :: t \in E \vdash \mathtt{this} :: t}\ (\textsc{expr-this}) \qquad \frac{}{E \vdash t \vdash \mathit{null} :: t}\ (\textsc{expr-null}) \qquad \frac{E \vdash e :: t \quad E \vdash t \leqslant t'}{E \vdash e :: t'}\ (\textsc{expr-subsumption})$$

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

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

By (EXPR-NEW), any well-formed class can be instantiated.

$$\frac{E \vdash e :: p{:}c\langle\sigma\rangle \qquad \mathcal{M}_c(md) = t_{j\in 1..m} \to t_0 \qquad \mathtt{this} \in \mathsf{owners}(\mathcal{M}_c(md)) \Rightarrow e \equiv \mathtt{this} \qquad E \vdash e_j :: \sigma'(\sigma^p(t_j)) \ \text{ for all } j \in 1..m}{E \vdash e.md(e_{j\in 1..m}) :: \sigma'(\sigma^p(t_0))}\ (\textsc{expr-call})$$

The rule for method performs a static visibility test, just as for field access, which restricts expressions containing *this* in their type (as declared in the class) to being used only internally, that is, on *this*. The owners of the target type forms a substitution to translate the owners in the method's argument's types and return types into the corresponding types using the owners in scope. The value supplied to each argument of the method must have the type expected by the method.

This concludes the description of Joline's static semantics. The upcoming section deals with the dynamics before getting into the soundness proofs on page 71.

3.3 JOLINE, DYNAMICALLY

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

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

3.3.1 Syntax definitions

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

The syntax for heaps, stacks and values are shown in Figure 3.3. We write $S \bullet$ to mean $S \bullet \mathsf{nil}$. Stacks and store types have parallel structure. Stacks, S, consist of ordered frames, F. Frames consist of variables $x \mapsto v$ and owners $\alpha \mapsto n$ ordered by $\oplus$. Figure 3.3 describes the syntax for stacks and stores. The syntactic category V denotes zero or more fields, $f \mapsto v$. The region construct models the region *world*, with a nested subheap. The symbol H denotes zero or more objects, $n \mapsto c^{\sigma}[V; H]$, in a nested subheap. Additionally, a region has a *nested frame*, and the $\oplus$ operator pushes its right-hand side to the innermost compartment of the left-hand side, just as in the store-type. Thus, $R_n[H; \mathsf{nil}] \oplus F$ is equivalent to $R_n[H; F]$.

We now describe the rules for well-typed configurations roughly in the same order as the syntactic definitions in Figure 3.3. An overview of the judgements is found in Table 3.3.

Additional syntax for the store type is shown in Figure 3.4.

3.3.2 Store Type

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

Syntax		terms:
$S ::=$		*stack*
	F	*frame*
	$S \bullet F$	
$F ::=$		*frame*
	nil	*nil*
	$x \mapsto v, F$	*variable*
	$\alpha \mapsto n, F$	*owner biding*
	$R_n[H; F]$	*region*
$H ::=$		*subheap*
	nil	*nil*
	$n \mapsto o, H$	*id to object*
$V ::=$		*fields*
	nil	*nil*
	$f \mapsto v, V$	*field*
$o ::=$		*object (shorthand)*
	$c^{\sigma}[V; H]$	*object*
$v ::=$		*value*
	null	*null*
	$\uparrow n$	*pointer*

Figure 3.3: Syntax for stacks, heaps and values.

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

Note that the owner of an object is not encoded in its type but is implicit in the nesting. Thus, an object of class c with owner parameters σ nested inside some object m will have the "owner-less type" $c\langle\sigma\rangle$ in the store type. However, we can derive its "complete type" $m{:}c\langle\sigma\rangle$ from the nesting. The type $\mathfrak{R}$ denotes a *region*, a container in which a store-type is nested. It is used in Joline to encode *world*, the top-most owner and can be thought of as the initial stack frame for the program's "main method", but its use will be extended in later chapters. For now, all store-type information is nested

Syntax		terms:
$\Gamma ::=$		*Store type*
	nil	*nil*
	$\mathfrak{n} :: \mathsf{T}[\Gamma], \Gamma'$	*object with subheap*
	$\mathfrak{x} :: \mathfrak{t}, \Gamma$	*variable*
	$\alpha \mapsto \mathfrak{n}, \Gamma$	*owner*
	$\Gamma \bullet \Gamma'$	*generation*
$\mathsf{T} ::=$		*Owner-less type*
	$\mathfrak{c}\langle\sigma\rangle$	*object type*
	$\mathfrak{R}$	*region*
$\Gamma_{\langle\,\rangle} ::=$		*Store type with hole*
	$\langle\,\rangle$	*hole*
	$\mathfrak{n} :: \mathsf{T}[\Gamma_{\langle\,\rangle}], \Gamma'$	
	$\mathfrak{n} :: \mathsf{T}[\Gamma], \Gamma'_{\langle\,\rangle}$	
	$\mathfrak{x} :: \mathfrak{t}, \Gamma_{\langle\,\rangle}$	
	$\alpha \mapsto \mathfrak{n}, \Gamma_{\langle\,\rangle}$	
	$\Gamma_{\langle\,\rangle} \bullet \Gamma'$	
	$\Gamma \bullet \Gamma'_{\langle\,\rangle}$	

Figure 3.4: Store type. We write $\Gamma \bullet$ to mean $\Gamma \bullet \mathsf{nil}$.

inside the region corresponding to world and no other regions exist in the system. Figure 3.5 shows a sample object graph and its corresponding store-type.

The syntactic category $\Gamma_{\langle\,\rangle}$ describes a store type with a hole. The syntax $\Gamma\langle\cdots\rangle_{\mathfrak{m}}$ means the stack Γ extended by $\cdots$ inside the subheap of some object $\mathfrak{m}$. We sometimes write $\Gamma = \Gamma'\langle H\rangle_{\mathfrak{m}}$ to mean that Γ can be factored into some stack Γ' with a *hole* in $\mathfrak{m}$ and H, which are part of the contents of $\mathfrak{m}$ in Γ, or, equivalent, that stack Γ' can be extended by H inside $\mathfrak{m}$ to yield stack Γ. As an example, if Γ is the resulting store-type of Figure 3.5, then Γ can be factored into, for example, $\Gamma'\langle o :: \mathfrak{c}_3\langle\sigma_3\rangle\rangle_{\mathfrak{m}}$, where Γ' equals Γ but where $o :: \mathfrak{c}_3\langle\sigma_3\rangle$ is replaced by a hole.

We now proceed by defining a few helper functions.

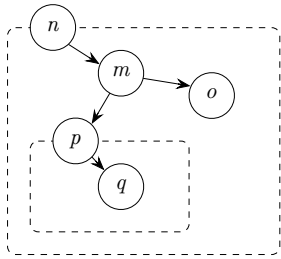

Figure 3.5: An object graph. Its corresponding store-type, for some owner-less types $c_i\langle\sigma_i\rangle$ for $i = 1..5$ is $n :: c_1\langle\sigma_1\rangle[\, m :: c_2\langle\sigma_2\rangle, o :: c_3\langle\sigma_3\rangle, p :: c_4\langle\sigma_4\rangle[\, q :: c_5\langle\sigma_5\rangle\,]\,]$. As is visible from the picture, the owner of objects m, o and p is n. This is mirrored by the nesting structure of store-type.

Definition of defs, owners and vars for Γ

We define the function $\mathsf{defs}(\Gamma)$ to be the set of all identities of objects typed in Γ and names of variables typed in Γ. Formally:

$$\begin{aligned}
\mathsf{defs}(\mathsf{nil}) &= \emptyset \\
\mathsf{defs}(n :: T[\Gamma], \Gamma') &= \{n\} \cup \mathsf{defs}(\Gamma) \cup \mathsf{defs}(\Gamma') \\
\mathsf{defs}(x :: t, \Gamma) &= \{x\} \cup \mathsf{defs}(\Gamma) \\
\mathsf{defs}(\alpha \mapsto n, \Gamma) &= \mathsf{defs}(\Gamma) \\
\mathsf{defs}(\Gamma \bullet \Gamma') &= \mathsf{defs}(\Gamma) \cup \mathsf{defs}(\Gamma')
\end{aligned}$$

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

$$\begin{aligned}
\mathsf{owners}(\mathsf{nil}) &= \emptyset \\
\mathsf{owners}(n :: T[\Gamma], \Gamma') &= \mathsf{owners}(\Gamma) \cup \mathsf{owners}(\Gamma') \\
\mathsf{owners}(x :: t, \Gamma) &= \mathsf{owners}(\Gamma) \\
\mathsf{owners}(\alpha \mapsto n, \Gamma) &= \{\alpha\} \cup \mathsf{owners}(\Gamma) \\
\mathsf{owners}(\Gamma \bullet \Gamma') &= \mathsf{owners}(\Gamma')
\end{aligned}$$

We define the function $\mathsf{vars}(\Gamma)$ to be the set of all variable names mapping on the topmost stack frame Γ.

$$
\begin{aligned}
\mathsf{vars}(\mathsf{nil}) &= \emptyset \\
\mathsf{vars}(\mathfrak{n} :: \mathsf{T}[\Gamma], \Gamma') &= \mathsf{vars}(\Gamma) \cup \mathsf{vars}(\Gamma') \\
\mathsf{vars}(\mathfrak{x} :: \mathfrak{t}, \Gamma) &= \{\mathfrak{x}\} \cup \mathsf{vars}(\Gamma) \\
\mathsf{vars}(\alpha \mapsto \mathfrak{n}, \Gamma) &= \mathsf{vars}(\Gamma) \\
\mathsf{vars}(\Gamma \bullet \Gamma') &= \mathsf{vars}(\Gamma')
\end{aligned}
$$

Definition of $\oplus$ for Γ

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

$$
\begin{aligned}
\mathsf{nil} \oplus \Gamma &= \Gamma \\
(\alpha \mapsto \mathfrak{n}, \Gamma') \oplus \Gamma &= \alpha \mapsto \mathfrak{n}, (\Gamma' \oplus \Gamma') \\
(\mathfrak{x} :: \mathfrak{t}, \Gamma') \oplus \Gamma &= \mathfrak{x} :: \mathfrak{t}, (\Gamma' \oplus \Gamma) \\
(\mathfrak{n} :: \mathfrak{R}[\Gamma']) \oplus \Gamma &= \mathfrak{n} :: \mathfrak{R}[\Gamma' \oplus \Gamma] \\
(\mathfrak{n} :: \mathfrak{c}\langle\sigma\rangle[\Gamma'], \Gamma'') \oplus \Gamma &= \mathfrak{n} :: \mathfrak{c}\langle\sigma\rangle[\Gamma'], (\Gamma'' \oplus \Gamma) \\
(\Gamma' \bullet \Gamma'') \oplus \Gamma &= \Gamma' \bullet (\Gamma'' \oplus \Gamma)
\end{aligned}
$$

Rules for Well-formed Store Type

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

By (STORE-TYPE-EMPTY), the empty store type is well-formed. By (GOOD-OWNER), $\mathfrak{p}$ is a good owner under Γ if it is in the set $\mathsf{defs}(\Gamma)$, the set of all ids of all objects, blocks and variables in Γ.

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

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

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

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

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

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

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

The (store-type-generation) rule governs the well-formedness of generations in Γ. A new generation corresponds to a stack frame created by method invocation on some reference $\uparrow n$. It contains the owners of type of $\uparrow n$, the static type of `this`, and a variable to $\uparrow n$ binding to make the receiver accessible on the frame.

Owner Orderings

Owner orderings are crucial to keeping the strong containment invariant of ownership types. Naturally, we can derive some of the orderings directly from the nesting of own-

ers in Γ. However, in presence of generations this becomes a little more complicated.

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

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

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

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

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

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

3.3.3 Configurations

As we saw in Figure 3.3, heaps are nested inside stacks. Starting configurations have the form $\langle S \mid s \rangle$, where S is a stack and s is a statement, and resulting configurations are either $\langle S \mid v \rangle$, where v is the resulting value of evaluating an expression, or $\langle S \rangle$, a single stack. The initial configuration where $s\,;e$ is the "main method" of the program is:

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

The rules for well-formed configurations are slightly unorthodox. To the left of the turnstile is the store-typing for the entire store. To the right of the $\gg$ operator is a subset of said store-typing, namely the subset that exactly corresponds to the structure which the judgement is typing. This will be used later to deal with certain visibility restrictions.

Definition of $\oplus$ for Ss and Fs

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

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

Configurations

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

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

Table 3.3: Table over judgements

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

$$\frac{\Gamma \vdash S \quad \Gamma \vdash e :: t}{\Gamma \vdash \langle S \mid e \rangle :: t}\ (\textsc{config-expr}) \qquad \frac{\Gamma \vdash S \quad \Gamma \vdash v :: t}{\Gamma \vdash \langle S \mid v \rangle :: t}\ (\textsc{config-val})$$

The rules (CONFIG-EXPR) and (CONFIG-VAL) are straightforward.

Stacks

$$\frac{}{\mathsf{nil} \vdash \mathsf{nil}}\ (\textsc{stack-empty}) \qquad \frac{\Gamma \vdash S \quad \Gamma \bullet \Gamma' \vdash F \gg \Gamma'}{\Gamma \bullet \Gamma' \vdash S \bullet F}\ (\textsc{stack-gen})$$

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

Frames

$$\frac{}{\Gamma \vdash \diamond}\ (\textsc{frame-empty}) \qquad \frac{\Gamma \vdash F \gg \Gamma' \quad \Gamma \vdash v :: t}{\Gamma \vdash F \oplus x \mapsto v \gg \Gamma' \oplus x :: t}\ (\textsc{variables}) \qquad \frac{\Gamma \vdash F \gg \Gamma'}{\Gamma \vdash F \oplus \alpha \mapsto n \gg \Gamma' \oplus \alpha \mapsto n}\ (\textsc{owners})$$

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

Heaps and objects

$$\frac{\Gamma \vdash m}{\Gamma; m \vdash \mathsf{nil} \gg \mathsf{nil}}\ (\textsc{heap-empty}) \qquad \frac{\Gamma; m \vdash n \mapsto o \gg \Gamma' \quad \Gamma; m \vdash H \gg \Gamma''}{\Gamma; m \vdash n \mapsto o, H \gg \Gamma', \Gamma''}\ (\textsc{heap-object})$$

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

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

By (OBJECT), an object is well-formed inside m under Γ if it has m as an owner in Γ, its subheap is well-formed inside the object itself, and its fields have good types using:

$$\frac{\Gamma \vdash v :: t \quad \Gamma \vdash V \gg \Gamma'}{\Gamma \vdash f \mapsto v, V \gg f :: t, \Gamma'}\ (\textsc{fields})$$

Fields work like variables, but their order is insignificant.

Values

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

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

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

$$
\frac{\Gamma \vdash t \quad \Gamma(m) = t}{\Gamma \vdash \uparrow m :: t}\ (\textsc{val-pointer}) \qquad
\frac{\Gamma \vdash t}{\Gamma \vdash \mathit{null} :: t}\ (\textsc{val-null}) \qquad
\frac{\Gamma \vdash v :: t' \quad \Gamma \vdash t' \leqslant t}{\Gamma \vdash v :: t}\ (\textsc{val-subsumption})
$$

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

Static and Dynamic Types

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

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

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

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

3.3.4 Operational Semantics for Joline

We now describe Joline's operational semantics.

Variable Look-up and Assignment

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

$$\begin{aligned}
\mathsf{nil}(x) &::= \bot \\
(S \bullet F)(x) &::= F(x) \\
(F \oplus \alpha \mapsto n)(x) &::= F(x) \\
(F \oplus y \mapsto \uparrow n)(x) &::= F(x) \qquad x \neq y \\
(F \oplus x \mapsto \uparrow n)(x) &::= \uparrow n
\end{aligned}$$

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

$$\begin{aligned}
\mathsf{nil}[x \mapsto v] &::= \bot \\
(S \bullet F)[x \mapsto v] &::= S \bullet (F[x \mapsto v]) \\
(x' \mapsto v', F)[x \mapsto v] &::= x' \mapsto v', (F[x \mapsto v]) \\
(x \mapsto v', F)[x \mapsto v] &::= x \mapsto v, F \\
(\alpha \mapsto n, F)[x \mapsto v] &::= \alpha \mapsto n, (F[x \mapsto v])
\end{aligned}$$

Field Look-up and Assignment

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

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

respective (for field update):

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

Dispatch

The help function $\mathcal{D}_t(md)$ returns a (b, t') tuple where $b = s\,\texttt{;return}\,e$ corresponding to the body of the method that the message *md* is bound to when passed to an object of type t, and t' is the type of the receiver, as viewed by the b body.

$$
\mathcal{D}_{p:c\langle\sigma\rangle}(md) = \begin{cases} \bot, & \text{if } c \equiv \texttt{Object} \\ (b, p:c\langle\sigma\rangle), & \text{if } \texttt{class}\, c \cdots \{\cdots \_\, md(\_)\{b\} \cdots\} \in P \\ \mathcal{D}_{p:c'\langle\sigma(\sigma')\rangle}(md), & \text{if } \texttt{class}\, c\langle\_\rangle\, \texttt{extends}\, c'\langle\sigma'\rangle \cdots \in P \end{cases}
$$

Just as is the definition of $\mathcal{M}_c$, we omit *md* not in the body of c in the bottom case, for presentation reasons.

Expressions

$$\frac{\text{(EXPR-THIS)}\quad S(\texttt{this}) = \uparrow n}{\langle S \mid \texttt{this}\rangle \rightarrow \langle S \mid \uparrow n\rangle} \qquad \frac{\text{(EXPR-NULL)}}{\langle S \mid \mathit{null}\rangle \rightarrow \langle S \mid \mathit{null}\rangle} \qquad \frac{\text{(EXPR-VAR)}\quad S(x) = v}{\langle S \mid x\rangle \rightarrow \langle S \mid v\rangle}$$

Without loss of generality, we use "named form" for the expressions in order to simplify the formal account of Joline. We only allow field look-up, field update and method calls to be performed local variables and not directly on the result of an expression. Thus, instead of writing $e.f$, we write $x = e; x.f$, where x is a variable of the appropriate type. Clearly, the forms are equivalent.

$$\frac{\text{(EXPR-FIELD)}\quad S(x) = \uparrow n \quad (S)_{n.f} = v}{\langle S \mid x.f\rangle \rightarrow \langle S \mid v\rangle}$$

Given the definitions of variable and field look-up above, (EXPR-FIELD) is straightforward. We look-up the value of x in S, which must be a pointer, and then perform a field look-up on field f on the appropriate object using the helper function $(S)_{n.f}$.

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

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

$$\frac{\begin{array}{c}\text{(EXPR-CALL)}\\ \langle S \mid e\rangle \to \langle S_1 \mid v\rangle \quad S_1(x) = \uparrow n \\ S_1 = S_2\langle n \mapsto c^{\sigma}[\_]\rangle_m \quad \mathcal{D}_{m:c\langle\sigma\rangle}(md) = (\mathtt{s;return}\ e', m{:}c_2\langle\sigma_2\rangle) \\ \langle S_1 \bullet {\sigma_2}_n^m \oplus \mathtt{this} \mapsto \uparrow n \oplus y \mapsto v \mid s\rangle \to \langle S_3\rangle \quad \langle S_3 \mid e'\rangle \to \langle S_4 \bullet F \mid v'\rangle\end{array}}{\langle S \mid x.md(e)\rangle \to \langle S_4 \mid v'\rangle}$$

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

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

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

Statements

$$\frac{\begin{array}{c}\text{(STAT-LOCAL)}\\ \langle S \mid e\rangle \to \langle S' \mid v\rangle\end{array}}{\langle S \mid t\ x = e\rangle \to \langle S' \oplus x \mapsto v\rangle} \qquad \frac{\begin{array}{c}\text{(STAT-UPDATE)}\\ \langle S \mid e\rangle \to \langle S' \mid v\rangle\end{array}}{\langle S \mid x := e\rangle \to \langle S'[x \mapsto v]\rangle}$$

Local variable declaration and initialisation is straightforward: a binding from the variable name to its value is appended to the stack. Local variable update is equally trivial and works as expected.

$$\frac{\begin{array}{c}\text{(UPDATE-FIELD)}\\ \langle S \mid e\rangle \to \langle S' \mid v\rangle \quad S'(x) = \uparrow n \qquad S'' = (S')_{n.f} := v\end{array}}{\langle S \mid x.f := e\rangle \to \langle S''\rangle}$$

Field update works as expected, using the previously described $(S)_{n.f} := v$ helper function.

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

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

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

3.4 SOUNDNESS OF THE JOLINE LANGUAGE

Just as the rest of the formal account of Joline's semantics, the presentation of the soundness proof is complicated by the upcoming extensions to the language with additional constructs to deal with unique pointers, etc. To avoid presenting two proofs, we omit certain details from this chapter that would otherwise be deprecated by the extended versions. A few constructs and lemmas should also be ignored upon a first read as they relate to constructs not yet introduced. We try to point these out.

3.4.1 Helper Functions

Definition of $\rightsquigarrow$

The symbol $\rightsquigarrow$ describes the relation between store-typings of different configurations. We only define $\rightsquigarrow$ for well-formed store-typings.

$$\begin{aligned}
\Gamma \vdash \diamond \text{ and } \Gamma \quad &\rightsquigarrow \quad \Gamma \\
\Gamma \vdash \diamond \text{ and } \Gamma \quad &\rightsquigarrow \quad \Gamma\langle n :: c\langle\sigma\rangle\rangle \text{ and } \Gamma\langle n :: c\langle\sigma\rangle\rangle \vdash \diamond \\
\Gamma \vdash \diamond \text{ and } \Gamma \quad &\rightsquigarrow \quad \Gamma' \text{ if there exists } \Gamma'' \text{ s.t. } \Gamma \rightsquigarrow \Gamma'' \text{ and } \Gamma'' \rightsquigarrow \Gamma'
\end{aligned}$$

Note that this means that within all existing stack frames, only new *objects* are added.

Definition of $\leqslant$

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

$$
\begin{array}{lcl}
\Gamma & \leqslant & \Gamma \\
\Gamma & \leqslant & \Gamma', \text{ if } \Gamma \rightsquigarrow \Gamma' \\
\Gamma & \leqslant & \Gamma \oplus x :: t \\
\Gamma & \leqslant & \Gamma \oplus \alpha \mapsto n \\
\Gamma & \leqslant & \Gamma \oplus n :: \mathfrak{R} \\
\Gamma & \leqslant & \Gamma \oplus n :: \mathfrak{B} \\
\Gamma & \leqslant & \Gamma \bullet n :: \mathfrak{R} \\
\Gamma & \leqslant & \Gamma \bullet n :: \mathfrak{B} \\
\Gamma & \leqslant & \Gamma', \text{ if there exists } \Gamma'' \text{ s.t. } \Gamma \leqslant \Gamma'' \text{ and } \Gamma'' \leqslant \Gamma'
\end{array}
$$

For now, ignore the cases containing $n :: \mathfrak{B}$. They correspond to constructs added in later chapters. We only define $\leqslant$ for well-formed store-typings, so $\Gamma \leqslant \Gamma'$ implies $\Gamma \vdash \diamond$ and $\Gamma' \vdash \diamond$ in our system. We omit this for brevity.

3.4.2 Lemmas

Many proofs or formulas in this chapter include "borrowing blocks" and unique pointers (introduced in Chapter 6), and other extensions. The reader can think of a borrowing block as a *region*, like the one corresponding to the all-enclosing, top-level owner *world* for now, but parts of the proof regarding non-introduced constructs should be skipped on a first reading.

The Extension lemma shows that extensions to a valid judgement produces another valid judgement.

Lemma 3.4.1 (Extension)**.**

1. *If* $\Gamma \oplus \Gamma'' \vdash F \gg \Gamma''$ *and* $\Gamma\langle\Gamma'\rangle_p \vdash \diamond$, *then* $\Gamma\langle\Gamma'\rangle_p \oplus \Gamma'' \vdash F \gg \Gamma''$.
2. *If* $\Gamma\langle\Gamma''\rangle_m; m \vdash n \mapsto o \gg \Gamma''$ *and* $\Gamma\langle\Gamma'\rangle_p \vdash \diamond$, *then* $(\Gamma\langle\Gamma'\rangle_p)\langle\Gamma''\rangle_m; m \vdash n \mapsto o \gg \Gamma''$.
3. *If* $\Gamma\langle\Gamma''\rangle_m; m \vdash H \gg \Gamma''$ *and* $\Gamma\langle\Gamma'\rangle_p \vdash \diamond$, *then* $(\Gamma\langle\Gamma'\rangle_p)\langle\Gamma''\rangle_m; m \vdash H \gg \Gamma''$.

4. *If* $\Gamma \vdash_m V :: \Gamma''$ *and* $\Gamma \leqslant \Gamma'$, *then* $\Gamma' \vdash_m V :: \Gamma''$.
5. *If* $\Gamma \vdash v :: t$ *and* $\Gamma \leqslant \Gamma'$, *then* $\Gamma' \vdash v :: t$.
6. *If* $\Gamma \vdash t$ *and* $\Gamma \leqslant \Gamma'$, *then* $\Gamma' \vdash t$.
7. *If* $\Gamma \vdash n$ *and* $\Gamma \leqslant \Gamma'$, *then* $\Gamma' \vdash n$.
8. *If* $\Gamma \vdash n \mathrel{R} m$ *and* $\Gamma \leqslant \Gamma'$, *then* $\Gamma' \vdash n \mathrel{R} m$.
9. *If* $\Gamma(n) = t$ *and* $\Gamma \leqslant \Gamma'$, *then* $\Gamma'(n) = t$.
10. *If* $\Gamma \vdash e :: t$ *and* $\Gamma \rightsquigarrow \Gamma'$, *then* $\Gamma' \vdash e :: t$.
11. *If* $\Gamma \vdash s \gg \Gamma \oplus \Gamma''$ *and* $\Gamma \rightsquigarrow \Gamma'$, *then* $\Gamma' \vdash s \gg \Gamma' \oplus \Gamma''$.

Note that $\Gamma' \vdash \diamond$ is implicit in $\Gamma \leqslant \Gamma'$ and $\Gamma \rightsquigarrow \Gamma'$. Also remember the unique variable names assumption that takes care of any potential name clashes due to uniques in F, o, H, V, v in (1-5) above.

Proof. Follows by straightforward derivation of the well-formedness rules. □

Lemma 3.4.2 (Well-formed construction)**.**

1. *If* $\Gamma \vdash p \mathrel{R} q$, *then* $\Gamma \vdash p$ *and* $\Gamma \vdash q$.
2. *If* $\Gamma \vdash \mathfrak{I}$, *then* $\Gamma \vdash \diamond$ *(*$\mathfrak{I}$ *means any possible judgement).*
3. *If* $\Gamma; p \vdash \mathfrak{I}$, *then* $\Gamma \vdash \diamond$ *and* $\Gamma \vdash p$ *(*$\mathfrak{I}$ *means any possible judgement).*
4. *If* $\Gamma \bullet \Gamma' \vdash \diamond$, *then* $\Gamma \vdash \diamond$.
5. *If* $\Gamma \oplus \Gamma' \vdash \diamond$, *then* $\Gamma \vdash \diamond$.
6. *If* $\Gamma \vdash v :: t$, *then* $\Gamma \vdash t$.
7. *If* $\Gamma \vdash \uparrow n :: t$, *then* $\Gamma \vdash n$ *and* $\Gamma \vdash t$.
8. *If* $\Gamma \vdash t$, *then* $\Gamma \vdash p$ *for all* $p \in \mathsf{owners}(t)$.
9. *If* $\Gamma \vdash S$ *and* $\Gamma(n) = p : c\langle\sigma\rangle$, *then* $S = S'\langle n \mapsto c^{\sigma}[V; H]\rangle_p$.

Proof. Follows by straightforward derivation of the well-formedness rules. □

Lemma 3.4.3 (Generation Removal)**.**

1. *If* $\Gamma \bullet \Gamma' \vdash t$ *and* $\Gamma \vdash p$, *then* $\Gamma \vdash t$ *where* $t = p{:}c\langle\sigma\rangle$ *or* $t = \mathtt{unique}_p{:}c\langle\sigma\rangle$.
2. *If* $\Gamma \bullet \Gamma' \vdash_n v :: t$ *and* $\Gamma \vdash p$, *then* $\Gamma \vdash_n v :: t$ *where* $t = p{:}c\langle\sigma\rangle$ *or* $t = \mathtt{unique}_p{:}c\langle\sigma\rangle$.
3. *If* $\Gamma \bullet \Gamma'; p \vdash H \gg \Gamma''$ *and* $\Gamma \vdash p$, *then* $\Gamma; p \vdash H \gg \Gamma''$.
4. *If* $\Gamma \vdash p \mathrel{R} q$, $\Gamma \bullet \Gamma' \vdash \diamond$, $\Gamma'(m) = p$ *and* $\Gamma'(n) = q$, *then* $\Gamma \bullet \Gamma' \vdash m \mathrel{R} n$.

Proof. Cases 1 and 4 are independent. Cases 2 and 3 are proven with mutual induction.

Case 1) By (CLASS) and (TYPE), $\Gamma \bullet \Gamma' \vdash p \prec^* q$ for all $q \in \mathit{rng}(\sigma)$. By (IN-*), this implies $\Gamma \vdash q$ and $\Gamma \vdash p \prec^* q$. Thus, $\Gamma \vdash t$.

Case 2) By case analysis on the shape of v. There are three cases: (a) $v = \mathit{null}$, (b) $v = \uparrow m$ and (c) $v = U_n[v'; H]$.

Case a) Follows immediately from (VAL-NULL) and case 1 of this lemma.

Case b) By (VAL-POINTER), $\Gamma \bullet \Gamma' \vdash t$ and $(\Gamma \bullet \Gamma')(m) = t$. By def. of lookup from Γ, $\Gamma(m) = t$. By case 1 of this lemma , $\Gamma \vdash t$. Thus, $\Gamma \vdash_n \uparrow m :: t$.

Case c) By (VAL-UNIQUE), $\Gamma_1 = \Gamma\langle n :: \mathfrak{U}[\Gamma_2]\rangle \bullet \Gamma'$ s.t. $\Gamma_1 \vdash_n \uparrow m :: n{:}c\langle\sigma\rangle$ and $\Gamma_1; n \vdash H \gg \Gamma_2$. Clearly, $\Gamma\langle n :: \mathfrak{U}[\Gamma_2]\rangle \vdash n$. Thus, by part 1 of this lemma, $\Gamma\langle n :: \mathfrak{U}[\Gamma_2]\rangle \vdash_n \uparrow m :: n{:}c\langle\sigma\rangle$ and $\Gamma\langle n :: \mathfrak{U}[\Gamma_2]\rangle; n \vdash_n H \gg \Gamma_2$ by induction. Thus, by (VAL-UNIQUE), $\Gamma \vdash_n U_n[\uparrow m; H] :: \mathtt{unique}_p{:}c\langle\sigma\rangle$.

Case 3) By induction on the shape of H. There are two cases, (a) $H = \mathsf{nil}$ and (b) $H = n \mapsto o, H'$.

Case a) Immediate from (HEAP-EMPTY).

Case b) By (HEAP-OBJECT), $\Gamma \bullet \Gamma'; p \vdash n \mapsto o \gg n :: c\langle\sigma\rangle[\Gamma_1]$ and $\Gamma \bullet \Gamma'; p \vdash H' \gg \Gamma_2$ where $\Gamma'' = n :: c\langle\sigma\rangle[\Gamma_1], \Gamma_2$. By (OBJECT), $o = c^{\sigma}[V; H'']$ s.t. $\Gamma \bullet \Gamma' \vdash \uparrow n :: p{:}c\langle\sigma\rangle$, $\Gamma \bullet \Gamma'; n \vdash H'' \gg \Gamma_1$ and $\Gamma \bullet \Gamma' \vdash_n V \gg \sigma^p_n(\mathcal{F}_c)$. By induction, $\Gamma \vdash \uparrow n :: p{:}c\langle\sigma\rangle$ and thus, $\Gamma \vdash n$. By induction hypothesis, $\Gamma; n \vdash H'' \gg \Gamma_1$. By (TYPE) and (CLASS), $\Gamma \vdash n \prec^* q$ for all $q \in \mathit{rng}(\sigma^p)$. Thus, $\Gamma \vdash q$ for all such q's. Thus, by induction $\Gamma \bullet \vdash_n V \gg \sigma^p_n(\mathcal{F}_c)$. By (OBJECT),

$\Gamma; p \vdash n \mapsto o \gg n :: c\langle\sigma\rangle[\Gamma_1]$. By induction, $\Gamma; p \vdash H' \gg \Gamma_2$ and thus, by (HEAP-OBJECT), $\Gamma; p \vdash H \gg \Gamma'$.

Case 4) Follows immediately from (IN-OWNER-EQUAL).

□

Look-up Lemmas

This section presents lemmas dealing with the look-up of variables, fields and objects on a stack.

Lemma 3.4.4 (Variable Look-up)**.** *If* $\Gamma \vdash S$, $\Gamma \vdash x :: t$ ***ref*** *and* $S(x) = v$, *then* $\Gamma \vdash_x v :: t$.

Proof. By case analysis on the shape of S. There are two cases, as S clearly cannot be nil: (a) $S = F$ and (b) $S = S' \bullet F$.

Case a) Assume $\Gamma \vdash F$, $\Gamma \vdash x :: t$ **ref** and $F(x) = v$. By (STACK-GEN), $\Gamma \vdash F \gg \Gamma$. By (LVAL-VAR), $x :: t \in \Gamma$. The proof relies on the following fact.

> If $\Gamma \vdash F \gg \Gamma'$, $x :: t \in \Gamma'$ and $F(x) = v$, then $\Gamma \vdash_x v :: t$.

This fact follows simply by induction on the shape of F, observing that if $F = F' \oplus x \mapsto v$, then by (VARIABLES), $\Gamma \vdash F' \gg \Gamma''$ and $\Gamma \vdash_x v :: t$ where $\Gamma' = \Gamma'' \oplus x :: t$.

Case b) Assume $\Gamma \vdash S' \bullet F$, $\Gamma \vdash x :: t$ **ref** and $(S' \bullet F)(x) = v$. By (STACK-GEN), $\Gamma' \vdash S'$ and $\Gamma \vdash F \gg \Gamma''$ where $\Gamma = \Gamma' \bullet \Gamma''$. By (LVAL-VAR), $x :: t \in \Gamma''$ By def. of $S(\ )$, $F(x) = v$. The rest is similar to Case a).

□

Lemma 3.4.5 (Object Look-up)**.** *If* $\Gamma \vdash S$, *and* $\Gamma(n) = p : c\langle\sigma\rangle$, *and* $S(n) = o$, *then* $\Gamma; p \vdash n \mapsto o \gg n :: c\langle\sigma\rangle[\Gamma_n]$, *for some* Γ_n.

Proof. By case analysis on the shape of S. There are two cases, as S clearly cannot be nil: (a) $S = F$ and (b) $S = S' \bullet F$.

Case a) Assume $\Gamma \vdash F$, and $\Gamma(n) = p{:}c\langle\sigma\rangle$, and $S(n) = o$. By (STACK-GEN), $\Gamma \vdash F \gg \Gamma$. Clearly, $F(n) = o$. Proof continues by induction on the shape of F. The cases $F = F' \oplus \alpha \mapsto m$ and $F = F' \oplus x \mapsto v$ follow by induction. The cases $F = R_m[H; F']$ and $F = B^b_m[H; F']$ are similar and rely on the following fact:

> If $\Gamma; q \vdash H \gg \Gamma'$, and $\Gamma(n) = p{:}c\langle\sigma\rangle$, and $H(n) = o$, then $\Gamma; p \vdash n \mapsto o \gg n :: c\langle\sigma\rangle[\Gamma_n]$, for some Γ_n.

This fact follows simply by induction on the shape of H observing that if $H = n \mapsto o, H'$, then by (HEAP-OBJECT), $\Gamma; q \vdash n \mapsto o \gg \Gamma_1$ and $\Gamma; q \vdash H' \gg \Gamma_2$, where $\Gamma' = \Gamma_1, \Gamma_2$. By (OBJECT), $p = q$ and $\Gamma_1 = n :: c\langle\sigma\rangle[\Gamma_n]$. Thus $\Gamma; p \vdash n \mapsto o \gg n :: c\langle\sigma\rangle[\Gamma_n]$.

Case b) Assume $\Gamma \vdash S' \bullet F$, and $\Gamma(n) = p{:}c\langle\sigma\rangle$, and $S(n) = o$. By (STACK-GEN), $\Gamma' \vdash S'$ and $\Gamma \vdash F \gg \Gamma''$ where $\Gamma = \Gamma' \bullet \Gamma''$. By def. of $S(\ )$, there are two cases, (i) $S'(n) = o$ and (ii) $F(n) = o$. Case (i) follows by induction. Case (ii) is similar to Case a) above.

□

Lemma 3.4.6 (Field lookup). *If* $\Gamma \vdash S$, $\Gamma \vdash x :: p{:}c\langle\sigma\rangle$, $t = \sigma^p(\mathcal{F}_c(f))$, $\texttt{this} \in \mathsf{owners}(\mathcal{F}_c(f)) \Rightarrow x \equiv \texttt{this}$, $S(x) = \uparrow n$ *and* $(S)_{n.f} = v$, *then* $S \vdash_{n.f} v :: t$.

Proof.

1. By Lemma 3.4.4 (Variable Look-up), $\Gamma \vdash \uparrow n :: p{:}c\langle\sigma\rangle$.
2. By 1.) (VAL-POINTER) and (VAL-SUBSUMPTION),
   (a) $\Gamma(n) = p{:}c_1\langle\sigma_1\rangle$ where
   (b) $\Gamma \vdash p{:}c_1\langle\sigma_1\rangle \leqslant p{:}c\langle\sigma\rangle$.
3. By 2.a) and Lemma 3.4.5 (Object Look-up), $\Gamma; p \vdash n \mapsto {c_1}^{\sigma_1}[V; H] \gg c_1\langle\sigma_1\rangle[\Gamma_1]$, (Note that $(S)_{n.f} = v$ implies $S(n) = o$, where $o = {c_1}^{\sigma_1}[V; H]$ for some c_1, σ, V and H.)
4. By 3.a), and (OBJECT),
   (a) $\Gamma(n) = p{:}c_1\langle\sigma_1\rangle$,

(b) $\Gamma \vdash_n V :: \sigma_1{}^p_n(\mathcal{F}_{c_1})$.

(c) $\Gamma; n \vdash H \gg \Gamma_1$.

5. By 4.b) and (FIELDS), $\Gamma \vdash_{n.f} v :: \sigma_1{}^p_n(\mathcal{F}_{c_1}(f))$.

6. By 2.b) and Lemma 3.4.7 (See just below), $\sigma^p(\mathcal{F}_c(f)) = \sigma_1{}^p(\mathcal{F}_{c_1}(f))$.

7. By Lemma 3.4.8 (See just below), if $\texttt{this} \in \mathsf{owners}(\mathcal{F}_c(f))$, then $\Gamma \vdash n = \texttt{this}$. Thus, if $\texttt{this} \in \mathsf{owners}(t)$, $\Gamma \vdash \sigma^p(\mathcal{F}_c(f)) = \sigma_1{}^p_n(\mathcal{F}_{c_1}(f))$ by 6.).

8. By 5-7.), $\Gamma \vdash_{n.f} v :: \sigma^p(\mathcal{F}_c(f))$.

□

Lemma 3.4.7 (Subtyping Preserves Field Typing)**.** *If* $\Gamma \vdash p : c\langle\sigma\rangle \leqslant p : c_1\langle\sigma_1\rangle$ *and* $\mathcal{F}_c(f) \neq \bot$*, then* $\sigma^p(\mathcal{F}_c(f)) = \sigma_1^p(\mathcal{F}_{c_1}(f))$

Proof. Follows immediately from the def. of field look-up and (SUB-CLASS). □

Lemma 3.4.8. *If* $\Gamma \vdash S$ *and* $S(\texttt{this}) = \uparrow n$*, then* $\Gamma \vdash \texttt{this} = n$.

Proof. Follows immediately from def. of variable look-up and (STORE-TYPE-GENERATION). □

Update Lemmas

The subsequent lemmas deal with updating variables on the stack.

Lemma 3.4.9 (Variable Update)**.** *If* $\Gamma \vdash S$*, and* $\Gamma \vdash x :: t$ ***ref****, and* $\Gamma \vdash_x v :: t$*, then* $\Gamma \vdash S[x \mapsto v]$.

Proof. By induction case analysis on the shape of S. There are two cases, as S clearly cannot be nil: (a) $S = F$ and (b) $S = S' \bullet F$.

Case a) Assume $\Gamma \vdash F$, and $\Gamma \vdash x :: t$ **ref**, and $\Gamma \vdash_x v :: t$. By (STACK-GEN), $\Gamma \vdash F \gg \Gamma$. By (LVAL-VAR), $x :: t \in \Gamma$. The proof relies on the following fact:

> If $\Gamma \vdash F \gg \Gamma'$, and $x :: t \in \Gamma'$, and $\Gamma \vdash_x v :: t$, then
> $\Gamma \vdash F[x \mapsto v] \gg \Gamma'$.

The fact follows simply by induction on the shape of F, observing that if $F = F' \oplus x \mapsto v'$, then by (VARIABLES), $\Gamma \vdash F' \gg \Gamma''$ and $\Gamma \vdash_x v' :: t$ where $\Gamma' = \Gamma'' \oplus x :: t$. By (VARIABLES), $\Gamma \vdash F' \oplus x \mapsto v \gg \Gamma'$, which is equivalent to $\Gamma \vdash F[x \mapsto v] \gg \Gamma'$.

Case b) Assume $\Gamma \vdash S' \bullet F$, and $\Gamma \vdash x :: t$ **ref**, and $\Gamma \vdash_x v :: t$. By (STACK-GEN), $\Gamma' \vdash S'$ and $\Gamma \vdash F \gg \Gamma''$ where $\Gamma = \Gamma' \bullet \Gamma''$. By (LVAL-VAR), $x :: t \in \Gamma''$. The rest is similar to Case a).

□

Lemma 3.4.10 (Object Update)**.** *If* $\Gamma \vdash S\langle n \mapsto o\rangle$, $\Gamma; p \vdash n \mapsto o \gg c\langle\sigma\rangle[\Gamma']$, *and* $\Gamma; p \vdash n \mapsto o' \gg n :: c\langle\sigma\rangle[\Gamma', \Gamma'']$, *then* $\Gamma\langle\Gamma''\rangle_n \vdash S\langle n \mapsto o'\rangle$.

Proof. By case analysis on the shape of $S_{\langle\,\rangle}$. There are three cases: (a) $S_{\langle\,\rangle} = F_{\langle\,\rangle}$, (b) $S_{\langle\,\rangle} = S'_{\langle\,\rangle} \bullet F$ and (c) $S_{\langle\,\rangle} = S' \bullet F_{\langle\,\rangle}$.

Case a) Assume $\Gamma \vdash S\langle n \mapsto o\rangle$, $\Gamma; p \vdash n \mapsto o \gg c\langle\sigma\rangle[\Gamma']$, and $\Gamma; p \vdash n \mapsto o' \gg n :: c\langle\sigma\rangle[\Gamma', \Gamma'']$. By (STACK-GEN), $\Gamma \vdash F\langle n \mapsto n\rangle \gg \Gamma$. Proof continues by induction on the shape of $F_{\langle\,\rangle}$. The cases $F = F'_{\langle\,\rangle} \oplus \alpha \mapsto n$, $F = F'_{\langle\,\rangle} \oplus x' \mapsto v'$, $F = R_m[H; F'_{\langle\,\rangle}]$ and $F = B^b_m[H; F'_{\langle\,\rangle}]$ follow by induction. The cases $F = R_m[H_{\langle\,\rangle}; F']$ and $F = B^b_m[H_{\langle\,\rangle}; F']$ are similar and rely on the following fact:

> If $\Gamma; q \vdash H\langle n \mapsto o\rangle \gg \Gamma'$, $\Gamma; p \vdash n \mapsto o \gg c\langle\sigma\rangle[\Gamma_n]$, and
> $\Gamma; p \vdash n \mapsto o' \gg n :: c\langle\sigma\rangle[\Gamma_n, \Gamma'']$, then
> $\Gamma\langle\Gamma''\rangle_n; q \vdash H\langle n \mapsto o'\rangle \gg \Gamma'\langle\Gamma''\rangle_n$.

This fact follows by induction on the shape of $H_{\langle\,\rangle}$. The cases: $H = n'' \mapsto o''$, $H'_{\langle\,\rangle}$ and $H = n'' \mapsto o''_{\langle\,\rangle}$, H' follow by induction. The case $H = \langle\,\rangle$ is immediate. The case $H = \langle\,\rangle, H'$ follows by the following reasoning: Clearly, $p = q$. By (HEAP-OBJECT), $\Gamma; q \vdash n \mapsto o \gg \Gamma_1$ and $\Gamma; q \vdash H' \gg \Gamma_2$ where $\Gamma' = \Gamma_1, \Gamma_2$. Clearly, $n \notin \mathsf{defs}(\Gamma_2)$. Thus, by Lemma 3.4.1 (Extension), $\Gamma\langle\Gamma''\rangle_n; q \vdash H' \gg \Gamma_2$. By (HEAP-OBJECT), $\Gamma\langle\Gamma''\rangle_n; q \vdash n \mapsto o', H' \gg \Gamma'\langle\Gamma''\rangle_n$.

Case b) Assume $\Gamma \vdash S'\langle n \mapsto o\rangle \bullet F$, $\Gamma; p \vdash n \mapsto o \gg c\langle\sigma\rangle[\Gamma']$, and $\Gamma; p \vdash n \mapsto o' \gg n :: c\langle\sigma\rangle[\Gamma', \Gamma'']$. By (STACK-GEN), $\Gamma_1 \vdash S'\langle n \mapsto o\rangle$ and $\Gamma \vdash F \gg \Gamma_2$, where $\Gamma = \Gamma_1 \bullet \Gamma_2$. The result follows by induction.

Case c) Assume $\Gamma \vdash S' \bullet F\langle n \mapsto o\rangle, \Gamma; p \vdash n \mapsto o \gg c\langle\sigma\rangle[\Gamma']$, and $\Gamma; p \vdash n \mapsto o' \gg n :: c\langle\sigma\rangle[\Gamma', \Gamma'']$. By (STACK-GEN), $\Gamma_1 \vdash S'$ and $\Gamma \vdash F\langle n \mapsto o\rangle \gg \Gamma_2$, where $\Gamma = \Gamma_1 \bullet \Gamma_2$. The rest is similar to case a).

□

Lemma 3.4.11 (Field Update). *If* $\Gamma \vdash S, S(n) = o, \Gamma \vdash \uparrow n :: p : c\langle\sigma\rangle$ *and* $\Gamma \vdash_{n.f} v :: \sigma_n^p(\mathcal{F}_c(f))$, *then* $\Gamma \vdash (S)_{n.f} := v$.

Proof.

1. By (VAL-POINTER) and (VAL-SUBSUMPTION),
   (a) $\Gamma(n) = p : c_1\langle\sigma_1\rangle$ s.t.
   (b) $\Gamma \vdash p : c_1\langle\sigma_1\rangle \leqslant p : c\langle\sigma\rangle$.
2. By 1.b) and def. of field lookup, ${\sigma_1}_n^p(\mathcal{F}_{c_1}(f)) = \sigma_n^p(\mathcal{F}_c(f))$.
3. By 1.a) and Lemma 3.4.5 (Object Look-up), $\Gamma; p \vdash o \gg p : c_1\langle\sigma_1\rangle[\Gamma']$.
4. By 3.) and (OBJECT),
   (a) $\Gamma(n) = p : c_1\langle\sigma_1\rangle$,
   (b) $\Gamma \vdash V :: {\sigma_1}^p(\mathcal{F}_{c_1})$ and
   (c) $\Gamma; n \vdash H \gg \Gamma'$ where
   (d) $o = {c_1}^{\sigma_1}[V; H]$.
5. By 4.b) and (FIELDS),
   (a) $\Gamma \vdash_{n.f} v' :: {\sigma_1}_n^p(\mathcal{F}_{c_1}(f))$ and
   (b) $\Gamma \vdash_n V' :: \Gamma''$ where
   (c) $V = f \mapsto v', V'$ and
   (d) ${\sigma_1}^p(\mathcal{F}_{c_1}(f)) = f :: {\sigma_1}^p(\mathcal{F}_{c_1}(f)), \Gamma''$.
6. By 2.), 5.b-d) and (FIELD), $\Gamma \vdash_n V[f \mapsto v] :: {\sigma_1}^p(\mathcal{F}_{c_1})$.
7. By 4.a,c-d), 6.) and (OBJECT), $\Gamma; p \vdash o[f \mapsto v] \gg p : c_1\langle\sigma_1\rangle[\Gamma']$.
8. By 3.), 7.) and Lemma 3.4.10 (Object Update), $\Gamma \vdash (S)_{n.f} := v$.

□

3.4.3 Subject Reduction

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

Theorem 3.4.12 (Subject Reduction)**.**

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

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

Proof. By structural induction over the shapes of e and s. Note that the owner free never appears in Γ; it is clearly not in Γ in the starting configuration, and when a unique with free as id is stored in a field or variable, the id always changes to that of the field or variable.

Case (EXPR-VAR) Assume $\Gamma \vdash \langle S \mid x \rangle :: t$.

1. By (CONFIG-EXPR),
   (a) $\Gamma \vdash S$ and
   (b) $\Gamma \vdash x :: t$.
2. By 1.b) and (EXPR-LVAL),
   (a) $\Gamma \vdash x :: t$ **ref** and
   (b) $\neg\mathsf{isunique}(t)$.
3. By 1.a), 2.a), Lemma 3.4.4 (Variable Lookup), $\Gamma \vdash_x v :: t$.
4. By 2.b), 3.) and Lemma 6.5.1 (Omit qualifiers, not yet introduced), $\Gamma \vdash_{\text{free}} v :: t$.
5. By 1.a), 4.) and (CONFIG-VAL), $\Gamma \vdash \langle S \mid v \rangle :: t$.

Case (EXPR-THIS) Proof is similar to (EXPR-VAR).

Case (STAT-UPDATE) Assume $\Gamma \vdash \langle S \mid x := e \rangle$.

1. By (CONFIG-STAT),
   (a) $\Gamma \vdash S$ and
   (b) $\Gamma \vdash x := e \,;\Gamma$.
2. By 1.b) and (STAT-UPDATE),
   (a) $\Gamma \vdash x :: t$ **ref** and
   (b) $\Gamma \vdash e :: t$.
3. By 1.a), 2.b) and (CONFIG-EXPR), $\Gamma \vdash \langle S \mid e \rangle :: t$.
4. By 3.) and the induction hypothesis, if $\langle S \mid e \rangle \to \langle S' \mid v \rangle$, there exists a Γ' such that
   (a) $\Gamma \rightsquigarrow \Gamma'$ and
   (b) $\Gamma' \vdash \langle S' \mid v \rangle :: t$.
5. By 4.b) and (CONFIG-VAL),
   (a) $\Gamma' \vdash S'$ and
   (b) $\Gamma' \vdash_{\text{free}} v :: t$.
6. By 2.a), 4.a) and Lemma 3.4.1 (Extension), $\Gamma' \vdash x :: t$ **ref**.
7. By 5.b) and Lemma 6.5.1 (Omit qualifiers), $\Gamma' \vdash_x v :: t$.
8. By 5.a), 6.), 7.) and Lemma 3.4.9 (Variable Update), $\Gamma' \vdash S'[x \mapsto v]$.
9. By 8.) and (CONFIG-FINAL), $\Gamma' \vdash \langle S'[x \mapsto v] \rangle$.

Case (EXPR-NULL) Immediate.

Case (EXPR-FIELD) Assume $\Gamma \vdash \langle S \mid x.f \rangle :: t$.

1. By (CONFIG-EXPR),
   (a) $\Gamma \vdash S$, and
   (b) $\Gamma \vdash x.f :: t$.
2. By 1.b) and (EXPR-LVAL),
   (a) $\Gamma \vdash x.f :: t$ **ref**.
   (b) $\neg \mathsf{isunique}(t)$.

3. By 2.a), and (LVAL-FIELD),
   (a) $\Gamma \vdash x :: p\!:\!c\langle\sigma\rangle$,
   (b) $\sigma^p(\mathcal{F}_c(f)) = t$, and
   (c) $\texttt{this} \in \mathsf{owners}(\mathcal{F}_c(f)) \Rightarrow x \equiv \texttt{this}$.
4. By 1.a), 3.a,b,c), and Lemma 3.4.6 (Field Look-up), $\Gamma \vdash_{n.f} v :: t$.
5. By 2.b), 4.) and Lemma 6.5.1 (Omit qualifiers), $\Gamma \vdash_{\text{free}} v :: t$.
6. By 1.a), 5.) and (CONFIG-VAL), $\Gamma \vdash \langle S \mid v\rangle :: t$.

Case (UPDATE-FIELD) Assume $\Gamma \vdash \langle S \mid x.f := e\rangle$.

1. By (CONFIG-STAT),
   (a) $\Gamma \vdash S$, and
   (b) $\Gamma \vdash x.f := e; \Gamma$.
2. By 1.b) and (STAT-UPDATE),
   (a) $\Gamma \vdash x.f :: t$ **ref**, and
   (b) $\Gamma \vdash e :: t$.
3. By 1.a), 2.b) and (CONFIG-EXPR), $\Gamma \vdash \langle S \mid e\rangle :: t$.
4. By 3.) and induction hypothesis, if $\langle S \mid e\rangle \to \langle S' \mid v\rangle$, then there exists Γ' s.t.
   (a) $\Gamma \rightsquigarrow \Gamma'$ and
   (b) $\Gamma' \vdash \langle S' \mid v\rangle$.
5. By 2.a), 4.a) and Lemma 3.4.1 (Extension), $\Gamma' \vdash x.f :: t$ **ref**.
6. By 5.) and (LVAL-FIELD),
   (a) $\Gamma' \vdash x :: p\!:\!c\langle\sigma\rangle$,
   (b) $\sigma^p(\mathcal{F}_c(f)) = t$, and
   (c) $\texttt{this} \in \mathsf{owners}(\mathcal{F}_c(f))$ implies $x \equiv \texttt{this}$.
7. By 4.b) and (CONFIG-VAL),
   (a) $\Gamma' \vdash S'$ and
   (b) $\Gamma' \vdash_{\text{free}} v :: t$.
8. By 6.a), 7.a) and Lemma 3.4.4 (Variable Look-up), $\Gamma' \vdash_x \uparrow n :: p\!:\!c\langle\sigma\rangle$.
9. By 6.a,b,c), 7.a,b), 8.) and Lemma 3.4.11 (Field Update), $\Gamma' \vdash (S')_{n.f} := v$.
10. By 9.) and (CONFIG-STAT), $\Gamma' \vdash \langle (S')_{n.f} := v\rangle$.

Case (STAT-SKIP) Assume $\Gamma \vdash \langle S \mid \texttt{skip} \rangle$.

1. By (CONFIG-STAT),
   (a) $\Gamma \vdash S$ and
   (b) $\Gamma \vdash \texttt{skip} \gg \Gamma$.
2. By 1.a) and (CONFIG-FINAL), $\Gamma \vdash \langle S \rangle$

Case (STAT-LOCAL) Assume $\Gamma \vdash \langle S \mid \mathrm{t}\ x = e \rangle$.

1. By (CONFIG-STAT),
   (a) $\Gamma \vdash S$, and
   (b) $\Gamma' \vdash \mathrm{t}\ x = e$; Γ where $\Gamma = \Gamma' \oplus x :: \mathrm{t}$
2. By 1.b) and Lemma 3.4.2, and (STAT-LOCAL),
   (a) $x \notin \mathsf{vars}(\Gamma')$ and
   (b) $\Gamma' \vdash e :: \mathrm{t}$.
3. By 1.a), 2.b) and (CONFIG-EXPR), $\Gamma' \vdash \langle S \mid e \rangle :: \mathrm{t}$.
4. By 3.) and the induction hypothesis, if $\langle S \mid e \rangle \to \langle S' \mid v \rangle$, then there exists a Γ'' s.t.
   (a) $\Gamma \rightsquigarrow \Gamma''$ and
   (b) $\Gamma'' \vdash \langle S' \mid v \rangle :: \mathrm{t}$.
5. By 4.b) and (CONFIG-VAL),
   (a) $\Gamma'' \vdash S'$, and
   (b) $\Gamma'' \vdash_{\text{free}} v :: \mathrm{t}$.
6. By 5.a) and (STACK-GEN),
   (a) $\Gamma_1 \vdash S''$ and
   (b) $\Gamma'' \vdash F \gg \Gamma_2$ where
   (c) $\Gamma'' = \Gamma_1 \bullet \Gamma_2$ and $S' = S' \bullet F$.
7. By 5.b) and Lemma 6.5.1 (Omit Qualifiers), $\Gamma'' \vdash_x v :: \mathrm{t}$.
8. By 6.b), 7.) and (VARIABLES), $\Gamma'' \oplus x :: \mathrm{t} \vdash F \oplus x \mapsto v' \gg \Gamma_2 \oplus x :: \mathrm{t}$.
9. By 6.a,c), 8.) and (STACK-GEN), $\Gamma'' \oplus x :: \mathrm{t} \vdash S' \oplus x \mapsto v'$.
10. By 9.) and (CONFIG-FINAL), $\Gamma'' \oplus x :: \mathrm{t} \vdash \langle S' \oplus x \mapsto v' \rangle$.

Case (STAT-EXPR) Assume $\Gamma \vdash \langle S \mid e \rangle :: t$.

1. By the induction hypothesis, if $\langle S \mid e \rangle \to \langle S' \mid v \rangle$, then there exists a Γ' s.t.,
   (a) $\Gamma \rightsquigarrow \Gamma'$ and
   (b) $\Gamma' \vdash \langle S' \mid v \rangle :: t$.
2. By 1.b) and (CONFIG-VAL),
   (a) $\Gamma' \vdash S'$ and
   (b) $\Gamma' \vdash v :: t$.
3. By 2.a) and (CONFIG), $\Gamma' \vdash \langle S' \rangle$.

Case (STAT-SEQUENCE) Assume $\gg \Gamma \vdash \langle S \mid s_1 ; s_2 \rangle$

1. By (CONFIG-STAT),
   (a) $\Gamma \vdash S$, and
   (b) $\Gamma' \vdash s_1 ; s_2 \gg \Gamma$ where $\Gamma = \Gamma' \oplus \Gamma''$.
2. By 1.b) and (STAT-SEQUENCE),
   (a) $\Gamma' \vdash s_1 ; \Gamma_1$, and
   (b) $\Gamma_1 \vdash s_2 ; \Gamma$.
3. By 1.a), 2.a) and (CONFIG-STAT), $\Gamma_1 \vdash \langle S \mid s_1 \rangle$.
4. By induction hypothesis, if $\langle S \mid s_1 \rangle \to \langle S'' \rangle$, then there exists a Γ_2 s.t.,
   (a) $\Gamma_1 \rightsquigarrow \Gamma_2$ and
   (b) $\Gamma_2 \vdash \langle S'' \rangle$.
5. By 4.b) and (CONFIG-FINAL), $\Gamma_2 \vdash S''$.
6. By 2.b), 4.a) and Lemma 3.4.1 (Extension), $\Gamma_2 \vdash s_2 \gg \Gamma_3$ (where Γ_3 is Γ_2 extended with the delta between Γ_1 and Γ).
7. By 5.), 6.) and (CONFIG-STAT), $\Gamma_3 \vdash \langle S'' \mid s_2 \rangle$.
8. By 7.) and the induction hypothesis, if $\langle S'' \mid s_2 \rangle \to \langle S' \rangle$, then there exists a Γ_4 s.t.,
   (a) $\Gamma_3 \oplus \Gamma_4$ and
   (b) $\Gamma_4 \vdash \langle S' \rangle$.

Case (EXPR-NEW) Assume $\Gamma \vdash \langle S \mid \texttt{new}\ p{:}c\langle\sigma\rangle\rangle :: p{:}c\langle\sigma\rangle$.

We ignore this case for now, as (EXPR-NEW) is updated to deal with uniqueness in Chapter 6.

Case (EXPR-CALL) Chapter 4 will extend method definitions and method invocations. We thus postpone the proof of this case to the formal treatment in that chapter.

□

3.4.4 Canonical Forms

Lemma 3.4.13 (Canonical Forms)**.** *If* $\Gamma \vdash S$ *and* $\Gamma \vdash v :: t$*, then the following holds for the possible forms of* v*:*

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

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

From $\Gamma \vdash S$ and $\Gamma(n) = p{:}c'\langle\sigma'\rangle$, clearly $S(n) = o$, for some o, as Γ and S are parallel. By Lemma 3.4.5, $\Gamma; p \vdash n \mapsto o \gg \Gamma'$ for some Γ'. By (OBJECT), $f \in dom(V)$ for all $f \in dom(\mathcal{F}_{c'})$ where $o = c'^{\sigma'}[V; H]$.

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

□

3.4.5 Progress

In this section, we present the progress lemma. First, however, we introduce additional evaluation rules for the dynamic semantics, that deal with trapping and propagating errors in the system. For now, we trap only one kind of errors, null-pointer errors. (Please note that some of these error handling rules deal with constructs not yet introduced.)

Error Trapping Rules

$$\frac{\mathsf{S}(\texttt{this}) = \mathit{null}}{\langle \mathsf{S} \mid \texttt{this} \rangle \longrightarrow \langle \mathsf{S} \mid \texttt{ERROR} \rangle} \text{(EXPR-THIS)}$$

The rule (EXPR-THIS) captures an attempt to look-up `this` is a context where this is not defined.

$$\frac{\mathsf{S}(\mathsf{x}) = \mathit{null}}{\langle \mathsf{S} \mid \mathsf{x}.\mathsf{f} \rangle \longrightarrow \langle \mathsf{S} \mid \texttt{ERROR} \rangle} \text{(EXPR-FIELD-ERR)} \qquad \frac{\mathsf{S}(\mathsf{x}) = \mathit{null}}{\langle \mathsf{S} \mid \mathsf{x}.\mathsf{f} := e \rangle \longrightarrow \langle \mathsf{S} \mid \texttt{ERROR} \rangle} \text{(UPDATE-FIELD-ERR-1)}$$

The rules (EXPR-FIELD-ERR) and (UPDATE-FIELD-ERR-1) trap looking up, or updating, a field on a null-pointer.

$$\frac{\mathsf{S}(\mathsf{x}) = \mathit{null}}{\langle \mathsf{S} \mid \mathsf{x}.\mathsf{f}\texttt{--} \rangle \longrightarrow \langle \mathsf{S} \mid \texttt{ERROR} \rangle} \text{(EXPR-DREAD-FIELD-ERR)} \qquad \frac{\mathsf{S}(\mathsf{x}) = \mathit{null}}{\langle \mathsf{S} \mid \mathsf{x}.\mathit{md}\langle\_\rangle(e) \rangle \longrightarrow \langle \mathsf{S} \mid \texttt{ERROR} \rangle} \text{(EXPR-CALL-ERR-1)}$$

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

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

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

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

$$\frac{\langle S \mid e\rangle \to \langle S \mid \texttt{ERROR}\rangle}{\langle S \mid x.md\langle\_\rangle(e)\rangle \to \langle S \mid \texttt{ERROR}\rangle}\ (\textsc{expr-call-err-2})$$

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

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

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

$$\frac{\langle S \mid s\rangle \to \langle S' \mid \texttt{ERROR}\rangle \quad \vee \quad \langle S \mid s\rangle \to \langle S''\rangle \wedge \langle S'' \mid s'\rangle \to \langle S' \mid \texttt{ERROR}\rangle}{\langle S \mid s\,s'\rangle \to \langle S' \mid \texttt{ERROR}\rangle}\ (\textsc{stat-sequence-err})$$

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

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

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

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

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

$$\frac{\langle S \mid e\rangle \to \langle S \mid \texttt{ERROR}\rangle}{\langle S \mid x.f := e\rangle \to \langle S \mid \texttt{ERROR}\rangle}\ (\text{UPDATE-FIELD-ERR-2})$$

$$\frac{\langle S \mid e\rangle \to \langle S \mid \texttt{ERROR}\rangle}{\langle S \mid x := e\rangle \to \langle S \mid \texttt{ERROR}\rangle}\ (\text{STAT-UPDATE-ERR}) \qquad \frac{\langle S \mid e\rangle \to \langle S \mid \texttt{ERROR}\rangle}{\langle S \mid \texttt{t}\ x := e\rangle \to \langle S \mid \texttt{ERROR}\rangle}\ (\text{STAT-LOCAL-ERR})$$

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

Following Ernst et. al [52], we define a *finite evaluation* relation thus:

Definition 3.4.1 (Finite Evaluation)**.** An evaluation relation $\to_k$, which is a copy of the rules in the operational semantics (including the ones for error handling), where each occurrence of $\to$ in a premise is replaced by $\to_{k-1}$. For axioms and conclusions, replace $\to$ with $\to_k$ and add the following axioms:

$$\frac{}{\langle S \mid s\rangle \to_0 \langle S \mid \texttt{ERROR}\rangle}\ (\text{STAT-KILL}) \qquad \frac{}{\langle S \mid e\rangle \to_0 \langle S \mid \texttt{ERROR}\rangle}\ (\text{EXPR-KILL})$$

This means that the evaluation will return with a "kill error", if the derivation is more than n derivations deep [52]. This allows us to state a progress lemma for a finite n and thus need not account for diverging evaluations due to infinite loops, which would terminate with a kill error when the number of derivations exceeded n.

Lemma 3.4.14 (Progress)**.**

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

Following Ernst et al., a terminating expression is one for which there is an n such that the evaluation does not result in a kill error. If the application does not result in a kill

error, then it cannot have used (EXPR-KILL) or (STAT-KILL) (the kill error would have been propagated), and thus, the derivation in $\rightarrow_n$ can be translated to a derivation in $\rightarrow$.

Proof. By mutual induction over the possible shapes of s and e. The key property is the completeness of the rules, that is, capturing all possible errors during evaluation and propagating them properly. Assume $\Gamma \vdash \langle S \mid s \rangle$ or $\Gamma \vdash \langle S \mid e \rangle :: t$.

Case (EXPR-THIS) Immediate.

Case (EXPR-NUL) Immediate.

Case (LVAL-VAR) Immediate.

Case (LVAL-FIELD) By (CONFIG-EXPR) and (LVAL-FIELD), x has non-unique type t. If $S(x) = \mathit{null}$, then the configuration is reduced to $\langle S \mid \texttt{ERROR} \rangle$ by (EXPR-FIELD-ERR). Otherwise, $S(x) = \uparrow n$, and by Lemma 3.4.13 (Canonical Forms), $(S)_{n.f} = v$ so the configuration is reduced to $\langle S \mid v \rangle$.

Case (EXPR-NEW) Immediate.

Case (EXPR-DREAD-LOCAL) Immediate.

Case (EXPR-DREAD-FIELD) Similar to (EXPR-LVAL).

Case (EXPR-LOSE-UNIQUENESS) Straightforward from (EXPR-LOSE-UNIQUENESS-$*$), (EXPR-LOSE-UNIQUENESS-ERR) and the induction hypothesis.

Case (EXPR-CALL) Similar to (LVAL-FIELD), except that (EXPR-CALL-ERR-2) also handles the propagation of null-pointer errors in argument expressions and (EXPR-CALL-ERR-3) in the body of the invoked method.

Case (STAT-SKIP) Immediate.

Case (STAT-EXPR) Straightforward from (STAT-EXPR), (STAT-EXPR-ERR) and the induction hypothesis.

Case (STAT-SEQUENCE) Follows immediately from the induction hypothesis.

Case (STAT-LOCAL) Straightforward from the definition of (STAT-LOCAL-$*$), (STAT-LOCAL-ERR) and the induction hypothesis.

Case (STAT-UPDATE) Straightforward from the definition of (STAT-UPDATE-*), (STAT-UPDATE-ERR) and the induction hypothesis.

Case (STAT-SCOPED-REGION) Straightforward from (STAT-SCOPED-ERR) and the induction hypothesis.

Case (STAT-BORROW) Straightforward from (STAT-BORROW), (STAT-BORROW-ERR) and the induction hypothesis.

Case (UPDATE-FIELD) Similar to (LVAL-FIELD), except that the expression on the RHS may result in an error. In this case, the result follows from (UPDATE-FIELD-ERR-2).

□

3.5 OWNERS-AS-DOMINATORS

In this section, we formalise and prove that the *owners-as-dominators* property holds for our system. In a well-formed configuration, all external aliasing of an object comes from its owner or siblings. We model this fact using holes—in a well-formed configuration that can be factored as a stack with a hole containing an object, there are no references from objects outside the hole on the same or previous frame to the *contents* of the object in the hole. We give some auxillary definitions before presenting the result.

3.5.1 Helper Functions

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

$$\begin{aligned}
\mathsf{uses}(\mathsf{nil}) &= \emptyset \\
\mathsf{uses}(S \bullet F) &= \mathsf{uses}(S) \cup \mathsf{uses}(F) \\
\mathsf{uses}(F \oplus \alpha \mapsto n) &= \mathsf{uses}(F) \\
\mathsf{uses}(F \oplus x \mapsto v) &= \mathsf{uses}(F) \cup \mathsf{uses}(v) \\
\mathsf{uses}(F \oplus R_n[F'; H]) &= \mathsf{uses}(F) \cup \mathsf{uses}(F') \cup \mathsf{uses}(H) \\
\mathsf{uses}(F \oplus B^b_n[F'; H]) &= \mathsf{uses}(F) \cup \mathsf{uses}(F') \cup \mathsf{uses}(H)
\end{aligned}$$

$$\begin{aligned}
\mathsf{uses}(n \mapsto c^{\sigma}[V; H], H') &= \mathsf{uses}(V) \cup \mathsf{uses}(H) \cup \mathsf{uses}(H') \\
\mathsf{uses}(f \mapsto v, V) &= \mathsf{uses}(v) \cup \mathsf{uses}(V) \\
\mathsf{uses}(U_n[\uparrow m; H]) &= \{m\} \cup \mathsf{uses}(H) \\
\mathsf{uses}(\uparrow n) &= \{n\} \\
\mathsf{uses}(\mathit{null}) &= \emptyset
\end{aligned}$$

The cases $R_n[F'; H]$, $B^b_n[F'; H]$ and $U_n[\uparrow m; H]$ deal with constructs not yet introduced. Ignore them for now.

As an example, for the stack frame

$$S = n_1 \mapsto o \bullet \texttt{owner} \mapsto n_1 \oplus \texttt{this} \mapsto \uparrow n_1 \oplus R_{n_2}[\mathsf{nil}; n_3 \mapsto c^{\sigma}[f \mapsto n_4; n_4 \mapsto o']]$$

where $o = c'^{\sigma'}[\mathsf{nil}; \mathsf{nil}]$ and $o' = c''^{\sigma''}[\mathsf{nil}; \mathsf{nil}]$, $\mathsf{uses}(S) = \{n_1, n_4\}$, as the only fields and variables in S are `this` on the top-generation and the field f in object n_3.

Similarly, we define $\mathsf{defs}(S)$ to be the set of all identities of all objects, regions, borrowing blocks and uniques in S.

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

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

We can now define the structural invariants, *owners-as-dominators* and *extenal-uniqueness-as-dominating-edges* in terms of uses, defs, and #.

3.5.2 Owners-as-Dominators

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

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

We now prove owners-as-dominators for Joline using a slightly different formulation than the one found in Clarke's thesis [38]. We believe that our formulation is easier to understand as it is more clearly based on what parts of a stack or heap may reference an object.

Theorem 3.5.1 (Owners-as-Dominators). *If* $\Gamma \vdash S \bullet F\langle n \mapsto c^{\sigma}[V; H]\rangle$, *then* $\mathsf{defs}(H) \# (\mathsf{uses}(S) \cup \mathsf{uses}(F))$.

Proof. We prove this in two steps; 1) $\mathsf{defs}(H) \# \mathsf{uses}(S)$ and 2) $\mathsf{defs}(H) \# \mathsf{uses}(F)$. Note that we do not consider the case when n is nested inside a unique, as this case is covered by the stronger external-uniqueness-as-dominating-edges property, introduced in Chapter 6.

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

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

   Without loss of generality, we consider only the top-level of H. Thus, $\mathsf{owner}(t) = n$. There are two possible cases, either a) $\Gamma_1 \vdash \uparrow m :: t$, or b) $\Gamma_1\langle\Gamma_3\rangle \vdash \uparrow m :: t$ for some Γ_3. The latter covers the case when $\uparrow m$ originates from within a unique, in which case additional type information is available where the pointer is typed (see Chapter 6). In case a), $n \in \mathsf{defs}(\Gamma_1)$, by Lemma 3.4.2, which contradicts the unique names assumption as $n \in \mathsf{defs}(\Gamma_2)$. Case b)

gives rise to a similar contradition as it requires $n \in \mathsf{defs}(\Gamma_1\langle\Gamma_3\rangle)$. Thus, the pointer $\uparrow m$ cannot exist as it cannot be well-typed.

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

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

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

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

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

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

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

As an example, consider the picture in Figure 3.6. There are three possible paths to the grey object from the root: $q \to u \to r$, $q \to d \to s \to r$ and $q \to d \to f$. By the structural invariant, there may be no pointers to objects inside n from outside of

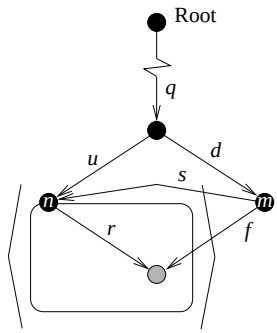

Figure 3.6: Possible paths. The $\langle\,\rangle$ denotes a hole in the store with n and its subheap as its contents. n is an object or a uniqueness wrapper.

n and thus, the path $q \rightarrow d \rightarrow f$ is invalid. As all other paths to the grey object go via n, n is a dominating node for it, meaning that the property is satisfied.

In a stack that satisfies the owners-as-dominators-property, any path to an object from the root of the object hierarchy must contain the object's owner. This means that the owner is a dominating node for all objects nested inside it [38].

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

We believe that this formalisation of the containment invariant is easier to understand than the original containment invariant from Clarke's thesis [38]: $\iota \rightarrow \iota' \Rightarrow \iota \prec^* \mathsf{owner}(\iota')$, that is, "if object ι references the object ι', then ι is inside the owner of ι'. It is not trivial to understand what this means in terms of valid aliases in a system. *Additionally, our system also deals with stack variables on previous generations*, whereas the original formulation only considered paths from the root object in the system.

3.6 Concluding Remarks

We have now presented the Joline programming language, the vehicle for our presentation. The next three chapters introduce our proposed constructs, *owner-polymorphic methods*, *scoped regions* and *external uniqueness*, and how they work together to enable alias control. In these chapters, we fill in a few of the blanks in the formalism of this chapter and give the required cases for the soundness proof.

Chapter 4

Owner-Polymorphic Methods

4.1 Ownership and Dynamic Aliasing

The encapsulation model of ownership types is only concerned with owners on paths from the root of the object graph [38]. This excludes dynamic aliases in stack-based variables or aliases in objects not reachable from the root. Relaxing constraints on dynamic aliases is sensible as they exist only for a limited scope and are destroyed when the scope is exited, which makes them less troublesome than static ones [74].

Ownership types places equal restrictions on dynamic and static aliases. Specifically, arguments to a method must have types that use only owners from the receiver's type. As an object only has a fixed set of owners, reusing a method on arguments with different owners is not even possible.

This chapter presents a way of relaxing the restrictions on dynamic aliases in a controlled way using owner-polymorphic methods. They protects the encapsulation and are expressive enough to enable a programmer to express that an argument will not be statically aliased or that a return value will not be an alias to a representation object. Thus, this solution aids preservation of separation between an object's internals and the input and output from methods.

But first, the following section shows how ownership types limits the power of polymorphism.

4.1.1 Ownership and Argument-Polymorphic Methods

Many languages place convenience methods in classes. In Java, for example, methods for converting strings into numbers are class methods in the *Integer* class. This allows for easy conversion of strings into integers and also makes conceptual sense—asking the class to create an instance of itself from another object. As a static method in Joline only has access to world, such a method would look like this:

```
static world:Integer parseInt( world:String )
{
    ... // Code omitted for brevity
}
```

Thus, it cannot be used to operate on strings that belong to an object's representation and can only produce globally accessible integers. This is far too restrictive.

Not only class methods suffer from this restriction. If a utility method that is supposed to operate on its argument is to be used with arguments belonging to different representations, we need different instances of the object defining the utility method. One needs only to consider a method such as *System.out.println( )* to realise that this situation is unacceptable—we would need multiple copies of *out*, one for each owner that need's its representation printed.

Our final example considers the implementation of structural equality tests, such as the Java *equals( )* methods. In ownership types systems, objects with different owners are guaranteed not to be aliases, but may still be structurally equivalent. This raises the interesting question of what the owner of the type of the parameter of an *equals( )* method should be. Regardless of what we choose, we are still prevented from comparing objects of different owners. Again, this is simply not acceptable.

Ownership types systems prior to our original Joline proposal [44] only allowed types to be formed using owners in scope. This is a powerful restriction that prevents static aliases to objects not part of an aggregate's representation unless the aggregate is given explicit permission to reference the object. This requires that the aggregate's type is parameterised with the necessary permission which is given for and fixed for an object's lifetime.

As the above examples show, even types of dynamic aliases suffer from only being created from owners in scope, as types must have the same owners to be assignment compatible. This rule applies to method parameters, method returns and local vari-

ables as well as fields. Thus, a class' methods can only reference objects that are safe to be statically aliased by an instance of itself. As an object has a fixed set of owners, this is very inflexible.

In conclusion, the ownership information in the types is limiting the polymorphism of variables and method parameters.

4.1.2 Borrowing and the Preservation of Separation

Borrowing, used in uniqueness proposals to avoid manual reinstatement of unique receivers and arguments [73, 92, 7, 8, 30, 32, 27, 6, 87], effectively enables a form of *preservation of separation* between an object's inners and the input and output of a method. An argument annotated with the borrowed keyword will not be statically aliased (see for example *addFrom( )* in Figure 2.6 on page 19). This annotation is useful for both clients of a method and the implementer of a class. Borrowed references have been proposed by several researchers but has yet to find its way into mainstream languages.

In existing programming languages, it is not possible to express that an argument is not supposed to be captured by the method. This can lead to rep exposure through incoming aliasing and to faulty assumptions on how a method aliases argument objects by clients of a method.

In Joline, it is immediately visible from a argument's type if belongs to the object's representation, or to some external objects, but this says nothing about whether the argument is captured or not. As any type that is a valid argument type is also valid field type, we cannot express borrowing and therefore not the separation of an argument to an object and the object's state.

Consider the example in Figure 4.1. From a client's perspective, by looking at the interface of the class *Example*, we cannot determine if *arg* will be statically aliased or not by *method( )*. From an implementer's perspective, we cannot determine anything about what client methods will assume about how *method( )* aliases its arguments. Furthermore, if we by mistake create an alias to *arg* or return an alias to *field*, the compiler will not prevent it. This is to be expected, but the opportunity for this error is likely to arise more often, as we are forced to use the object's owner parameters even for temporary variables.

In conclusion, this is inexpressive.

```
class Example< data outside owner >
{
    data:Object field = null;
    data:Object method( data:Object arg )
    {
        ... // omitted for brevity
    }
}
```

Figure 4.1: Cannot express that *arg* should be borrowed

4.1.3 Problem Analysis

Perhaps unsurprisingly, the problems discussed in Section 4.1.1 and Section 4.1.2 share a common root: the inability to temporary give an object the permission reference a set of objects for the duration of a method. If a method can be passed the necessary permissions, a single method in one object can be reused on several arguments with different owners. Furthermore, if these owners are different from the owners in the class headers, we are effectively prevented from confusing representation for arguments or vice versa, as these will have different owners and no longer be assignment compatible.

The next section presents owner-polymorphic methods that enable a flexible dynamic alias management and preservation of separation while still retaining the owners-as-dominators property for static aliases.

4.2 Owner-Polymorphic Methods

Following the pattern proposed by Clarke [38], we introduce methods in Joline that allows the creation of both stack-based and heap-based aliases with a constrained lifetime without requiring that the necessary owners are present in the receiver's type. We allow a method to take owners as parameters to enable flexible dynamic alias management—the necessary permissions to hold dynamic aliases are provided by the caller. This also allows the same method to be reused on arguments with different owners.

Owner-polymorphic methods are similar to regular, type-polymorphic methods.

They were first proposed in Clarke's dissertation [38] where they were called context-polymorphic methods and formalised in the object calculus of Abadi and Cardelli [1]. A similar proposal can also be found in Buckley's dissertation [34], but as this proposal lacks the relationships between the owner parameters, it is severly limited.

Subsequent to our original Joline proposal [44] that included owner-polymorphic methods, Boyapati included owner-polymorphic methods very similar to ours in SafeJava [21], but with no formalisation. Recently, Krishnaswami and Aldrich [80] introduced owner-polymorphic methods for a shallow ownership setting.

4.2.1 Informal Syntax and Semantics

The syntax for owner polymorphic methods is similar to the parametrically polymorphic methods in Java 5.0 [63] and slightly reminiscent of the polymorphic λ-calculus:

< borrowed **inside** *foo >* **void** *aMethod(borrowed:Object o) { ... }*

The *< borrowed* **inside** *foo >* declares a temporary *owner variable* named *borrowed* for the scope of the method. It does not introduce a new owner, but must be bound to an owner visible to the caller when the method is invoked. The clause works just like the ownership parameter clause in the class header. Owners to the right of the relation, here *foo*, are either owner variables declared in the same clause, *world*, or an owner from the header of the declaring class. Just as for class headers, the ordering relation can be both *inside* and *outside* (corresponding to $\prec^*$ and $\succ^*$ in the formalism).

Invoking *aMethod( )* requires passing the owner of the first reference argument as an owner argument:

x.aMethod< b >(**new** *b:Object( ));*

We allow owner arguments to be used as regular owners. They may be used to form types, instantiate new objects and may be passed as owner arguments to other owner-polymorphic methods. Figure 4.2 shows the use of owner parameters in a factory-like method where the caller specifies the "target owner" of the result. As the target owner is inside the owner *mydata*, we can populate the list with data from an existing list. This is a useful pattern that we expect will come in handy.

```
<res inside mydata> res:List< mydata > almostFactory( )
{
    res:List< mydata > temp = new res:List<mydata>( );
    // mylist has type this:List< mydata >
    temp.populateWithList( mylist );
    return temp;
}
```

Figure 4.2: Owner-Polymorphic Factory Method that creates a list instance and populates it with elements from an existing list

4.2.2 Solutions to the Problems in Sections 4.1.1 and 4.1.2

As was pointed out earlier in this chapter, class methods suffer by only having access to the *world* owner. Thus, class methods can only reference globally accessible objects, which is very restrictive as we showed in Section 4.1.1.

Luckily, this is easily overcome by our owner-polymorphic methods by passing in the owner of the string to be converted:

```
<temp inside world> static temp:Integer parseInt( temp:String )
{
    ... // Code omitted for brevity
}
```

Here, *temp* is an owner parameter that can be bound to any owner inside *world*. When the method is called, the caller supplies the owner arguments as well as the reference arguments and the type system makes sure that the owner arguments respect the nesting requirements of the method header. In this case, checking is trivial as all owners are inside *world*. Thus, our conversion method can be used on all strings in a system, which is exactly what we want.

In a similar fashion, owner-polymorphic methods allow an *equals( )* method to reference an object with different owners which is essential to its implementation. The code below shows an example of an equals method that compares two *Person* instances.

```
<borrowed inside world> world:Boolean equals( borrowed:Person person )
{
    // Cannot alias person, but is allowed to use it
    borrowed:String name = person.getName();
    borrowed:String birthDate = person.getBirthDate();
    return this.name.equals< borrowed >( name ) and
                this.birthDate.equals< borrowed >( birthDate );
}

this:Person p1 = new this:Person( "Barbarella", "1937-12-21" );
world:Person p2 = new world:Person( "Jane Fonda", "1937-12-21" );
p2.equals< this >( p1 );
```

The last line gives a world-owned object temporary permission to reference rep of the instance enclosing the last three lines. For this example to work, the types of *name* and *birthDate* in *Person* have owner *owner*.

Interestingly, *person* cannot be statically aliased in the enclosing object. As the owner *borrowed* is only defined for the scope of the method, it cannot be used in a type of a field in the object. Thus, the owner-polymorphic method allows the programmer to express that an argument object cannot be captured by the object, or that an argument object may not store capture references to the receiver's state. Thus, we achieve a preservation of separation between argument objects and the receiver's state.

In our proposal, owner parameters are regular owners. Thus, we can even implement factory methods in classes that create objects of a desired owner:

```
<target inside world> static target:Object factory( )
{
    return new target:Object();
}
```

In previous systems with deep ownership, factory class methods were almost unusable as they were constrained to return only global objects.

We now proceed by describing how owner-polymorphic methods can be used to simulate a form of borrowing, and how they can be used with proxies and subtyping to create quite useful static aliases in the absence of the correct owner in the type of the receiver.

4.2.3 Borrowing and Preservation of Separation

An Owner-polymorphic method enables the caller to grant the receiver permission to reference any object outside the receiver.

Depending on the owner parameter's relation to the owner *this* in the receiver, the permission can be temporary: the value of the reference argument may not flow into a field on the heap whose life-time exceeds that of the stack-frame. This is similar to how borrowed references work—a parameter annotated with a *borrowed* keyword may not be stored in a field or passed as a non-borrowed argument to another method, effectively confining the reference to stack-local variables [73, 29].

The expected behaviour of many methods is that they will not statically alias their arguments. For example, a map method that applies something to every object in a list should not keep a reference to the list nor should a method (such as the *max(Int a, Int b)* method) that returns one of its two arguments depending on some relational property. Again, if argument objects cannot be statically aliased we can unconditionally guarantee preservation of separation: argument objects cannot be mixed up with representation, as the types are incompatible.

We define *borrowed owners*, *borrowed types* and *borrowed references* thus:

Definition 4.2.1 (Borrowed Owner)**.** An owner parameter to the current method that cannot be statically determined to be *outside* the current *this*.

Clearly, this does not cover all possible owner parameters. Owner-parameters known to be outside *this* are safe to statically alias under deep ownership. Section 4.2.4 shows how subtyping can be used to create a static path from the receiver to such an object. Thus, we do not count these owners, or types or references that use them, as borrowed.

Definition 4.2.2 (Borrowed Type)**.** A type that uses a borrowed owner for one of its owner parameters.

Definition 4.2.3 (Borrowed Reference)**.** A reference that has a borrowed type.

As borrowed owners are in scope only for the duration of the method, the receiver cannot declare fields to be of borrowed types. Thus, the borrowed references cannot be statically aliased in the receiver. As a borrowed owner is also not outside any other owner statically known to the class, we cannot even create a statically aliasable proxy

object that stores a static alias to the borrowed reference. This is discussed in more detail in Section 4.2.4.

Below is the declaration of an *Person* class with its implementation details hidden. The owner parameter *some* is a borrowed owner as is it (statically) not outside any owner accessible to the *Person* class. Thus, *inc* is a borrowed reference and subsequently, without any knowledge of the implementation of *incSalary* or the rest of the class, we know the following: the method will not statically alias *inc*, nor will it return a representation object, even if the salary is represented by an *Int* internally.

```
class Person
{
    <some inside world> some:Int incSalary( some:Int inc )
    {
        ... // implementation hidden
    }
}
```

This is similar to most previous borrowing proposals [73, 92, 7, 8, 30, 32, 27, 6, 87], but actually even more flexible. There is nothing to prevent *incSalary( )* from creating static aliases to *inc* in itself, or creating an object with *some* as its owner and statically alias *inc* in that object. Allowing this makes sense as such objects will also be borrowed and unless returned from the method or stored in *inc*, all aliases to them will be destroyed (buried in Boyland's terminology [30]) when the method exits.

Note that we achieve this without introducing an additional kind of borrowed reference or owner, which is required in all other borrowing systems. We will revisit this topic in Section 4.2.5.

Declaring methods like in the example above makes sense, even if you expect them to be used with only owners that are in the receiver's type. For one, it makes the class less sensible to changes in the ownership structure, but it also prevents the method from mixing the arguments with the receiver's representation, even if the owners bound to the parameters are the receiver's own at run-time.

The aliasing properties of borrowed references can be stated thus:

Aliasing Property 4.2.1 (Borrowed Reference)**.** A borrowed reference cannot be confused for representation, or vice versa, as the types are incompatible. Thus, is it impossible to make a borrowed object part of the representation or violate rep exposure by creating aliases to parts of the representation in borrowed objects.

Where desirable, borrowed references can be used to preserve separation of an object's inners from the input and output from its methods.

Some of the above properties apply to non-borrowed references with owners from owner parameters to a method. The types will be incompatible, preventing argument objects to be mistakenly treated as representation. However, as the next section will show, static aliases can still be created.

Aliasing Property 4.2.2 (Borrowed References)**.** A method can only store static aliases to a borrowed object in objects obtained from the borrowed object or in objects created by the method using the borrowed owners.

Clearly, objects obtained from borrowed objects are also borrowed and suffer the same static alias restrictions. Thus, no static path can be created from the receiver object (or its rep) to any borrowed object.

While not as strong as for unique references, we feel that these are useful properties. Also, for the last property, the cases when static aliases may be created are exceptions rather than rule. Generally, a method only reads arguments and is concerned with modifying its receiver. Passing objects in to be updated and methods that modify the structure of their argument objects are generally considered "bad smells" [56]. Such methods should probably be moved into the argument object, which fits better with object-orientation [56].

We now explain why not all ownership parameters are borrowed owners and how this can be used to overcome lack of explicit right to reference using the proxy design pattern [57].

4.2.4 The "Hide Owner" Pattern

The encapsulation model of ownership types allows outgoing references to enclosing objects. The nesting of ownership information is crucial to maintaining this invariant. This is a powerful scheme. However, it can also be limiting, as the necessary ownership parameters to reference an outside object, again, perfectly legal, might not be present in the object. The following method declaration allows the manipulation of enclosing objects (at some level of indirection) using the owner parameter *some* passed to the method by the caller.

<some **outside** *this>* **void** *doSomethingWith(some:Stream stdout) { ... }*

As we know that *some* is outside *this* we can easily instantiate a wrapper object of the following type:

this:StreamWrapper< some > wrapper = **new** *this:StreamWrapper< some >( );*
wrapper.setObject(stdout);

The instantiated object is part of the representation and will exist on the heap with the permission to statically alias *stdout*. As *some* is only defined for the scope of the method, we cannot have fields in the current object using *some* in their type. We can however use subsumption to hide *some* making it possible to store a reference to the wrapper in the current object. Assuming the following class header of the *StreamWrapper* class:

class *StreamWrapper< w* **outside** *this >* **extends** *IOWrapper { ... }*

we can cast *wrapper* to an *IOWrapper* hiding the use of *some*.

this:IOWrapper field = (this:IOWrapper) wrapper;

Now, a static path from the receiver to the argument object has been created and the wrapper can be used by the object to write things to the underlying stream via the wrapper without knowledge of the actual owner. This is a powerful mechanism that makes ownership types a lot more flexible as it is no longer necessary to to statically know all owners of outgoing aliases. We particularly expect this pattern to come in handy during maintainence when changes to ownership could otherwise propagate through the system forcing many changes. On the negative side, the difference between a borrowed reference and a non-borrowed reference becomes subtle, which might possibly be overcome by introducing a "capturable keyword" to mean that an owner is outside this.

Note that owner parameters declared inside *world* can be bound to any possible owners in a system. As such an owner is not statically know to be outside anything, it is always a borrowed owner. This means that we can never be forced into a situation where the system requires us to use non-borrowed owners, except when we want the ability to statically alias an argument. If we wanted to preclude non-borrowed owners altogether, for example for the above-metioned reasons, it would be easy—just prevent owners parameters that are outside *this*.

The the use of non-borrowed owner parameters and the hide-owner pattern is an extension of previously published results [44, 45].

4.2.5 Discussion

Owner-polymorphic methods enable flexible dynamic alias management and preservation of separation. As the aliases are dynamic and cannot flow to the heap except in case just described, the owners-as-dominators property still holds.

We now compare the borrowing enabled by owner-polymorphic methods with the "traditional" form of borrowing [73, 92, 7, 8, 30, 32, 27, 6, 87] that we covered in Chapter 2. The following code snippet shows an example of uniqueness and traditional borrowing:

```
class Example
{
    Object field = null;

    Object method( borrowed Object arg ) anonymous
    {
        field = arg; // not okay          (*)
        List temp = new List();
        temp.add( arg ) // not okay       (**)
        temp.destroy( );
        return arg; // not okay           (***)
    }

    void test( ) anonymous
    {
        unique Object o = new Object();
        Object temp = method( o ); // borrows o
    }
}
```

The example above shows that a borrowed reference cannot be captured in a field (lines (*) and (**)). It would be okay to pass *arg* as a parameter to *add( )* on line (**), if the argument to *add( )* was borrowed. As *temp* is a list, this is naturally not the case, as the list would not be permitted to store the argument object. On line (***), we are not allowed to return a borrowed object. The reason why most proposals do not allow this, is because they would then lose track of the borrowed pointer. If it was allowed, the method *test* would create two pointers to a unique object, one in *o* and one in *temp*, after the invocation of *method( )*.

As is visible from the code in *method( )*, the list stored in *temp* is stack-local. When the method returns, the list is no longer reachable by the program. Thus, storing *arg* in the list would not violate borrowing. Existant borrowing proposals cannot express this (even if *temp* is borrowed) and subsequently cannot allow borrowed pointers from appearing in any field on the heap. As our example shows, this is *overly restrictive*.

Our proposal is an improvement on the following points:

1. We enable borrowing *without* introducing an additional kind of pointer that must be treated differently than other pointer kinds. Borrowed references are simply regular references, and the owners-as-dominators property makes sure that they do not escape. Thus, our proposal has no need for **borrowed** or **anonymous** annotations as in the previous example.

   The above is a result of owner-polymorphic methods being an *orthogonal addition* to our system. (This is discussed further in Section 7.2.) This is also the case in DeLine and Fähndrich's Vault system [47, 53]. The gains are less complexity and increased flexibility.

2. The type system in Joline is strong enough to allow borrowed references to be stored on the heap, still guaranteeing that these references will not be captured by static aliases in the receiver. This avoids the overly restrictive situation on line (**) in the previous example. In addition, we can allow borrowed references to be returned as the owner keeps track of to what representation the returned object belongs.

   Upcoming chapters will make borrowing and return of borrowed objects even more powerful, and allow return of borrowed pointers, even in the presence of uniqueness.

3. Owner parameters outside **this** allows argument objects to be statically aliased by the receiver, even if the receiver does not have the proper owners in its type. The owners-as-dominators property is still preserved. While not strictly borrowing in the traditional sense, we can use owner-polymorphic methods in a way that allows the situation on line (*) in the example.

Subsequent to our proposal, Krishnaswami and Aldrich [80] proposed owner-polymorphic methods as an orthogonal addition, similar to ours but without the ownership nesting as it is done in a shallow ownership setting. Having have several kinds

owners, their methods are more flexible than ours as they can restrict owner parameters from being used to instantiate objects. Adding a "mode on owners" to achieve the same flexibility in our system would be straightforward. However, we believe that it would be less useful in a system offering deep ownership, as the containment invariant of deep ownership prevents the created objects from escaping except as parts of the borrowed aggregate or back to the caller.

4.3 FORMALISING OWNER-POLYMORPHIC METHODS

Below we extend the Joline formalisation with owner-polymorphic methods.

4.3.1 Static Semantics

The extended syntax for method declaration and method call is shown below. To emphasise differences between new and previous versions, we highlight extensions thus.

The syntax for methods and method calls are replaced for the following:

$$\begin{array}{lll} \mathit{meth} & ::= & \langle \alpha_i\ R_i\ p_{i\in 1..m} \rangle\ t\ \mathit{md}(t_i\ x_{i\in 1..n})\ \{\ s\ \texttt{return}\ e\ \} \qquad \textit{Method} \\ e & ::= & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{Expression} \\ & & e.\mathit{md}\ \langle p_{j\in 1..m} \rangle\ (e_{i\in 1..n}) \qquad\qquad\qquad\qquad\qquad\quad \textit{method call} \end{array}$$

Below are the extended versions of (METHOD) and (EXPR-CALL) that includes owner parameters.

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

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

The (EXPR-CALL) is extended to handle the owner arguments passed to the method.

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

In this version, the owner arguments of the target type and the owner arguments supplied to the method form *two* substitutions to transform the method's argument and return types into types in terms of the owners in scope. The additional substitution, σ', translates the names of owner parameters used internally in the method to the actual owners at the call-site. The helper function $\mathcal{M}_c$ for looking up parameters and method bodies is extended in a straightforward fashion to also include owner parameter lists.

As pointed out by Clarke [38], owner argument passing could be replaced by an inference mechanism that infers the σ' binding of owners to the parameters of the method, just as in GJ [33]. That would introduce unnecessary complexity for our purposes here so we chose this way out for simplicity. A possible inference mechanism would look at the owner parameters of the method header and where these are used in the parameter types. It would then match the types of the arguments with the types of the parameters to derive the mapping.

4.3.2 Dynamic Semantics

The new rule for method call is presented here. The big difference is the passing in of the owner parameters which are then stored on the stack.

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

As can be seen in the above rule, the extensions are simple and straightforward. Again,

to simplify the formal account, we limit the number of owner and value arguments to one. The helper function $\mathcal{D}_t(md)$ is extended with owner parameters in a straightforward fashion.

The subject reduction proof for (EXPR-CALL) is as follows:

Proof. Assume $\Gamma \vdash \langle S \mid x.md\langle p\rangle(e)\rangle :: t'$, and $\langle S \mid x.md\langle p\rangle(e)\rangle \rightarrow \langle S' \mid v\rangle$.

1. By (CONFIG-EXPR),
   (a) $\Gamma \vdash S$, and
   (b) $\Gamma \vdash x.md\langle p\rangle(e) :: t'$.
2. By 1.b) and (EXPR-CALL),
   (a) $\Gamma \vdash x :: m{:}c_1\langle\sigma_1\rangle$,
   (b) $\mathcal{M}_{c_1}(md) = (\alpha \mathsf{R}\, q, t_2 \rightarrow t_3, y)$,
   (c) $\texttt{this} \in \mathsf{owners}(\mathcal{M}_{c_1}(md))$ implies $x \equiv \texttt{this}$,
   (d) $\Gamma \vdash p \mathsf{R}\, \sigma_1^m(q)$ and
   (e) $\Gamma \vdash e :: t_4$ where
   (f) $t' = \sigma_2(t_3)$,
   (g) $t_4 = \sigma_2(t_2)$ and
   (h) $\sigma_2 = \sigma_1^m \cup \{\alpha \mapsto p\}$.
3. By 1.a), 2.e) and (CONFIG-EXPR), $\vdash \langle S \mid e\rangle :: t_4 \gg \Gamma$.
4. By 3.) and the induction hypothesis, if $\langle S \mid e\rangle \rightarrow \langle S_1 \mid v\rangle$, there exists a Γ_1 s.t.,
   (a) $\Gamma \rightsquigarrow \Gamma_1$ and
   (b) $\Gamma_1 \vdash \langle S_1 \mid v\rangle :: t_4$.
5. By 4.b) and (CONFIG-VAL),
   (a) $\Gamma_1 \vdash S_1$ and
   (b) $\Gamma_1 \vdash v :: t_4$.
6. By 2.a), 4.a) and Lemma 3.4.1 (Extension), $\Gamma_1 \vdash x :: m{:}c_1\langle\sigma_1\rangle$.
7. By 2.d), 4.a) and Lemma 3.4.1 (Extension), $\Gamma_1 \vdash p \mathsf{R}\, \sigma_1^m(q)$.
8. By 5.a), 6.) and Lemma 3.4.4 (Variable Lookup), $\Gamma_1 \vdash \uparrow n :: m{:}c_1\langle\sigma_1\rangle$.
9. By 8.), (VAL-PTR) and (VAL-SUBSUMPTION),

   (a) $\Gamma_1(n) = m{:}c\langle\sigma\rangle$ and
   (b) $\Gamma_1 \vdash m{:}c\langle\sigma\rangle \leqslant m{:}c_1\langle\sigma_1\rangle$.

10. By 7.) and Lemma 3.4.2 (Well-Formed Construction), $\Gamma_1 \vdash p$.

11. By 9.a) and def. of dispatch, $\Gamma \vdash m{:}c\langle\sigma\rangle \leqslant m{:}c_2\langle\sigma_2\rangle$. ($m{:}c_2\langle\sigma_2\rangle$ is the type of the receiver of *md*)

12. By 2.b), (CLASS) and Lemma 4.3.1,

   (a) $\Gamma_1 \bullet \Gamma'' \vdash \mathit{meth}$ where
   (b) $\Gamma'' = \sigma_2{}^m_n \oplus \texttt{this} :: t_7$, and
   (c) $\mathit{meth} = \langle\alpha \mathsf{R}\, q_1\rangle\ t_5\ \mathit{md}(t_6\ x)\{\ s\texttt{;return}\ e\ \}$ where
   (d) $\sigma_2{}^m(t_7) = m{:}c_2\langle\sigma_2\rangle$.

13. By 2.b,f-h) def. of method look-up,

   (a) $\sigma_2^m(q_1) = \sigma^p(q)$ and
   (b) $(\sigma_2^m \cup \{\alpha \mapsto p\})(t_6) = t_4$,
   (c) $(\sigma_2^m \cup \{\alpha \mapsto p\})(t_5) = t'$.

14. By 12.a,b,d), Lemma 3.4.2 (Well-Formed Construction), Lemma 3.4.3 (Generation Removal), (OWNERS) and (VARIABLES),

   (a) $\Gamma_1 \bullet \Gamma'' \vdash F \gg \Gamma''$, where
   (b) $F = \sigma_2{}^m_n \oplus \texttt{this} \mapsto \uparrow n$.

15. By 10.), 14.a), Lemma 3.4.1 (Extension), (STORE-TYPE-OWNER) and (OWNERS), $\Gamma_1 \bullet \Gamma'' \oplus \alpha \mapsto p \vdash F \oplus \alpha \mapsto p \oplus \gg \Gamma'' \oplus \alpha \mapsto p$.

16. By 5.b), 13.b), 15.) Lemmas 3.4.1 and 3.4.2 (Extension and Well-formed Construction) and Lemma 3.4.3 (Generation Removal), $\Gamma_1 \bullet \Gamma'' \oplus \alpha \mapsto p \vdash v :: t_6$. (Clearly, $\mathsf{owners}(t_6) \subseteq \sigma_2^m \cup \{\alpha \mapsto p\}$.)

17. By 16.), (STORE-TYPE-VARIABLE) and (VARIABLES), $\Gamma_1 \bullet \Gamma'' \oplus \alpha \mapsto p \oplus x :: t_6 \vdash F \oplus \alpha \mapsto p \oplus x \mapsto v \gg \Gamma'' \oplus \alpha \mapsto p \oplus x :: t_6$.

18. By 5.a), 14.), 17.) (STACK-GEN),

   (a) $\Gamma_1 \bullet \Gamma' \vdash S_1 \bullet F' \gg \Gamma'$ where
   (b) $\Gamma' = \Gamma'' \oplus \alpha \mapsto p \oplus x :: t_6$ and
   (c) $F' = F \oplus \alpha \mapsto p \oplus x \mapsto v$.

19. By 2.d), 18.b), Lemma 3.4.1 (Extension) and (OWNER-EQUAL), (IN-OWNER-EQUAL), $\Gamma_1 \bullet \Gamma' \vdash \alpha \mathsf{R}\, q_1$.

20. By 12.b,c), 18.b), 19.) and (METHOD),
    (a) $\Gamma_1 \bullet \Gamma' \vdash s; \Gamma \bullet \Gamma' \oplus \Gamma'''$ and
    (b) $\Gamma''' \bullet \Gamma' \oplus \Gamma_1 \vdash e :: t_5$.
21. By 18.a), 20.a) and (CONFIG-STAT), $\Gamma_1 \bullet \Gamma' \oplus \Gamma''' \vdash \langle S_1 \bullet F' \mid s \rangle$.
22. By 21.) and induction hypothesis, if $\langle S_1 \bullet F' \mid s \rangle \rightarrow \langle S_2 \rangle$, then there exists Γ_2 s.t.
    (a) $\Gamma_1 \bullet \Gamma' \oplus \Gamma''' \rightsquigarrow \Gamma_2$ and
    (b) $\Gamma_2 \vdash \langle S_2 \rangle$.
23. By 22.b) and (CONFIG), $\Gamma_2 \vdash S_2$.
24. By 20b.), 22.a) and Lemma 3.4.1 (Extension), $\Gamma_2 \vdash e :: t_5$.
25. By 23.), 24.) and (CONFIG-EXPR), $\Gamma_2 \vdash \langle S_2 \mid e \rangle :: t_5$.
26. By 25.) and induction hypothesis, if $\langle S_2 \mid e \rangle \rightarrow \langle S_3 \mid v' \rangle$, then there exists Γ_3 s.t.
    (a) $\Gamma_2 \rightsquigarrow \Gamma_3$ and
    (b) $\Gamma_3 \vdash \langle S_3 \mid v' \rangle :: t_3$.
27. By 26.a) and (CONFIG-VAL),
    (a) $\Gamma_3 \vdash S_3$ and
    (b) $\Gamma_3 \vdash v' :: t_5$.
28. By 27.a) and (STACK-GEN),
    (a) $\Gamma_4 \bullet \vdash S_4$,
    (b) $\Gamma_3 \vdash F'' \gg \Gamma_5$ where
    (c) $\Gamma_3 = \Gamma_4 \bullet \Gamma_5$ and
    (d) $S_3 = S_4 \bullet F''$
29. By 12.b), 18.b), 22.a.), 26.a), 28.c) and def. of $\rightsquigarrow$,
    (a) $\Gamma_1 \rightsquigarrow \Gamma_4$ and
    (b) $\Gamma'' \oplus \alpha \mapsto p \rightsquigarrow \Gamma_5$.
30. By 27.b), 29.b) and Lemma 3.4.3 (Generation Removal),
    $\Gamma_4 \vdash v' :: (\sigma_2{}_n^p \cup \{\alpha \mapsto p\})(t_5)$. (Clearly $\Gamma_5(t_5) = (\sigma_2{}_n^m \cup \{\alpha \mapsto p\})(t_5)$ as $\mathsf{owners}(t_5) \subseteq \mathit{rng}(\sigma_2{}_n^m \cup \{\alpha \mapsto p\})$.)
31. By 13.c) and 30.), $\Gamma_4 \vdash v' :: t'$.

32. By 28.a), 31.) and (CONFIG-VAL), $\Gamma_4 \vdash \langle S_4 \mid v' \rangle :: t'$.

□

Lemma 4.3.1. *If* $\Gamma \vdash \uparrow n :: p : c\langle\sigma\rangle$ *and* `class` $c \cdots \{\, fd_{1..l}\ meth_{1..m} \,\} \in P$, *then* $\Gamma' \vdash fd_{1..l}$ *and* $\Gamma' \vdash meth_{1..m}$ *where* $\Gamma' = \Gamma \bullet \sigma^p_n \oplus$ `this` $:: t$ *s.t.* $\sigma(t) = p{:}c\langle\sigma\rangle$.

Proof.

1. By (CLASS),
   (a) $E \vdash fd_{1..l}$ and
   (b) $E \vdash meth_{1..m}$ where
   (c) $E =$ `owner` $\prec^*$ `world`, $\mathcal{P}_c$, `this` $\prec^*$ `owner`, `this` $:: t$ and
   (d) $t =$ `owner`$:c\langle\mathcal{P}_c\rangle$.
2. By (VAL-POINTER), $\Gamma \vdash \uparrow n :: p{:}c\langle\sigma\rangle$.
3. By 2.) and (STORE-TYPE-GENERATION),
   (a) $\Gamma \bullet \sigma^p_n \oplus$ `this` $:: t \vdash \diamond$ where
   (b) $\sigma^p(t) = p{:}c\langle\sigma\rangle$.
4. By 3.a,b) and (TYPE), $\Gamma \bullet \sigma^p_n \oplus$ `this` $:: t \vdash \mathcal{P}_c$.
5. By Lemma 3.4.2 (Well-formed construction), $\Gamma \vdash n$.
6. By 5.), (IN-WORLD) and (TYPE-EQUAL), $\Gamma \bullet \sigma^p_n \oplus$ `this` $:: t \vdash$ `owner` $\prec^*$ `world`.
7. By (VAL-POINTER), $\Gamma(n) = p{:}c\langle\sigma\rangle$. By def. of type look-up, $\Gamma = \Gamma'\langle n :: p\langle\sigma\rangle[\_]\rangle_p$. Thus, by (IN-OWNER), $\Gamma \vdash n \prec^* p$.
8. By 3.a) and 7.) and (TYPE-EQUAL), $\Gamma \bullet \sigma^p_n \oplus$ `this` $:: t \vdash$ `this` $\prec^*$ `owner`.
9. By 1.a,b), 4.), 6.), and 8.), clearly $\Gamma \bullet \sigma^p_n \oplus$ `this` $:: t$ satisfies the orderings and typings in E and therefore,
   (a) $\Gamma \bullet \sigma^p_n \oplus$ `this` $:: t \vdash fd_{1..l}$ and
   (b) $\Gamma \bullet \sigma^p_n \oplus$ `this` $:: t \vdash meth_{1..m}$.

□

4.4 Concluding Remarks

In this chapter, we introduced owner-polymorphic methods, and showed how they could be used to preserve the separation of argument objects and representation using borrowed references. We also presented a pattern for adding references to external objects using implicit permissions and proxies, and discussed related proposals. We concluded by extending the formalisation of Joline with support for owner-polymorphic methods.

The next chapter presents another extension to the Joline language, *scoped regions*, that can be used for confining heap-allocated objects to the stack and for additional preservation of separation.

Chapter 5

Scoped Regions

5.1 Stack-Based Confinement

The owner-polymorphic methods proposed in the previous chapter facilitates preservation of separation—preventing arguments to methods from being mistaken for representation and representation objects mistakenly returned from a method. In this chapter, we introduce scoped regions, which allow methods to create stack-local objects with access to any object visible at the point of creation. Similar to the owner-polymorphic methods, the scoped region introduces a temporary owner, that guarantees that stack-locals will not leak or even be visible when the scoped region exits.

It is not uncommon for a method to create stack-local, temporary objects whose life-times are tied to that of the method body. A frequently occuring example is intermediate values of a computation. As another example, a library might require data to be passed in in the form of a specific structure or object that can be discarded once the result is obtained. Generally, we would want to *preserve separation of such objects from representation* and also *prevent them from being exposed or returned from the method*—if a temporary object escapes, even if it is not part of the rep, it effectively breaks abstraction as it might expose details of the receiver's implementation.

As a consequence of these two points, we could then, and would like to be able to, *automatically garbage collect them when the method exists without creating any dangling pointers.* However, as we shall see, our present system is not strong enough to express this type of confinement, not even in the presence of owner-polymorphic methods.

The following section shows how our present system fails to express the points above including a brief analysis of the problem. We then move on to describe some related work in this area, before showing our proposal of *scoped regions*.

5.1.1 Stack-Based Confinement in Joline

The unit of confinement in our current Joline system is not fine-grained enough to confine an object to a specific stack-frame, making it a stack-local object. Sure enough, owner parameters could be used to instantiate borrowed objects inside the method, but these can still escape, either by the addition of a back-pointer to an argument object, or be returned. Thus, locally created objects with borrowed owners are not necessarily stack-local.

Consider the method below that serialises the contents of an object to a stream. The object connects an object-writing stream to the borrowed byte stream. In the programmer's mind, the *ObjectStream* instance is supposed to be stack-local and could thus be destroyed when the method exists. Our current Joline system lacks the means to express this as the *ObjectStream* must be in *io*'s representation to be allowed to refer to the *ByteStream*.

```
< io inside world > void serialiseContents( io:ByteStream bs )
{
    // create a temporary stream that write entire objects linked to bs
    io:ObjectStream< io > w = new io:ObjectStream< io >( bs );
    ... // actual writing omitted
    // GC.free( w ); // garbage collect the object stream
}
```

Sadly, in the eyes of the compiler, there are several ways the *ObjectStream* instance can escape the method. The constructor of *ObjectStream* might create an alias to *this* in *bs* or an exception may be thrown that keeps a reference to the object stream. Thus, the method in the last line, *GC.free(w)*, if it existed, could create dangling pointers. In any case, determining that no aliases to the object in *w* has escaped is tricky at best, and impossible at worst. So-called *Escape Analysis* deals with this problem, see Blanchet [17] and Gay and Steensgaard [59].

In our example above, the owner of *w* must be *io* in order to allow the object stream to statically reference the byte stream. The reason for this is that the only owner

statically known to be inside *io* is *io* itself (by reflexivity of the inside relation). If *w* was owned by *this*, the object stream could not be returned or aliased in *io*. However, this would force us to declare *io* as outside *this* in the method header, losing the borrowing. Furthermore, the temporary stream could still be aliased by representation objects or the current receiver, losing its intended temporality.

In conclusion, the unit of confinement for deep ownership types is not fine-grained enough to express stack-based confinement of an object. This is not surprising, as the unit of confinement is objects, which are heap-based.

5.1.2 Value Objects

Both C++ and Eiffel support value objects that can be used to implement stack-local objects. Value objects are located physically inside the enclosing object or stack frame. This might improve locality as a value objects and its subobjects would most likely reside on the same memory page and causes value objects to destroyed automatically when the enclosing object or frame is destroyed. In C++, this can result in dangling pointers as the language allows pointers to value objects to allow in-place updates and avoid costly copying, but it also allows destructors to run properly.

Eiffel addresses this problem with *expanded types* [91], which must be copied. Eiffel thus avoids C++'s problem of dangling pointers but increases the cost of dealing with expanded types.

As value objects are included in their enclosing objects, the size of a value object must be statically known as the corresponding number of bytes must be allocated to make room in the enclosing object of frame. This makes value objects sensitive to subtyping. In C++, polymorphic treatment of a value object can result in *slicing*. For example, let T be a type of a value object of some size n and T' a subtype of T of size $n + m$ for some additional fields declared in T'. If a value object of type T' is stored in a field of type T, the field will not be large enough to hold the additional variables, effectively cutting off a slice of the object.

Eiffel avoids this problem by preventing expanded types from participating in polymorphism. Thus, Eiffel does not suffer from slicing, but the flexibility of polymorphism, crucial to object-oriented systems, is lost. For an account of additional problems with expanded types in Eiffel, and ways to overcome them, see work by Kent and Howse [76].

In conclusion, the value objects of Eiffel and C++ are useful and can be used to simulate stack-based confinement. On the downside, they are quite problematic. They can either cause dangling pointers or lose polymorphism entirely. Thus, in our mind, the value object solutions are insatisfactory.

Stack-based confinement aids garbage collection as confining a temporary object to a scope such as a method or a block makes it possible to statically determine the sites where the object can be removed [93]. For similar reasons, it would also *help reasoning about aliasing*, as the life-time time of an alias to a temporary object is known. Most importantly, as the owner of the scoped objects is disjoint from all other owners in scope, the temporary object is guaranteed not to be statically aliased, mistaken for representation or escape the enclosing method. It is guaranteed to be separated from argument objects and representation. We now move on to describe *scoped regions*, our implementation of stack confinement in Joline.

5.2 Scoped Regions

This section describes the scoped region construct, its aliasing guarantees and its formalisation in the context of the Joline language. It also discusses related work.

A scoped region is a block that introduces a new, *scoped owner* for its scope, that is potentially inside all other visible owners. One can think of the owner introduced by the scoped region as corresponding to the block and of the block as a closure. The nesting of the owner inside all other owners in scope allows objects owned by the scoped region to access any visible object. This includes representation objects as and borrowed objects. When the block exits, the owner can no longer be named, which means that all types that use the owner, *scoped types*, become invalidated and subsequently, so are all references to objects of scoped type. Thus, all aliases to representation objects or borrowed objects will be destroyed and subsequent statements can disregard them, aiding reasoning in the presence of aliasing. The syntax of a scoped region is thus:

(scopedOwner) `{ /* statments */ }`

We define scoped owners, types and references thus:

Definition 5.2.1 (Scoped Owner)**.** An owner defined by the scoped region construct.

Definition 5.2.2 (Scoped Type)**.** A type that uses a scoped owner for one of its owner parameters.

An interesting observation is that, as the only owners known to be inside a scoped owner is the scoped owner itself and possible scoped owners of nested regions, if a type uses a scoped owner in a non-owner position, its owner must also be scoped.

Definition 5.2.3 (Scoped Reference)**.** A reference pointing to an object of scoped type. Such a reference and its referenced object are effectively confined to the range of the scoped owner.

The code below shows an example of using a scoped regions to implement the code example on Page 108. A nice feature of our proposal is that by virtue of deep ownership, objects owned by scoped owners are automatically prevented from escaping the region as all owners outside it lack the necessary permissions to reference the scoped objects. Thus, an object need not be implemented in any special way to use the confinment offered by a scoped region. Subsequently, the *ObjectStream* from the previous example in unchanged.

```
< io inside world > void serialiseContents( io:ByteStream bs )
{
    // create a temporary stream that write entire objects linked to bs
    (temp)
    {
        // ok as temp is inside io
        temp:ObjectStream< io > w = new temp:ObjectStream< io >( bs );
        ... // actual writing omitted
    }
    // the object stream object can now be garbage collected
}
```

Wrapping the creation of the object stream in a scoped region effectively confines it to the stack, or more specifically, to the block of the scoped region. Subsequently, the writing of the objects to the stream must also be inside the region. Once the region is exited, *w*, and all aliases to it goes out of scope. As *temp* is inside all owners in scope, including *this* and *io*, there can be no aliases from *bs* to *ws*, nor from any representation objects in the current object. Thus, the object stream is unreachable by the running program, and can thus be safely deleted without leaving any dangling pointers.

As a scoped owner is fresh, no preexisting objects have permission to reference objects owned by it. Thus, exporting scoped references to objects outside the region requires the use of owner-polymorphic methods. As preexisting objects are by definition outside the region, the owner parameters of the owner-polymorphic methods must be borrowed owners. Thus, aliases to the borrowed objects created by such a method are destroyed when the method exits, unless the aliases are stored inside the borrowed object or returned to the scoped region as the method's return value. Thus, when the block exits, the last references to the scoped objects goes out of scope. Thus, no external objects can be dependent on the scoped objects and they can be safely deleted without risk of creating dangling pointers or. An example of this is shown in Figure 5.1.

As the scoped owner is inside all visible owners, it allows the creation of scoped objects that can manipulate borrowed pointers. In some cases, such objects could also be created using the appropriate borrowed owners, but the confinement of the temporary object to the current stack frame would be lost as would the preservation of separation of argument objects and temporary objects. This is also shown in Figure 5.1.

No preexisting object has the scoped owner in their type. As scoped owners are not outside any preexisting object, the trick to obtain static aliases to argument objects in Section 4.2.4 will not work. Thus, no preexisting object can become dependent on the scoped objects. Consequently:

Aliasing Property 5.2.1 (Scoped Reference)**.**

1. Scoped references and objects do not escape their scoped region.

2. Scoped objects can safely be deleted when the scope is exited.

This confinement of temporary objects has an interesting side-effect: scoped objects *need not be mentioned in pre or post conditions of a method, or in a class' invariants*. Neither need they be considered by outside objects. This is a powerful consequence.

Aliasing Property 5.2.2 (Scoped Reference)**.** As types of scoped references are incompatible with types of representation or argument objects, temporary objects cannot be confused for representation objects or argument objects. Thus, scoped references enable preservation of separation.

```
class CGIProgram
{
    <request inside world, ext outside request>
                        void get( request:HTTPRequest<ext> request )
    {
        // process the request and print using the request object
    }
}

class CGIRunTimeSystem
{
    <system inside world>void prepareGet( system:String uri )
    {
        (request) // conceptually matches the process of the request
        {
            // Create a new request object that parses the uri
            request:HTTPRequest<system> req =
                                    new request:HTTPRequest<system>( uri );
            // Registering output streams, system vars. etc. in req omitted

            // Obtain a reference to the client program
            owner:CGIProgram client = this.clientProgram( );

            // Invoke get in the client program
            client.get<request, system>( req );
        }   // request is now processed, and can be deleted
    }
}
```

Figure 5.1: Scoped regions and owner-polymorphic methods in a simple CGI example. The *prepareGet( )* method is invoked by the run-time system with the get-uri from the browser. The method then creates a more high-level request object from the uri and passes that to the actual client program that should do the actual processing. As the request object only makes sense during the processing and answering (as HTTP is a stateless protocol), the request object should never survive the processing. This is achieved with a scoped region.

We have now described why stack-based confinement is desirable, why simple value objects are not enough to achieve it, and how we can achieve it in Joline with the help of a stack-based owner. We now move on to describe the formal account of scoped regions, before discussing related work.

5.3 Formalising Scoped Regions

Below, we show the extensions of the Joline formalisation with scoped regions. Basically, the scoped region construct allows us to create stack-based owners, by introducing regions other than the *world* region at the bottom of the stack (introduced in Section 3.3.2). A region is pushed to the top of the stack. From the existing rule for generational ownership ((IN-GENERATION), introduced on page 53), the region owner is inside all owners on previous stacks.

5.3.1 Static Semantics

The syntax for scoped region is shown below. It is an addition to the syntactic category of statements. Note that scoped regions can be nested, which works in a straightforward fashion.

s	::=	*Statement*
	(α) { s }	*scoped region*

The scoped region is a block, proceeded by a region name with parentheses. As α is pushed to the top of the stack, it becomes nested inside every other owner on the stack, which has an interesting effect: *ownership no longer forms a tree, but a dag* (see Chapter 7 for an extended discussion). This is the case as it enables one owner to be nested inside several other owners that have no nesting relation among themselves. We exemplify this below.

Consider the following stack:

$$\left(\mathsf{R}_{\texttt{world}}[\, m \mapsto c^{\sigma}[V; H]; \mathsf{nil}\,] \bullet F \right) \oplus \mathsf{R}_{n}[\, H'; F'\,]$$

Here, m and H are nested inside `world` on the bottom stack frame. By the rule (IN-GENERATION) (page 53), any owner introduced on a later stack, is inside all owners in-

troduced on earlier stacks. Thus, n is inside `world`, m and all other owners introduced in H.

Statically, the correct owners must be explicitly passed to the method to allow the references from the later frames to earlier ones. Thus, statically, n will only be considered as inside all *visible* owners, which is likely to be a subset of the owners in H. As any object created on the stack frame is inside all objects on earlier frames, references from the former to the latter respect the owners-as-dominators property.

Now, we extend the syntax of E to allow an owner to be inside several or possibly unrelated owners:

$$E ::= E, \alpha \prec^* \bigsqcup \{p_{i \in 1..n}\}$$

where $\alpha \prec^* \bigsqcup \{p_{i \in 1..n}\}$ means α is inside all owners in $\{p_{i \in 1..n}\}$. We also replace the rule for deriving inside relations, (IN-ENV1), with a more general version respecting the new "inside multiple owner" relation:

$$\frac{\alpha \prec^* \bigsqcup \{p_{i \in 1..n}\} \in E \quad p \in \{p_{i \in 1..n}\}}{E \vdash \alpha \prec^* p} \text{(IN-ENV1)}$$

We write $\alpha \prec^* p$ to mean $\alpha \prec^* \bigsqcup \{p\}$ for simplicity. We can now give the type rule for scoped regions:

$$\frac{E, \alpha \prec^* \bigsqcup \{p_{i \in 1..n}\} \vdash s ; E' \quad \{p_{i \in 1..n}\} \subseteq \mathsf{owners}(E)}{E \vdash (\alpha)\ \{\ s\ \}\ ; E} \text{(STAT-SCOPED-REGION)}$$

The rule (STAT-SCOPED-REGION) introduces a new owner variable that corresponds to a block and is only defined for the scope of the block. The bounds $\{p_{i \in 1..n}\}$, though unspecified in code, determine which objects may be accessed by objects created in this scope. Statically, the owner is inside (a subset of) all owners in the lexical scoped of the block.

5.3.2 Dynamic Semantics

We now give the dynamic semantics of scoped regions.

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

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

The subject reduction proof case for scoped regions is as follows:

Proof. Assume $\Gamma \vdash \langle S \mid (\alpha)\,\{\ s\ \}\rangle$.

1. By (CONFIG-STAT),
   (a) $\Gamma_1 \vdash S$ and
   (b) $\Gamma_1 \vdash (\alpha)\,\{\ s\ \}\,;\Gamma$.
2. By 1.a), and (STORE-TYPE-REGION), $\Gamma_1 \oplus n :: \mathfrak{R} \vdash \diamond$.
3. By 1.a) and (STACK-GEN),
   (a) $\Gamma_2 \vdash S''$ and
   (b) $\Gamma_1 \vdash F' \gg \Gamma_3$ where
   (c) $S = S'' \bullet F'$ and
   (d) $\Gamma_1 = \Gamma_2 \bullet \Gamma_3$.
4. By 2.), 3.b), (FRAME-REGION) and (FRAME-EMPTY),
   $\Gamma_1 \oplus n :: \mathfrak{R} \vdash F' \oplus R_n[\mathsf{nil}] \gg \Gamma_3 \oplus n :: \mathfrak{R}$.
5. By 2.), and (STORE-TYPE-OWNER), $\Gamma_1 \oplus n :: \mathfrak{R} \oplus \alpha \mapsto n \vdash \diamond$.
6. By 4-5.), and (OWNERS),
   $\Gamma_1 \oplus n :: \mathfrak{R} \oplus \vdash F' \oplus R_n[\alpha \mapsto n] \gg \Gamma_3 \oplus n :: \mathfrak{R} \oplus \alpha \mapsto n$.
7. By 3.a,c-d), 6.) and (FRAME-REGION),
   (a) $\Gamma_4 \vdash S \oplus R_n[\alpha \mapsto n]$, where
   (b) $\Gamma_4 = \Gamma_1 \oplus n :: \mathfrak{R} \oplus \alpha \mapsto n$.

8. By 1.b) and (STAT-SCOPED-REGION), $\Gamma_4 \vdash s \gg \Gamma_4 \oplus \Gamma_5$.

   The addition of α to be inside potentially all other owners in scope is equivalent in the dynamics to adding n as the top-most (and thus inner-most—see definitions of (IN-OWNER) and (IN-GENERATION) on page 53) owner in the system and Γ_5 is type information for variables declared in s.

9. By 7.a), 8.) and (CONFIG-STAT), $\Gamma_4 \oplus \Gamma_5 \vdash \langle S \oplus R_n[\alpha \mapsto n] \mid s \rangle$.

10. By 9.) and the induction hypothesis, if $\langle S \oplus R_n[\alpha \mapsto n] \mid s \rangle \rightarrow \langle S' \oplus R_n[H; F] \rangle$, then there exists a Γ_6 such that

    (a) $(\Gamma_4 \oplus \Gamma_5) \rightsquigarrow \Gamma_6$ and

    (b) $\Gamma_6 \vdash \langle S' \oplus R_n[H; F] \rangle$.

11. By 10.b) and (CONFIG), $\Gamma_6 \vdash S' \oplus R_n[H; F]$.

12. By 10.a), 11.), (STACK-FRAME) and definition of $\rightsquigarrow$,

    (a) $\Gamma_6 = \Gamma_7 \oplus n :: \mathfrak{R}[\Gamma_8]$, where

    (b) $\Gamma_1 \rightsquigarrow \Gamma_7$ and

    (c) $n :: \mathfrak{R}[\alpha \mapsto n, \Gamma_5] \rightsquigarrow n :: \mathfrak{R}[\Gamma_8]$, and

    (d) $\Gamma_7 \vdash S'$ and

    (e) $\Gamma_7 \vdash R_n[H; F] \gg n :: \mathfrak{R}[\Gamma_5]$.

13. By 12.b) and 12.d) and (STACK-GEN), $\Gamma_7 \vdash \langle S' \rangle$. Note that by (STAT-SCOPED-REGION), $\Gamma = \Gamma_1$, and thus $\Gamma \rightsquigarrow \Gamma_7$.

□

5.3.3 Related Work

Our scoped region construct is similar to the lexically scoped "letregion" construct used in region-based memory management [123, 125]. There are a number of differences. Firstly, our construct is under programmer control, as in Cyclone [67], whereas the regions calculus is the basis for a compiler's intermediate language. Secondly, we introduce the scoped region to control aliasing, which is not the aim of region-based memory management. Also, the technical machinery used to achieve safety differs: 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 [38]), whereas the regions calculus uses effects to determine that references into a deleted region are never dereferenced.

Both Cyclone [67] and Gay and Aiken's RC [58] manage a nesting relationship which captures when one object *outlives* another, very similar to how our system works, although their systems have neither classes nor subtyping. While some attempts to explicitly add region-based memory management to Java exist [132, 37], 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 [28]. (Other styles of effects system also exist for Java [65, 39, 27, 22].) Although the structural approach lacks the delicacy of the regions calculus, we believe that the scoped regions are better suited to an object-oriented programming language. Most importantly, we deal with subtyping, which the region calculus does not.

Real-time Java [19] includes ScopedMemory objects which behave similarly to our scoped regions, albeit without the static safety guarantees. Later work by Zhao, Noble and Vitek [133] introduces a Scoped Types discipline that enforces these nesting invariants statically.

A number of systems in the literature combine linearity and regions [129, 46], using linearity to track the use of regions to avoid the lexical scoping of region allocation and deallocation in the regions calculus.

5.4 Concluding Remarks

Scoped regions enable confinement of objects to a method body. Without no restriction on the expressive power, scoped objects need not be mentioned in pre or post conditions of a method, or in a class' invariants. This is a powerful consequence.

Scoped regions enable method-local objects to be created with a guarantee that they will not escape. They can thus be safely deleted when the region exits and aliases from scoped objects can be safely disregarded, which facilitates reasoning and preserves separation of temporary objects and an object's representation.

Using owner-polymorphic methods, a completely orthogonal construct, defined in Chapter 4, scoped references can be exported to subsequent stack frames, similar to borrowed pointers.

In addition to confining pointers to a specific scope, scoped regions have memory management advantages in that memory transitively consumed by objects belonging to the scoped region can be immediately reclaimed when the region exits.

Pizlo et al. [108] suggest two types of functions that will especially benefit from scoped regions in terms of memory management: *sinks*—methods whose results are not returned but relayed to an external entity such as a file, and *pure functions*—methods that do not modify external state. For example, a method that prints data to a file might create a temporary file object, a temporary stream to write to the file and possibly several temporary objects for concatenated data to write to file in fewer, but larger chunks.

We now move on to describing a third orthogonal construct that extends Joline with externally unique pointers, "non-unique uniques that are effectively unique".

Chapter 6

External Uniqueness

6.1 Contribution

Unique pointers are simple and powerful. In many proposals [73, 92, 7, 27, 6, 32, 87], their power is, however, hampered by the fact that the uniqueness is for only one object as opposed to an entire aggregate. Also, it turns out that all *realisations of uniqueness* prior to our original external uniqueness paper [44] violates the principle of abstraction, suggesting that the traditional uniqueness definition is not well suited to object-oriented programming.

In this chapter, we introduce *external uniqueness*, a realisation of uniqueness constructed on top of deep ownership. In short, external uniqueness *overcomes the abstraction problem* and allows a flexible form of aggregate uniqueness that *relaxes the uniqueness definition for internal pointers without compromising effective uniqueness*. Some subsequent proposals [21] have adopted our approach.

In the presence of ownership types, external uniqueness comes virtually for free in a programming language. Moreover, as the proposal is ownership-based, it integrates perfectly with the previously proposed constructs, owner-polymorphic methods and scoped regions to simulate borrowing of unique values. Thus, we realise uniqueness without introducing additional pointer categories and can even allow borrowed references to flow into the heap. All in all, we believe that our uniqueness proposal is better suited to object-oriented programming that traditional uniqueness.

We begin this undertaking by recapping some fundamental points of uniqueness

before describing realisations of uniqueness in an object-oriented setting, the problems and present solutions and how this causes a problem with abstraction.

Large parts of this chapter was published by Clarke and Wrigstad [44]. The presentation has however evolved with improved examples and illustrations. Specifically, the analysis of constructors is new, as is the dynamic semantics and proof of soundness.

6.2 Recapping Uniqueness

In Chapter 2, we introduced the concepts of uniqueness, borrowing and destructive read. In this section, we detail the description of uniqueness, just as we did with ownership types in the description of Joline. In particular, we identify three shortcomings of (most) existing uniqueness proposals: they only apply to single objects and not aggregates; back-pointers to unique bridges are not possible in conceptually unique aggregates; and, most importantly, they all violate the principle of abstraction.

6.2.1 Uniqueness and Object-Orientation

A variable or field annotated with the keyword *unique* contains a unique pointer or *null*. Unique stack variables have an interesting, strong aliasing property:

Aliasing Property 6.2.1 (Unique Variables)**.** Statements and expressions that do not explicitly involve the unique variable cannot effect the object to which the variable refers.

A study by Noble and Potanin [98] suggests that uniqueness as a concept fits well with the current ways of constructing object-oriented software: inspection of heap dumps of running programs from the Purdue Benchmark Suite has shown that as much as 85% of all objects are uniquely referenced in a program. This study is optimistic, as uniqueness violations could occur in-between the heap dumps and this go undetected by the analysis. However, more fine-grained, less optimistic studies of smaller programs have shown similar results, suggesting that Noble and Potanin's optimistic results are correct.

The aliasing property of unique stack variables is not shared by unique fields as the object containing the unique field can be arbitrarily shared. Thus, the field can be accessed by any statement and expression that have access to the object. To obtain

as strong a property for unique fields as for unique stack variables, some proposals restrict unique fields to only appear in unique objects [127]. We believe that this is not well-suited to object-oriented programming as a unique field would require that the enclosing object was uniquely referenced. This is very inflexible. Unique references could not be stored in double-linked lists and introduction of a unique field in a subclass would require all instances of superclasses to be uniquely referenced. Furthermore, letting an object's implementation control how it can be referenced externally leads to a problem with abstraction. This will be discussed shortly.

Unique values are painful to program with as they are destroyed when read. For example, passing a unique object as an argument to a method will consume the argument. If the argument was to be used only temporarily by the method, it must be explicitly returned. An example of this was shown in Figure 2.5. As the receiver of a method is really an implicit argument, when a method is invoked, an alias to the receiver is implicitly created on the new stack-frame. This alias will invalidate uniqueness of a unique receiver, unless it is destroyed at te call-site, which requires it to be explicitly returned and manually reinstated. This is naturally tedious and prone to errors. An example of destructive reads and manual reinstatement for receiver arguments is found in Figure 6.1. A less tedious solution using alias burying is found in Figure 6.2.

```
unique List prepend( Object data ) // method in List class
{
    this.first = new Link( data, this.first );
    return this--;
}

unique List list = new List( );
list = (list--).prepend( new Object( ) );
```

Figure 6.1: Destructive reads and manual reinstatement. To preserve uniqueness of *list* when the method is invoked, we are required to nullify *list*, indicated using --. Thus, the list must be explicitly returned when the method exits and manually reinstated.

As it turns out, the presence of a *this* pointer (or equivalent) increases the complexity of 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

```
void prepend( Object data ) // method in List class
{
    this.first = new Link( data, this.first );
}

unique List list = new List( );
list.prepend( new Object( ) );
```

Figure 6.2: Alias burying to avoid destructive reads. From the code, we can statically infer that *prepend( )* will not create an alias to its receiver. At the call-site, the unique receiver is stored on the stack and can thus not be accessed during the method call. Thus, *list* is sufficiently buried and the method call does not invalidate its uniqueness.

assigns *this* to a variable or field, invoking the method on a unique variable should consume the variable's contents to maintain uniqueness of a reference. In the presence of borrowed pointers, such receiver consuming methods must not be invoked on borrowed values, which introduces additional complexity to the treatment of uniqueness. In addition, if a constructor stores *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 return a non-unique reference.

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

1. the implicit destruction of unique receivers,

2. how to restrict a method's use of its receiver to allow it to be reinstated after a call, and

3. how to make sure that a constructor returns a unique reference.

Different approaches in the literature reflect the treatment of *this* by annotations on a class' interface in two ways. The annotations are necessary to achieve modular checking of the uniqueness invariant and are either at class-level or method level:

Via class annotation Classes are divided into two kinds, those whose instances may assign *this* internally, and those whose instances may not. Only instances of the latter may be referenced uniquely [92].

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

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

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

Figure 6.3: Using class-level annotations to control how *this* is treated internally. The *connect( )* method will not compile as it is creates an alias to *this* and passes it as an argument to the *setManager( )* method in *client*.

Via method annotation Methods are annotated to indicate that they may consume *this* [73, 30]. Calling such a method requires that its target be destructively read (or equivalent, in the presence of an effective uniqueness scheme).

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

Class-level Annotations

Class-level uniqueness annotations, proposed by Minsky [92] in Eiffel*, decorates class declarations and controls whether instances of a particular class can or cannot be uniquely referenced. In the example in Figure 6.3, a class *Server* is annotated with the uniqueness keyword allowing its instances to be uniquely referenced. The unique annotation requires that all methods in the server class are anonymous (do not capture this), that is they don't assign *this*, or pass *this* as an argument to a method. As all methods in a unique class are anonymous, *this* can be safely used as a receiver in all methods as no method will alias *this*.

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

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

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

Figure 6.4: Using method-level annotation to control subjective treatment of *this*

Method-level Annotations

Method-level annotations, used by for example Hogg [73] and Boyland [30], are annotations placed on methods instead of on a class. Each method is annotated to reflect its treatment of *this* and allow only methods that do not capture this to be invoked on unique receivers. Thus, method-level annotations allows an object to be uniquely referenced regardless of its class' implementation. This is more fine-grained and thus more flexible than class-level annotation. The price is a slightly increased syntactic overhead. Figure 6.4 shows the *Server* class from Figure 6.3 using *consumes* and *anonymous* annotations, similar to Boyland's proposal.

The *getConnection()* method is now annotated with the *anonymous* keyword, meaning that it does not create an alias to this *this* on the heap. The *connect()* method is annotated with *consumes* meaning that it will create an alias to its receiver object if invoked on a unique pointer. Validity of these annotations can be controlled by a simple compile-time check

Method-level annotations allow the mixing of consuming and non-consuming methods in the same class. As with class-level annotations, some additional constructs are required to enable unique references. We either need destructive reads to ensure that a variable used to invoke a consuming method will be destructively read to preserve uniqueness, or some equivalent mechanism such as alias burying [30] to make

sure that uniqueness is maintained.

A third alternative is to simply weaken the uniqueness invariant and allow several pointers to a supposedly unique object to exist simultaneously. Examples of systems where the uniqueness invariant is weakened are Eiffel* [92], AliasJava [6], Balloons [7], Pivot Uniqueness [87] and Capabilities for sharing [32]. In these systems, borrowing weakens uniqueness since the unique reference is still visible in the system during the borrowing. Thus, another thread, or a reentrant method in the same thread, might use the reference during the borrowing, effectively violating the uniqueness.

6.2.2 Problems with Class-Level and Method-Level annotations

Problems with Class-Level Annotations

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

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

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

Last, instances of classes not annotated with the *unique* keyword cannot be uniquely referenced, even if the class' implementation would allow it as only instances of classes *annotated* with unique are allowed to be referenced uniquely.

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

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

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

Figure 6.5: Subclassing with class-level annotations. Instances of class *A* are never unique. However, if we are allowed to subclass *A* with a unique class *B*, uniqueness of unique references to *B* objects can be invalidated if a method call binds to a method defined in *A*.

Constraining Evolution As is illustrated in Figure 6.5, the (non)uniqueness annotation must be preserved through subclassing as uniqueness could otherwise be invalidated by overriding methods that created aliases to *this*. This makes extension via subclassing harder or less powerful since the annotation of the superclass must be respected by all subclasses.

It might be possible to allow unique classes to have non-unique subclasses as the implementation of the unique superclasses work even if *this* is not unique, but as Figure 6.5 clearly shows, not the other way around.

Problems with Method-Level Annotations

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

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

6.2.3 Uniqueness and a Problem with Abstraction

In this section, we detail the discussion about the abstraction problem with both styles of annotation above. In both cases, and for methods and constructors alike, a problem surfaces when the implementation of a class changes the way it uses *this* which leads to a violation of the principle of abstraction. We will now examine this problem in more detail, recapping the arguments from Clarke and Wrigstad [44].

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

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

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

```
unique BlackBox bb;
```

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

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

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

With Class Annotations

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

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

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

With Method Annotations

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

Using method annotations, the *xyzzy( )* method would have been annotated **anonymous**. However, the addition to the method forces it to be changed to **consumes**. Potentially, this forces much fewer changes to the program compared to class-level annotations, as instances of *BlackBox* may still be uniquely referenced. However, the call *bb.xyzzy( )* will now create an alias to its receiver requiring that the variable *bb* is nullified to preserve uniqueness. Depending on the realisation of uniqueness this change from **anonymous** to **consumes** might propagate as an addition of a destructive read operation to the calls. If this is not the case, the result is even more drastic, as the behaviour of the method call has changed silently from the original program to consume its target. This is both awkward and counter-intuitive.

Concluding Remarks

In both cases, a purely internal change to the implementation of the *BlackBox* class 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, also it means that objects cannot be treated like black boxes, because:

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

Thus extant uniqueness proposals break abstraction.

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

6.2.4 Uniqueness and Aggregates

Unique pointers provide effective encapsulation of part of an object's representation. Even though there is nothing to prevent uniquely referenced parts of the state from *escaping* its enclosing object, they cannot be aliased. When exported, rather than creating an incoming alias, a unique reference is *moved out of its aggregate*. As this is the only reference to the object in the system, it means that the object has effectively moved too, and become part of some other object's representation.

Consider an aggregate object with a *bridge object* providing an interface to the aggregate. Even if the aggregate is unique, its representation need not be and so, even though it may look as if the aggregate is uniquely referenced, incoming aliases to its representation might exist that prevents us from treating the aggregate uniquely, which is a problem with threads, for example.

For concreteness, we present a small code example:

```
unique Aggregate agg = new Aggregate( );
Object incoming = agg.rep;
Thread t1 = new Thread( agg-- );
incoming.messUp( );
```

Here, *agg* is a unique reference to an aggregate. We obtain a reference to a representation object and store it in *incoming*. Now, passing the unique *agg* reference to the thread's constructor in line 3, one would expect the aggregate to become thread-local to the new thread. However, as the last line shows, this is not the case, as *residual aliasing* to parts of the aggregate exists in the current thread. Thus, only the bridge object is moved, while (some or all of) the objects in its representation become effectively shared between two threads or two aggregates. Graphical depictions of this are found in Figures 6.6 and 6.7. We call this *horizontal slicing* and it is a clear breach of encapsulation. Thus, even though unique references can be said to provide some form of encapsulation, the encapsulation is not transitive.

Clearly, this behaviour may cause objects the be shared between threads, suggesting that using uniqueness in place of synchronisation is only suitable for individual objects.

To tackle the problem of incoming aliases to a uniquely referenced object's representation, the encapsulation of uniqueness must apply to an entire aggregate. This is the case in Hogg's Islands [73] and in Boyapati et al.'s Parameterised Race-Free Java [27].

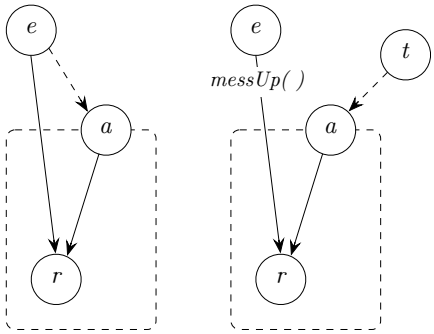

Figure 6.6: Sharing precludes thread-localness. The dashed arrow denotes a unique reference. The object *a* is a uniquely referenced aggregate and *r* is its supposed representation, shared with some external object *e*. *t* is the thread object from the code example on the previous page.

However, Islands does not allow outgoing aliasing, which we believe is too restrictive to support actual programming, and Parameterised Race-Free Java suffers from the abstraction problems identified earlier in this chapter. Furthermore, its allowing the unique keyword as an owner parameter violates parametricity [112] causing an additional abstraction problem, as this leaks the treatment of internal variables out into the interface. This is in contrast to regular ownership types, where owners have no effect on semantics. We have discribed this elsewhere [131].

We now move on to describe our take on uniqueness that not only overcomes the abstraction problem, but also enables a strong notion of uniquely referenced aggregate which allows back-pointers to the unique bridge object without effectively weakening the uniqueness invariant.

6.3 External Uniqueness

In this section, we describe external uniqueness, a uniqueness built on top of Joline's deep ownership types.

Definition 6.3.1 (Externally Unique Object)**.** An object is externally unique if the number of external references to it is at most one.

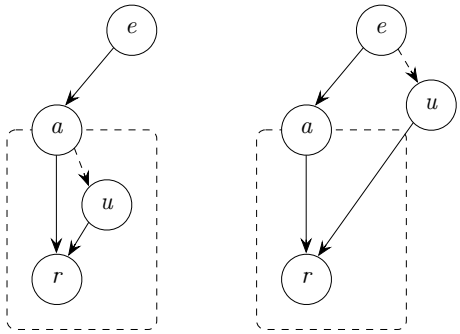

Figure 6.7: Moving a unique can mean slicing an aggregate. The dashed arrow denotes a unique reference. The object *a* is an aggregate and *u* and *r* is its representation. Object *r* is shared internally between *u* and *a*. When *u* is moved outside of *a*, *r* is exposed externally of *a*, causing "slicing".

This is a reasonable definition. First, it considers *aggregate uniqueness*, much in the style of Hogg's Islands [73], but allows outgoing aliases to objects external to the aggregate. Second, it even allows *back pointers* to the bridge object from objects within the aggregate. None of the existing proposals allow this. Surprisingly, as we shall see later, externally unique pointers are effectively unique.

External uniqueness is realised as an extension to ownership types. The extension is minimal, but the consequences great. As our uniqueness is built on owners, it also allows us to use the aforementioned constructs, scoped regions and owner-polymorphic methods, to simulate borrowing, without having to extend them. Because of the nature of our borrowing proposal, we also allow borrowed pointers to be stored temporarily on the heap, which was previously not possible in existing systems. (A subsequent proposal by Boyapati has adopted our way of realising uniqueness [21].)

We begin by describing unique owners, the key to achieving external uniqueness. We then describe operations on externally unique pointers, discuss ways of maintaining uniqueness when accessing unique variables and how to deal with constructors. Last, we discuss the formal properties of external uniqueness, why external uniqueness is effectively unique, and how external uniqueness overcomes the abstraction problem, before showing the extension of Joline's formalisation to include externally unique pointers.

From now on, we will write unique to mean externally unique, adding additional qualifiers when a distinction is necessary.

6.3.1 Unique Owners

Our realisation of external uniqueness relies on ownership types.

It is instructive to view owners as permissions—unless an object is explicitly given the permission to reference some other object's representation, the necessary types cannot be formed and thus such references cannot exist. The design rationale behind our external uniqueness proposal is the following: if an object has a properly confined, *unique owner*, the object's type can only be formed at *one* specific place, precluding aliasing.

The syntax for unique types is similar to the traditional ownership types syntax:

$$\textit{unique:ClassName}\langle \mathrm{p}_{\mathrm{i}\in 1..\mathrm{n}} \rangle$$

The owner parameters $\mathrm{p}_{\mathrm{i}\in 1..\mathrm{n}}$ are regular owner parameters. The *unique* keyword is not an owner in itself. It is simply shorthand for saying that *the owner of the object of this type is the the field or variable that holds the reference to the object*. Thus, the declaration "*unique:List*⟨*data*⟩ *x*" is equivalent to "*x:List*⟨*data*⟩ *x*", where *x* is a variable name rather than an owner parameter from a class or method header. By not allowing variable names to be used directly as owners, we implicitly prevent a programmer from giving the type of another variable the owner *x*. As follows from our interpretation of the unique keyword, no two unique types can be the same as variables or field are "dynamic locations" and therefore *unique in the system*—the field *f* in different objects are different fields and two variables named *x* on different stack frames are different variables.

As an example of the use of the *unique* keyword, the types of the two variable declarations below are syntactically equivalent, but denote different types—one for objects owned by the variable *x* and one for objects owned by the variable *y*.

unique:List< data > x;
unique:List< data > y;

This design decision makes the syntax clearly reflect the semantics. It also resembles most other proposals for unique references.

A side-effect of unique owners is that assigning a unique reference from one variable to another requires a change of the owner of the referenced object. Consider the variables x and y from the example above. The types are different, but assignment compatible, as they are both denote uniquely referenced lists whose elements are owned by *data*. The assignment $x = y$ must change the owner of the object referenced by y from y to x. We call this *moving* as we require y to be nullified, as otherwise, there would be two references to the object with conflicting types and uniqueness would also be invalidated. Uniques that are free values are owned by the special owner *free*. Merely reading a unique value makes it free and gives it the *free* owner, and reading a unique variable and then storing its contents in another performs two moves, from y to *free* and from *free* to x.

Having introduced the unique owners concept, we can define unique types and unique references:

Definition 6.3.2 (Unique Type)**.** A type with the unique keyword as its owner.

Definition 6.3.3 (Unique Reference)**.** A reference that has a unique type.

In ownership types, the owner of an instance is statically accessible to its representation via the keyword *owner*. As a consequence, even if the unique keyword produces a unique owner *externally*, the owner denoted by the unique keyword is accessible *internally* in the object via the keyword *owner*. Thus, it is possible to form types internally that could alias the unique reference, rendering the reference not really unique. As these types will not use the unique keyword, they will not denote unique references. However, as we do not allow field access or method invocation on unique references (we will return to this issue shortly). This means that aliases to the object cannot be stored in or retrieved from the fields of an object when it is uniquely referenced. Thus, any internal pointers are inaccessible and cannot be used to obtain an alias that would break the object's external uniqueness. We can now restate Definition 6.3.1 in terms of Definition 6.3.3:

Definition 6.3.4 (Externally Unique Object)**.** An object is externally unique if it is referenced by a unique reference.

As opposed to Boyapati et al.'s [27] proposal, the unique keyword can only occur in the owner position of the type and not in any other owner parameter. As a consequence, we cannot specify externally whether the fields of an object will contain a

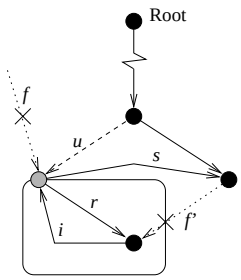

Figure 6.8: External Uniqueness. The gray object is externally unique, but has a back-pointer to itself internally.

unique reference or not. We chose to avoid this extra complexity for a number or reasons: First, we identify an additional problem with abstraction in the fact that additional clauses are necessary to prevent certain owner parameters from being instantiated with unique owners. Internal changes to a class' implementation may require that a specific owner parameter is no longer instantiated with unique. This might invalidate several previously well-formed types in the program. Second, whereas parametrically polymorphic classes have the same semantics and invariants regardless of the the type of its fields, this is not the case if the uniqueness of the fields can be controlled externally. Thus, this violates parametricity [112]. Third, we believe our solution is cleaner, and has a much more natural semantics.

Having introduced the unique owner concept, we move on to describe the operations on externally unique pointers.

6.3.2 Operations on Externally Unique Pointers

We aim to make our system as clean and simple as possible to make it play well with other constructs and for the constructs themselves to remain orthogonal and combinable. To this end, we allow only two operations on unique pointers, *movement* and *borrowing*. The borrowing operation is similar to borrowed references due to owner-polymorphic methods, but uses an locally created owner, similar to a scoped region.

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

Movement

To make the syntax clearer, and also to simplify the formal account of the Joline language, we chose to make destructive reads explicit. Consequently, a programmer must write *x* = *y*-- or **return** *y*-- instead of *x* = *y* and **return** *y* respectively. For fields this becomes *x* = *y.f*-- and **return** *y.f*--. Thus, the syntax of movement becomes:

```
y--;        // make contents of y a free value, nullify y
x = y--;    // move contents of y into x, nullify y
```

In our system, reading a unique variable has the side-effect of updating the variable with *null*. Thus, unique values are effectively *moved* instead of aliased when assigned from, as the source variable is nullified. As every unique type in a system has a unique owner, movement also implies *transfer of ownership* from one variable to another, or to the special owner *free* for free values. Reading a unique variable or field without assiging the result makes the unique *free*. Assigning it moves it into the owner of the target variable, possibly losing uniqueness depending on the target's type.

Movement and Subsumption

A unique may be moved anyplace where its owner parameters are already visible. Otherwise, uniques could be used to break the owners-as-dominators property. For example, an object with type *unique:Foo*< *bar* > can hold references to objects owned by *bar*. If a unique reference to such an object is allowed to move outside *bar*, the encapsulation of *bar*'s representation is breached, indirectly via the moved object. For a graphical depiction of this, see Figure 6.9.

However, in presence of subtyping, the situation becomes even more subtle. In ownership types, subsumption allows us to "forget" owner parameters, as long as the owner itself is invariant. For example, *bar:Object* is a supertype of *bar:List*< *data* >. This is safe, as the owner is invariant for non-unique references. For uniques, however, if *unique:Object* is a supertype of *unique:Foo*< *bar* >, the information that the object cannot move outside of *bar* is lost—if the object were to move outside of *bar*, *bar*'s representation could then be accessed without going through *bar*, which would break the owners-as-dominators property.

To address this problem, we introduce *movement bounds* that bound the movement of uniques and preserves owners-as-dominators in the presence of subtyping.

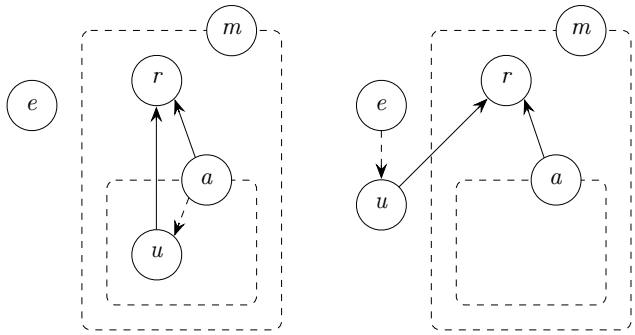

Figure 6.9: Movement breaking encapsulation. The reference $a \rightarrow u$ moves to $e \rightarrow u$ causing $u \rightarrow r$ to break m's encapsulation. With movement bounds (see text), movement of u must be restricted to inside m for u to be allowed to reference r. This would preclude the encapsulation breaking move in the figure.

Movement bounds govern the maximal outwards movement of a unique. The syntax for a unique type with a movement bound is *unique*[p_0]:*List*< $p_{i \in 1..n}$ >, where $p_{i \in 0..n}$ are regular owners, and p_0 is the movement bound. Similar to owners of non-unique types, p_0 must be inside $p_{i \in 1..n}$ when the type is formed as the object could otherwise move in a way that would break owners-as-dominators. When implicitly derived, the movement bound defaults to *owner* except in static contexts, where owner defaults to *world*.

In contrast to normal owners, movement bounds can change, but only inwards. This is always safe and can never lead to encapsulation breaches, as inner objects are allowed access to outer objects. Thus, the owners-as-dominators property is preserved. In the formalism, *unique*[p] is written $\texttt{unique}_{p}$. Movement bounds can be implicitly derived by the system, or explicitly stated.

Choosing movement bound requires a trade-off. An outer bound enables more movement, but limits what other objects the object can access (specifically, what ownership parameters can appear in its type). An inner bound enables less movement, but permit more objects to be accessed. Unique references with movement bound *world* can be moved anywhere in the system, but just as for objects owned by *world*, they can only statically alias other *world*-owned objects in addition to its representation.

In addition to governing movement of uniques in the presence of subtyping, movement bounds can also be used to restrict the movement of the values in a unique field,

effectively enforcing deep encapsulation even for the unique values themselves. For example, consider the class *Example* below using traditional uniqueness to controlling alias of a representation object stored in the field *rep*.

```
class Example
{
    unique Object rep;
}
```

In the example, even though the contents of *rep* will be encapsulated in the enclosing object, there is nothing to prevent its contents from being exported outside of the object. This can be achieved very simple with movement bounds:

```
class Example
{
    unique[this]:Object rep;
}
```

Here, the unique type of the variable *rep* has movement bound *this*. In our system, the contents of *rep* can thus only be moved to places inside *this*, guaranteeing that *rep* will always be encapsulated inside its enclosing object.

Borrowing

We begin our descrition of our borrowing construct for unique variables and fields with a recap of existing borrowing constructs in previously proposed systems.

Many proposed systems [73, 92, 88, 27, 6] use borrowing to tackle the "slipperiness" [31] of unique pointers. A unique variable may be passed as a borrowed parameter to a method. Borrowed arguments may only be used to invoke anonymous methods (methods that borrow their receiver argument) and can only be passed to another method as a borrowed argument. A borrowed argument may not be returned nor stored on the heap. Thus, all borrowed references created by a method will be destroyed when the method exits. This alleviates some of the pain of programming with unique values as uniques passed as borrowed (receiver) arguments can be automatically reinstated when the method that borrowed them exits as this method is guaranteed not to have created any static alias to the borrowed object.

In existing systems using borrowing, borrowed pointers are an additional kind of pointers that may not be stored in the heap nor returned from methods. These are

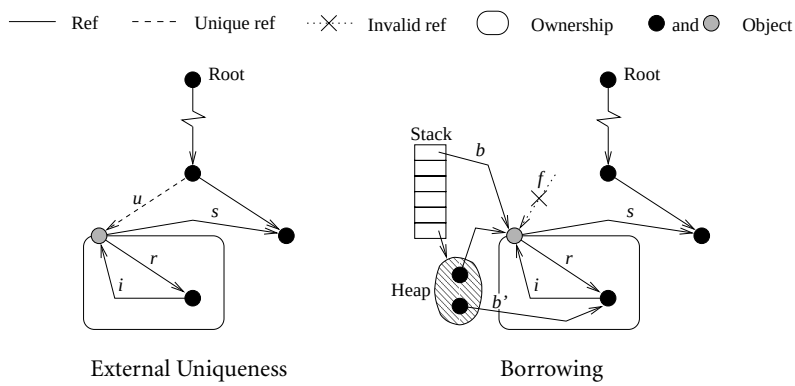

Figure 6.10: Mediating between external uniqueness and borrowing — b is the original borrowed reference. b, b′ are only valid during the borrowing. (Stack grows downwards.)

very arbitrary restrictions that are likely to make programming with borrowing more complex and less flexible. Provided these were destroyed when method exits, it would be safe to allow borrowed arguments to be stored on the heap, similar to how we can treat arguments to owner-polymorphic methods presented in Section 4.2.3. The type systems of previous proposals have not been strong enough to express such constraints. We chose to tackle borrowing in a completely different way, using owners to govern where borrowed pointers may flow.

In our system, we introduce an additional borrowing statement which temporarily moves a unique value into a non-unique value confined to a block. The syntax of the borrowing statement looks like this (to simplify matters, we use a slightly different syntax for the formal system.):

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

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

When the borrowing block is evaluated, the content of *lval* is destructively read and moved into *var*, with implicit transfer of ownership. The owner of *var* is *temp*, which is ordered inside the movement bound of the type of *lval*. As *temp* is a fresh owner, there can be no preexisting variables or fields with a compatible type, and the only variables that can have types that contain *temp*, must be created in the scope of the block.

Owner-polymorphic methods may be used to export *var* outside of the current method. As the owner introduced by the block is not visible to any preexisting object, *temp* can only be bound to borrowed owners. Thus, *var* can only be passed from the block as a borrowed argument. Any aliases created by such methods are lost when they exit, unless they are returned, something that previous borrowing proposals had to prevent not to lose track of the borrowed references.

As the borrowing blocks acts as guards that delay reinstatement of the borrowed references, returned aliases from an owner-polymorhic method pose no problem. As we delay the reinstatement till the exit of the borrowing block, all variables holding returned aliases from a borrowed method are out of scope. Thus, when the block exists, no references to the borrowed object can exist, and any any reference we store back into *lval* will be externally unique—*var* can be safely stored back into *lval*. This is powerful, as it allows us to have an externally unique aggregate that changes bridge object as a result of a borrowing.

In conclusion, our borrowing construct differs from previous ones in that it does not introduce an additional kind of pointer in the system. Rather, borrowing converts an object between being uniquely and non-uniquely referenced. Instead of using a crippled borrowed pointer, we rely on the ownership types system to make sure that when the borrowing ceases, all aliases to the borrowed object are invalidated, allowing us to once again view it as unique. For a graphic depiction of this, see Figure 6.10.

Maintaining Uniqueness

As we have already stated, we chose destructive reads as our approach to maintain uniqueness when moving unique values. We now discuss ways of maintaining uniqueness when a unique value is borrowed using our borrowing construct. We use the following example:

borrow *x* **as** *temp:var* **in** { *x.doSomething( );* }

There are essentially three approaches to maintaining uniqueness of *x* in the block above:

Do Nothing Rather than invalidate the contents of *x*, 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. This would allow *x.doSomething( )* to evaluate successfully.

Destructive We could nullify the borrowed variable during borrowing, and then do either of the following:

- Simply restore the original contents of the borrowed variable when the borrowing ceases. This would cause *x.doSomething( )* to throw a null-pointer error;
- Restore the final contents of the borrowing variable at the end of the borrowing.

  Restore the initial value is consistent with traditional uniqueness, whereas enable 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 would allow *x.doSomething( )* to evaluate successfully.

  This solution would give the programmer full control, dynamically, of the possible ways to handle an attempt to borrow an already borrowed variable or field.

Alias Burying The last possibility is to employ alias burying [30]; 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, goes out of scope or is otherwise invalidated. This would ensure that when the variable is read, all its aliases are unusable [30]. 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 [31].

This may well reintroduce the abstraction problem, for example if a method's implementation is changed so that a previously borrowed variable's content 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 equally strong aliasing properties given by both schemes ensure 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, such as allowing the simultaneous invocation of two read-only methods on one unique reference during a borrowing.

Borrowing can be implemented to maintain uniqueness or not. Our system can support both, but for simplicity, we only consider the second case here and in our formalism. For borrowing that maintains uniqueness for the borrowed pointers, see Section 9.1.

6.3.3 Creating Unique Objects

Unique objects can only be created through instantiation. However, object creation in presence of uniqueness requires special attention [45], as constructors could create aliases to the object and thus invalidate uniqueness.

Constructors are effectively methods called only once but otherwise 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, the result of an object creation would then not be an externally unique object.

An argument that could store static aliases to the object being created would need the created object's *owner* as an owner parameter. However, this is not possible, as the owner of an object being created is a fresh owner and there is no way it can be present in external, previously existing objects.

Having discussed how to deal with unique object creation in the presence of constructors, we omit constructors from our language description for simplicity.

6.4 Discussion

We have now described external uniqueness and how to realise it on top of a deep ownership system using only minor tweaks and with one additional construct, borrowing. As opposed to previous proposals [30, 32, 27, 6], our borrowed pointers are regular pointers rather than of a special kind. This is a more flexible and less complex solution as borrowed pointers can be stored on the heap and we avoid the need for a third reference category (non-unique, unique, borrowed). In this section, we introduce the external-uniqueness-as-dominating-edges property, argue why external uniqueness is effectively unique despite the presence of back-pointers, and show how external uniqueness overcomes the abstraction problem.

6.4.1 Aggregate Uniqueness and Dominating Edges

Most uniqueness proposals [92, 32, 53] facilitate unique references to individual objects. External Uniqueness facilitates *aggregate uniqueness*, where a unique pointer is the single entry point to an entire aggregate (or more specifically, all representation objects within the aggregate). This is very similar to the owners-as-dominators property of ownership types, only slightly stronger. An externally unique pointer is a *dominating edge* of the representation objects in the aggregate: all paths from the root of the object graph to the object *all go via the same edge (unique reference) to it*. As opposed to owners-as-dominators, the dominating edge property holds not only for the heap, but for the stack as well.

Ownership types enables a strong notion of aggregate [45]. With external uniqueness, a uniquely referenced object in a system with deep ownership is a uniquely referenced aggregate, similar to an Island [73], but allowing outgoing aliases to external objects.

6.4.2 Back-pointers and Effective Uniqueness

By virtue of the external-uniqueness-as-dominating-edges property, back-pointers, pointers from within an aggregate to the unique bridge-object, cannot become active when the unique pointer is in place.

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

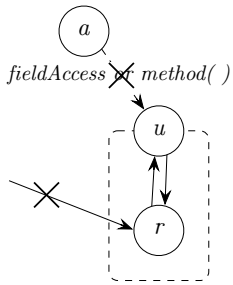

Figure 6.11: Back-pointers are innocuous since they cannot be accessed from outside. The reference $u \rightarrow r$ cannot escape through $a \rightarrow u$ as that reference may not be used to access fields or call methods. Also, as the representation of u is protected by deep ownership, references to r from outside u are not possible, and can thus not be used to access the back-pointer $r \rightarrow u$ outside u.

fields of the object it refers. Thus, we cannot reach the possible internal pointers that violate actual uniqueness. This is shown in Figure 6.11 below.

Thus, we can allow back-pointers without effectively weakening uniqueness, making external uniqueness effectively unique [45, 131]. Thus, the aliasing properties of externally unique variables are the same as for traditionally uniques ones (see Definition 6.2.1)

6.4.3 Overcoming the Abstraction Problem

As described earlier in this chapter, extant proposals for uniqueness suffer from an abstraction problem caused by annotations to reflect how an object treats its *this* variable. In the case of Boyapati et al.'s SafeJava [27], adding uniqueness cases an additional problem as the proposal violates parametricity.

Overcoming the abstraction problem with uniqueness requires that details about how the object treats its this pointer are not visible in its interface, and that an object's implementation cannot change how the object can be referenced externally. The key to achieving this in the presence of uniqueness is to preclude subjective movement, that is, not allow the object to move itself. In our system, the type of *this* always have the owner *owner*, which is non-unique and therefore cannot move. Thus, a class' implementation cannot effect the possibility of its instances to be uniquely referenced.

If an object does not create internal aliases to itself, is is actually unique (in the

traditional sense); if it does, it os 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.

Since instances will never see 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).

Furhtermore, instantiating an owner with *unique* does not propagate through implementation as in SafeJava. There is thus no need for *where*-clauses [27] or similar constructs to control which objects or owner parameters can be uniquely referenced respectively 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.

> *Our realisation of uniqueness thus decouples a class' implementation from how its instances can be referenced. Thus, details of the class' implementation need not propagate into its interface which preserves abstraction.*

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.

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 6.12 shows the implementation in Joline of a *Server* class used earlier in this chapter in the description of the abstraction problem. The difference between this figure and the previous one is the complete absence of any annotations concerning uniqueness or the treatment of *this* in this figure. This is consistent with external uniqueness. In the case of class-level annotations, the method *connect( )* would preclude unique references to the *Server* object, as the method stores a reference to the server in a client object. In the case of method-level annotations, this method would

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

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

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

Figure 6.12: The *Server* class example from Figures 6.3 and 6.4 encoded with external uniqueness

be *consuming* and not invokable on unique receivers. 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 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 ‡.

Now, let's consider the effects of changing the code of the figure, in particular replacing the entire *getConnections( )* method by the following code, making the method a consuming method:

```
Int getConnection( )
{
    this:BlackBox< owner > bb = new this:BlackBox< owner >( );
    bb.xyzzy( this );                  // Consumes this
    return this.noConnections;         // †
}
```

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 any change

to the method header. The *getConnections( )* method can only be invoked on a non-unique reference. Thus, the method can only be invoked from within the object, or an external, non-unique pointer, which precludes any unique aliases to *this*. Thus, creating an additional alias to *this* is perfectly valid.

In case of traditional uniqueness, using class-level annotations in the style of Minsky [92], 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 replaced for **neverunique** , a clear case of the internal use of *this* leaking out in interfaces.

Using method-level annotations in the style of Hogg [73] and Boyland [30], the same problem appears but with different symptoms: the *getConnections( )* method is forced to be declared as **consuming** its receiver, and the first invocation of it will steal the only reference to the server, causing a null-pointer exception at line †.

6.5 Formalising External Uniqueness

In this section we present the formalisation of external uniqueness in Joline. Before doing so, we give an example, which is not valid in our system, of how owner parameter mapping from subclasses to superclasses can make the unique owner appear in non-owner position of a type.

The *extends* clause in Joline allows the forgetting of owner parameters in the mapping of the parameters of the subclass to those of the superclass. An example of this is shown below where *owner* is mapped to *foo* in its superclass. This allows the owner parameter list to vary in the hierarchy.

class *Frob*< *foo* **outside** *owner* > { *foo:Object fu;* . . . }

class *Bar* **extends** *Frob*< *owner* > { . . . }

world:Bar sub = **new** *world:Bar( );*

world:Frob<*world*> *super* = *sub;*
super.fu; // has type world:Object

While syntactically valid, the code contains a hidden error. If the owner of *sub* is unique, then the type of *super* would end up as *unique:Frob*< *unique* >, and the type

of *super.fu* as *unique:Object*, giving the impression that *fu* is unique, which is not the case.

The simple solution to this problem is to ban *owner* from being used in the extends clause of a class declaration. This is a minor restriction, as owner is always accessible internal to the object anyway, via the *owner* keyword.

6.5.1 Static Semantics

Below, we show an extension to the syntax of Joline with destructive reads, borrowing blocks and unique owners.

e	::=		*Expression*
		lval`--`	*destructive read*
		`(p)` e	*lose uniqueness*
s	::=		*Statement*
		`borrow` *lval* t `as` ⟨α⟩ x `{` s `}`	*borrow*
p, q	::=		*Owners*
		$\texttt{unique}_{\texttt{p}}$	*unique*

We now present the static semantics of our extended system.

Expressions

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

With unique values in the system, it is no longer possible to treat all l-values directly as l-values as subsumption would allow them to be viewed as being of the corresponding non-unique type. This would allow method calls and field accesses on unique references, which would break external uniqueness. The rules (expr-lval) and (expr-dread) correspond to extracting the value within the l-value. If the type is non-unique, then (the contents of) an l-value can automatically be used as a value. If the type is unique, then a destructive read must be used to convert its contents into an expression. Destructive reads can also safely apply to non-unique l-values.

$$\frac{\mathsf{E} \vdash \mathtt{p} : \mathtt{c}\langle\sigma\rangle}{\mathsf{E} \vdash \mathtt{new}\ \mathtt{p} : \mathtt{c}\langle\sigma\rangle :: \mathtt{unique}_{\mathtt{p}} : \mathtt{c}\langle\sigma\rangle} \text{(EXPR-NEW)}$$

The modified (EXPR-NEW) contains a subtle detail: instantiating a class creates an externally unique object and the owner of the non-unique type becomes the movement bound.

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

$$\frac{\mathsf{E} \vdash \mathtt{unique}_{\mathtt{p}} : \mathtt{c}\langle\sigma\rangle}{\mathsf{E} \vdash \mathtt{unique}_{\mathtt{p}} : \mathtt{c}\langle\sigma\rangle \leqslant \mathtt{p} : \mathtt{c}\langle\sigma\rangle} \text{(SUB-UNIQUE)} \qquad \frac{\mathsf{E} \vdash \mathtt{q} \prec^* \mathtt{p}}{\mathsf{E} \vdash \mathtt{unique}_{\mathtt{p}} : \mathtt{c}\langle\sigma\rangle \leqslant \mathtt{unique}_{\mathtt{q}} : \mathtt{c}\langle\sigma\rangle} \text{(SUB-MOVE)}$$

Such rules would, however, allow the implicit conversion of objects from unique to non-unique type. This would have to be taken into consideration at many points in the formalism, complicating it furhter. Rather, we require conversion to be explicit:

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

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

Statements

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

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

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

6.5.2 Dynamic Semantics

Store Type

Possible types in Γ are extended with two construct that are very much like the scoped region: *uniqueness wrappers* and *borrowing blocks*. The uniqueness wrapper encapsulates the type information for all objects in a unique aggregate. The borrowing block does the same, but corresponds to a borrowed unique. The syntax for store-typing included $\mathfrak{n} :: \mathsf{T}[\Gamma]$, where $\mathsf{T} ::= \mathsf{c}\langle\sigma\rangle \mid \mathfrak{R}$. The complete extended T is now:

Extended syntax		terms:
$\mathsf{T} ::=$		*Owner-less type*
	$\mathfrak{U}$	*unique wrapper*
	$\mathfrak{B}$	*borrowing block*

The $\oplus$ operator is extended in a straightforward fashion to work for borrowing blocks in the store-type and on the stack (it works just like for the region).

We also extend the rules for well-formed store type introduced in Section 3.3.2 on page 51 with rules for extending a well-formed store-type with an empty uniqueness wrapper or an empty borrowing block, in both cases in some subheap $\mathfrak{m}$.

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

By (store-type-unique) and (store-type-borrow), a uniqueness wrapper or borrowing block $\mathfrak{n}$ in the subheap of some object (or unique, region or borrowing block) $\mathfrak{m}$ is well-formed if $\mathfrak{m}$ is a good owner (that is, $\mathfrak{m}$ is defined in Γ) and $\mathfrak{n}$ is not used in Γ. Borrowing blocks can also be pushed onto the store-type, much like regions. We extend the (store-type-regions) construct accordingly to govern pushing an empty borrowing block to the top of the stack.

$$\frac{\Gamma \vdash \diamond \quad \mathfrak{n} \notin \mathsf{defs}(\Gamma) \quad \mathsf{T} \in \{\mathfrak{R}, \mathfrak{B}\}}{\Gamma \oplus \mathfrak{n} :: \mathsf{T} \vdash \diamond}\ (\textsc{store-type-region})$$

As we can see, there are two rules for inserting a $\mathfrak{n} :: \mathfrak{B}$ into Γ. The first one applies to adding the borrowing block into its movement bound and the second applies to pushing the block onto the top of the stack.

Stacks and Frames

The syntactical categories frames and values are extended with a borrowing block and a unique value respectively.

$\mathsf{F} ::=$		*frame*
	$\mathsf{B}^{\mathfrak{b}}_{\mathfrak{n}}[\mathsf{H}; \mathsf{F}]$	*borrowing block*

$v ::=$		*value*
	$\mathsf{U}_{\mathfrak{n}}[v; \mathsf{H}]$	*unique*

The borrowing block works just like a region, but has an additional piece of information, here $\mathfrak{b}$, corresponding to the movement bound of the unique before it was borrowed. This is crucial in showing that reinstatement of a borrowed value produces a valid unique, though it has no effect on computation.

A unique value has a pointer compartment and a subheap compartment. Its identity, here n, corresponds to the field or variable owner.

The operation $\oplus$, and the helper function defs, and field and variable look-up functions for S and Γ are extended with cases to deal with the borrowing blocks, exactly as for regions.

Configurations

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

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

Variables and Frames

$$\frac{\Gamma \vdash F \gg \Gamma' \quad \Gamma \vdash_{x} v :: t}{\Gamma \vdash F \oplus x \mapsto v \gg \Gamma' \oplus x :: t} \text{ (VARIABLES)}$$

The (VARIABLES) judgement is extended with an owner subscript to capture that the identity of the value v, if unique, must be x. Technically, this judgement passes the information which is used in the value judgements.

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

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

Values

For keeping the identity of a uniqueness wrapper in sync with its variable or field, we slightly modify the judgement for good values.

$\Gamma \vdash_{\boxed{n}} v :: t$ Value v has type t in Γ; furthermore if v is unique, its owner is n

The judgement is extended with an owner subscript that captures the intended owner of the unique. For example, in the proof tree for the typing of a unique variable x containing value v and type t, will include the good value judgement:

$$\Gamma \vdash_x v :: t$$

This requires that v is either *null*, which is a legitimate unique, or a unique value with identity x: $U_x[\cdots]$. The chaining of the unique variable's name through the proof tree is they key here. If x is a unique variable, then x is the owner of any unique stored in the variable, which is illustratated by the following proof tree:

$$\dfrac{\dfrac{\cdots}{\Gamma \vdash F \gg \Gamma_1} \qquad \dfrac{\dfrac{}{\Gamma_3 = \Gamma\langle x :: \mathfrak{U}[\Gamma_2]\rangle} \quad \dfrac{\cdots}{\Gamma_3; n \vdash H \gg \Gamma_2} \quad \cdots}{\Gamma \vdash_x U_x[v; H] :: \mathtt{unique}_p : c\langle\sigma\rangle}}{\Gamma \vdash F \oplus x \mapsto U_x[v; H] \gg \Gamma_1 \oplus x :: \mathtt{unique}_p : c\langle\sigma\rangle}$$

$$\dfrac{\Gamma \vdash t \quad \Gamma(m) = t}{\Gamma \vdash_{\boxed{n}} \uparrow m :: t}\ (\textsc{val-pointer}) \qquad \dfrac{\Gamma \vdash t}{\Gamma \vdash_{\boxed{n}} \mathit{null} :: t}\ (\textsc{val-null}) \qquad \dfrac{\Gamma \vdash_{\boxed{n}} v :: t' \quad \Gamma \vdash t' \leqslant t}{\Gamma \vdash_{\boxed{n}} v :: t}\ (\textsc{val-subsumption})$$

For non-uniques and the special *null* value in (VAL-POINTER) and (VAL-NULL) respectively, the subscript on the turnstile is ignored. For (VAL-SUBSUMPTION), the subscript is simply "propagated".

$$\dfrac{\Gamma'' = \Gamma\langle n :: \mathfrak{U}[\Gamma']\rangle_b \quad \Gamma \vdash \mathtt{unique}_b : c\langle\sigma\rangle \quad \Gamma'' \vdash_n \uparrow m :: n : c\langle\sigma\rangle \quad \Gamma''; n \vdash H \gg \Gamma'}{\Gamma \vdash_n U_n[\uparrow m; H] :: \mathtt{unique}_b : c\langle\sigma\rangle}\ (\textsc{val-unique})$$

A unique value is well-formed if its pointer compartment and nested subheap are well-formed under an extended store type where the *type information from the unique is added in*. The bound $\mathfrak{b}$ of the unique determines where the type information of the unique is inserted in the store type as for the borrowing block. Also, the unique type must be well-formed under the original store-type where the unique's contents are not visible. Note how the subscript on the turnstile is put to use.

To reflect the visibility of uniques in our system, we use a slightly unorthodox formalisation. Just as only the store-typing for itself, previous frames, and not the entire store type, is visible to a stack frame, the type information for uniques is only visible inside the uniques themselves—the extension of Γ into Γ'' is only "visible locally", inside (VAL-UNIQUE). This enables us to formulate a nice theorem for external-uniqueness-as-dominating-edges without having to consider that parts of the store-typing change when uniques are moved, and also fits the way we think about uniques.

In (VAL-UNIQUE), it may look as if we are "pulling type information out of nowhere", but this is deceiving; Γ' must be parallel to H. Otherwise, $\Gamma''; n \vdash H \gg \Gamma'$ would not hold.

Heaps

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

Just as in (VARIABLES), (OBJECT) is extended with a subscript on the turnstile to capture the owner of the object.

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

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

Operational Semantics

This section presents the operational semantics for the new and extended constructs.

$$\frac{\langle S \mid e\rangle \rightarrow \langle S' \mid U_{\text{free}}[\uparrow n; H]\rangle}{\langle S \mid t\ x := e\rangle \rightarrow \langle S' \oplus x \mapsto U_x[\uparrow n; H[x/\text{free}]]\rangle}\ (\textsc{stat-local-2})$$

(STAT-LOCAL-2) extends variable declaration with an additional case that deals only with uniques as the original (STAT-LOCAL) is defined for $v ::= \uparrow n \mid \mathit{null}$. The owner of the unique variable is substituted for the variable name on assignment.

$$\frac{\langle S \mid e\rangle \rightarrow \langle S' \mid U_{\text{free}}[\uparrow n; H]\rangle}{\langle S \mid x := e\rangle \rightarrow \langle S'[x \mapsto U_x[\uparrow n; H[x/\text{free}]]]\rangle}\ (\textsc{stat-update-2})$$

(STAT-UPDATE-2) extends variable update with an additional case that deals only with uniques as the original (STAT-UPDATE) is defined for $v ::= \uparrow n \mid \mathit{null}$. The owner of the unique variable is substituted for the variable name on assignment.

$$\frac{\langle S \mid e\rangle \rightarrow \langle S' \mid U_{\text{free}}[\uparrow m; H]\rangle \qquad S'(x) = \uparrow n \qquad S'(n) = o}{\langle S \mid x.f := e\rangle \rightarrow \langle (S')_{n.f} := U_{n.f}[\uparrow m; H[n.f/\text{free}]]\rangle}\ (\textsc{update-field-2})$$

(UPDATE-FIELD-2) is extended with an additional case that deals only with uniques as the original (UPDATE-FIELD) is defined for $v ::= \uparrow n \mid \mathit{null}$.

$$\frac{S(x) = v}{\langle S \mid x\text{--}\rangle \rightarrow \langle S[x \mapsto \mathit{null}] \mid v[\text{free}/x]\rangle}\ (\textsc{expr-dread-local})$$

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

Unique local variables and fields must be read using the destructive read operation. The operational semantics for the destructive reads is similar to (EXPR-FIELD) and (EXPR-

LOCAL), except that the field or variable is updated with null. The unique value is also given the owner *free*, which corresponds to it being a free value. This is denoted by the substitution of x or $n.f$ for free.

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

$$\frac{\langle S \mid e\rangle \to \langle S' \mid U_{\text{free}}[v; H]\rangle}{\langle S \mid (p)\ e\rangle \to \langle S'\langle H[p/\text{free}]\rangle_p \mid v\rangle}\ (\text{EXPR-LOSE-UNIQUENESS}) \qquad \frac{\langle S \mid e\rangle \to \langle S' \mid \mathit{null}\rangle}{\langle S \mid (p)\ e\rangle \to \langle S' \mid \mathit{null}\rangle}\ (\text{EXPR-LOSE-UNIQUENESS2})$$

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

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

Finally, (STAT-BORROW) show the operational semantics for our borrowing operation.

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

Before presenting the subject reduction proof for the additional constructs, as well as the proof that externally unique pointers are dominating edges, we first show a few necessary lemmas.

6.5.3 Lemmas for Unique and Borrowing Pointers

Lemma 6.5.1 (Ignore Id)**.** *If* $\Gamma \vdash v :: p : c\langle\sigma\rangle$ *then* $\Gamma \vdash_* v :: p : c\langle\sigma\rangle$ *where* $* \in \{n.f, x\}$ *for any* n, f *and* x.

Proof. Follows immediately from the definition of the rules (VAL-$*$) since the subscript on the turnstile is ignored for all non-unique values. □

Lemma 6.5.2 (Movement)**.** *Let* $\mathfrak{I}$ *be a possible right-hand side of the typing judgements where* $\mathfrak{I} \neq U_n[v; H] :: t$ *in Case 1) (also ignore "*$; q,$*" where not applicable):*

1. $\Gamma\langle n :: \mathfrak{U}[\Gamma']\rangle_p; q \vdash \mathfrak{I}$ *iff* $\Gamma\langle n :: \mathfrak{B}[\Gamma']\rangle_p; q \vdash \mathfrak{I}$. *(Borrow/Reinstate)*

If $\Gamma\langle n :: \mathfrak{U}[\Gamma']\rangle_p; q \vdash \mathfrak{I}$*, then:*

2. $\Gamma\langle m :: \mathfrak{U}[\Gamma'[m/n]]\rangle_p; q[m/n] \vdash \mathfrak{I}[m/n]$ *where* m *is fresh.* *(Move)*
3. *If* $\Gamma \vdash m \prec^* p$*, then* $\Gamma\langle n :: \mathfrak{U}[\Gamma']\rangle_m; q \vdash \mathfrak{I}$. *(Tighten movement bound)*
4. $\Gamma\langle\Gamma'[p/n]\rangle_p; q \vdash \mathfrak{I}[p/n]$. *(Lose uniqueness)*

Proof.

Case 1) Follows immediately from the well-formedness rules.

Case 2) The proof of this fact is straightforward but tedious and therefore omitted. As nothing in Γ is dependent on n, the substitution is basically a global renaming.

Case 3) Proof by induction over the shape of $\mathfrak{I}$. It relies on the following fact:

> If $\Gamma\langle n :: T[\Gamma']\rangle_p \vdash p_1 \mathrel{R} p_2$ and $\Gamma \vdash q \prec^* p$, then $\Gamma\langle n :: T[\Gamma']\rangle_q \vdash p_1 \mathrel{R} p_2$.

which follows immediately from the observation that as $q \prec^* p$, the set of derivable owner relations when $n :: T[\Gamma']$ is directly inside q is a superset of the derivable relations when $n :: T[\Gamma']$ is directly inside p.

Case 4) Similar to cases two and three in the case above. The owner relationships visible at q is a superset of those visible at p, and, as clearly n is directly inside p and $\Gamma \vdash q \prec^* p$, q satisfies at least the same owner relations as n.

□

Lemma 6.5.3 (Move Unique). *If* $\Gamma \vdash_n \mathsf{U}_n[v; H] :: \mathtt{unique}_p : c\langle\sigma\rangle$ *and* $\Gamma \vdash q \prec^* p$ *where* m *is fresh, then* $\Gamma \vdash_m \mathsf{U}_m[v; H[m/n]] :: \mathtt{unique}_q : c\langle\sigma\rangle$.

Proof. By (VAL-UNIQUE), $\Gamma \vdash \mathtt{unique}_p : c\langle\sigma\rangle$ and $n \notin \mathsf{defs}(\Gamma)$. Therefore, clearly $n \notin \mathit{rng}(\sigma) \cup \{p, q\}$. The result thus follows immediately from Lemma 6.5.2. □

Visibility Lemma

The visibility lemma indirectly deals with what parts of the store are visible from another. The well-formedness of a type is not dependent on any possible siblings. First, we define a disjuction operator on store types thus: $\Gamma \# \Gamma' \iff \mathsf{defs}(\Gamma) \cap \mathsf{defs}(\Gamma') = \emptyset$.

Lemma 6.5.4 (Visibility). *If* $\Gamma \vdash \diamond$, *then* $\Gamma\langle\Gamma', \Gamma''\rangle_n \vdash \diamond$ *iff* $\Gamma\langle\Gamma'\rangle_n \vdash \diamond$, $\Gamma\langle\Gamma''\rangle_n \vdash \diamond$ *and* $\Gamma' \# \Gamma''$.

Proof. We prove this in the following steps:

1. If $\Gamma \vdash \diamond$, $\Gamma\langle\Gamma'\rangle_n \vdash \diamond$, $\Gamma\langle\Gamma''\rangle_m \vdash \diamond$ and $\Gamma' \# \Gamma''$, then $(\Gamma\langle\Gamma'\rangle_n)\langle\Gamma''\rangle_m \vdash \diamond$ (Note: $(\Gamma\langle\Gamma'\rangle_n)\langle\Gamma''\rangle_m = \Gamma\langle\Gamma', \Gamma''\rangle_n$ when $n = m$).
2. If $\Gamma\langle\Gamma', \Gamma''\rangle_n \vdash \diamond$ and $\Gamma \vdash \diamond$, then $\Gamma\langle\Gamma'\rangle_n \vdash \diamond$.
3. If $\Gamma\langle\Gamma', \Gamma''\rangle_n \vdash \diamond$ then $\Gamma' \# \Gamma''$.

Step 1 It is sufficient to prove a simpler merge involving only one object. The more general result follows by induction on the size of Γ''.

If $\Gamma \vdash \diamond$, $\Gamma\langle\Gamma'\rangle_p \vdash \diamond$, $\Gamma\langle n :: T\rangle_m \vdash \diamond$, and $n \notin \mathsf{defs}(\Gamma')$, then $(\Gamma\langle\Gamma'\rangle_p)\langle n :: T\rangle_m \vdash \diamond$.

There are four cases: (a) $T = c\langle\sigma\rangle$, (b) $T = \mathfrak{U}$, (c) $T = \mathfrak{R}$ and (d) $T = \mathfrak{B}$.

Case a) By (STORE-TYPE-OBJECT), $\Gamma \vdash m : c\langle\sigma\rangle$ and $m \notin \mathsf{defs}(\Gamma)$. By Lemma (Extension), $\Gamma\langle\Gamma'\rangle_p \vdash m : c\langle\sigma\rangle$. By (STORE-TYPE-OBJECT), $(\Gamma\langle\Gamma'\rangle_p)\langle m :: c\langle\sigma\rangle\rangle_n \vdash \diamond$.

Case b) By (STORE-TYPE-UNIQUE), $\Gamma \vdash m$ and $m \notin \mathsf{defs}(\Gamma)$. By Lemma (Well-formed construction), $\Gamma\langle\Gamma'\rangle_p \vdash \diamond$. By Lemma (Extension), $\Gamma\langle\Gamma'\rangle_p \vdash m$. By (STORE-TYPE-OBJECT), $(\Gamma\langle\Gamma'\rangle_p)\langle m :: \mathfrak{U}\rangle_n \vdash \diamond$.

Case c) Case c) is similar to case b) and is therefore omitted.

Case d) Case d) is similar to case b) and is therefore omitted.

Step 2 Proof by contradiction: assume $\Gamma\langle\Gamma', \Gamma''\rangle_n \vdash \diamond$, and $\Gamma \vdash \diamond$ and *not* $\Gamma\langle\Gamma'\rangle_n \vdash \diamond$.

If *not* $\Gamma\langle\Gamma'\rangle_n \vdash \diamond$, then either (a) some type t in Γ is ill-formed (that is not $\Gamma \vdash t$) or (b) an object id in Γ' already exists in Γ.

In case of b), this would contradict the well-formedness or $\Gamma\langle\Gamma', \Gamma''\rangle_n \vdash \diamond$. In case of a), there exists an object o in Γ' of type t s.t. some relation in its type is not satisfied by the owners in in $\Gamma\langle\Gamma'\rangle_n$. As clearly $\Gamma\langle\Gamma', \Gamma''\rangle_n \vdash t$ (otherwise, not $\Gamma\langle\Gamma', \Gamma'\rangle \vdash \diamond$, which was an assumption), the only case where t can be ill-formed under $\Gamma\langle\Gamma'\rangle_n$ is if t uses an owner who's ordering is dependent on some owner in Γ'' *i.e.,* $\Gamma\langle\Gamma', \Gamma''\rangle_n \vdash p \prec^* q$ where $p \in \mathsf{owners}(t)$ and $q \in \mathsf{defs}(\Gamma'')$. By (TYPE) and (CLASS) $p \in \mathsf{owners}(t)$ implies o is inside p. Clearly n is outside o and so, either p must be outside n or be defined somewhere in Γ'. For q to satisfy any of these conditions, it must either be defined in Γ' or outside n, which contradicts the requirement that all locations must be fresh when introduced (see also Subproof 3) and thus contradict the well-formeness of $\Gamma\langle\Gamma', \Gamma''\rangle_n$. Therefore, $\Gamma\langle\Gamma'\rangle_n \vdash \diamond$.

Step 3 Assume $\Gamma\langle\Gamma', \Gamma''\rangle_n \vdash \diamond$ and *not* $\Gamma' \# \Gamma''$. Then, there exists $n \in \mathsf{defs}(\Gamma') \cap \mathsf{defs}(\Gamma'')$, which contradicts the requirement that all locations must be fresh when introduced. Therefore, $\Gamma' \# \Gamma''$.

□

Borrowing Lemmas

The subsequent lemmas deal with borrowing and reinstating unique variables.

Lemma 6.5.5 (Borrow)**.** *If* $\Gamma \vdash S \oplus x \mapsto U_x[v; H]$ *and* $\Gamma \vdash x :: \mathtt{unique}_b : c\langle\sigma\rangle$ ***ref,*** *then* $\Gamma \oplus n :: \mathfrak{B}[\Gamma' \oplus \alpha \mapsto n \oplus y :: n : c\langle\sigma\rangle] \vdash S \oplus x \mapsto \mathit{null} \oplus B^b_n[H[n/x]; \alpha \mapsto n \oplus y \mapsto v]$*, where* n, y, α *are fresh.*

Proof. Assume $\Gamma \vdash S \oplus x \mapsto U_x[v; H]$ and $\Gamma \vdash x :: \texttt{unique}_b : c\langle\sigma\rangle$ **ref**.

1. By definition of variable look-up in stack, $(S \oplus x \mapsto U_x[v; H])(x) = U_x[v; H]$.
2. By 1.), and Lemma 3.4.4 (Variable Look-up in Stack), $\Gamma \vdash_x U_x[v; H] :: \texttt{unique}_b : c\langle\sigma\rangle$.
3. By 2.), Lemma 3.4.2 (Well-formed construction), and (VAL-NULL), $\Gamma \vdash_x \mathit{null} :: \texttt{unique}_b : c\langle\sigma\rangle$.
4. By 3.) and Lemma 3.4.9 (Variable Update in Stack), $\Gamma \vdash S \oplus x \mapsto \mathit{null}$.
5. By 3.) and Lemma 3.4.2 (Well-formed construction),
   (a) $\Gamma \vdash b$ and
   (b) $\Gamma \vdash \diamond$.
6. By 5.a,b) and (STORE-TYPE-BORROW), $\Gamma\langle n :: \mathfrak{B}\rangle_b \vdash \diamond$.
7. By 6.) and (IN-OWNER), $\Gamma\langle n :: \mathfrak{B}\rangle_b \vdash n \prec^* b$.
8. By 2.) and (VAL-UNIQUE),
   (a) $\Gamma\langle x :: \mathfrak{U}[\Gamma_1]\rangle_b \vdash_x v :: x : c\langle\sigma\rangle$,
   (b) $\Gamma\langle x :: \mathfrak{U}[\Gamma_1]\rangle_b; x \vdash H \gg \Gamma_x$ and
   (c) $\Gamma \vdash \texttt{unique}_b : c\langle\sigma\rangle$.
9. By 8.c), Lemma 3.4.2 (Well-formed construction) and (GOOD-OWNER), $\mathsf{owners}(\texttt{unique}_b : c\langle\sigma\rangle) \in \mathsf{defs}(\Gamma)$, *i.e.*, $x \notin \mathsf{owners}(\texttt{unique}_b : c\langle\sigma\rangle)$.
10. By 7.), 8.a), 9.), (IN-REFL) and Lemma 6.5.2
    (a) $\Gamma\langle n :: \mathfrak{B}[\Gamma_2]\rangle_b \vdash_n v :: n : c\langle\sigma\rangle$ where
    (b) $\Gamma_2 = \Gamma_1[n/x]$.
11. By 7.), 8.b), (IN-REFL) and Lemma 6.5.2, $\Gamma\langle n :: \mathfrak{B}[\Gamma_2]\rangle_b; n \vdash H[n/x] \gg \Gamma_2$.
12. By 10.), (FRAME-EMPTY), (VARIABLES) and Lemma 6.5.1 (Omit qualifiers), $\Gamma\langle n :: \mathfrak{B}[\Gamma_2]\rangle_b \vdash y \mapsto v \gg y :: n : c\langle\sigma\rangle$.
13. By 10.a) and (GOOD-OWNER), $\Gamma\langle n :: \mathfrak{B}[\Gamma_2]\rangle_b \vdash n$.

14. By 14.) and (STORE-TYPE-OWNER), $\Gamma\langle n :: \mathfrak{B}[\Gamma_2 \oplus \alpha \mapsto n]\rangle_b \vdash \diamond$.
15. By 12.), Lemma 3.4.2 (Well-formed construction) and (STORE-TYPE-VAR), $\Gamma\langle n :: \mathfrak{B}[\Gamma_2 \oplus \alpha \mapsto n \oplus y :: n{:}c\langle\sigma\rangle]\rangle_b \vdash \diamond$.
16. By 4.), 11.), 12.), 15.), (STACK-GEN), (FRAME-BORROW) and Lemma 3.4.1 (Extension),
    (a) $\Gamma \oplus n :: \mathfrak{B}[\Gamma_3] \vdash S \oplus x \mapsto \mathit{null} \oplus B^b_n[H[n/x]; y \mapsto v] \gg \Gamma \oplus n :: \mathfrak{B}[\Gamma_3]$ where
    (b) $\Gamma_3 = \Gamma_2 \oplus \alpha \mapsto n \oplus y :: n{:}c\langle\sigma\rangle$.

□

Lemma 6.5.6 (Reinstate). *If* $\Gamma \oplus n :: \mathfrak{B}[\Gamma'] \vdash S \oplus B^b_n[H; F']$ *and* $\Gamma\langle n :: \mathfrak{B}[\Gamma']\rangle_b \vdash v :: n{:}c\langle\sigma\rangle$, *then* $\Gamma \vdash_m U_m[v; H[m/n]] :: \mathtt{unique}_b{:}c\langle\sigma[m/n]\rangle$, *where* m *if fresh.*

Proof. Assume $\Gamma \oplus n :: \mathfrak{B}[\Gamma'] \vdash S \oplus B^b_n[H; F']$ and $\Gamma\langle n :: \mathfrak{B}[\Gamma']\rangle_b \vdash v :: n{:}c\langle\sigma\rangle$.

1. By (STACK-GEN),
   (a) $S = S_1 \bullet F$ and $\Gamma = \Gamma_1 \bullet \Gamma_2$, s.t.
   (b) $\Gamma_1 \vdash S_1$ and
   (c) $\Gamma_1 \oplus n :: \mathfrak{B}[\Gamma'] \vdash F \oplus B^b_n[H; F'] \gg \Gamma_2 \oplus n :: \mathfrak{B}[\Gamma']$.
2. By 1.c) and (FRAME-BORROW),
   (a) $\Gamma \vdash F \gg \Gamma_2$,
   (b) $\Gamma_3; n \vdash H \gg \Gamma_4$ and
   (c) $\Gamma_3 \vdash F' \gg \Gamma_5$ where
   (d) $\Gamma_3 = \Gamma\langle n :: \mathfrak{B}[\Gamma']\rangle_b$ and
   (e) $\Gamma' = \Gamma_4, \Gamma_5$.
3. By 2.b) and Lemma 6.5.2, $\Gamma\langle m :: \mathfrak{U}[\Gamma_4[m/n]]\rangle_b; m \vdash H[m/n] \gg \Gamma_4[m/n]$.
4. By Lemma 6.5.2, $\Gamma\langle m :: \mathfrak{U}[\Gamma_4[m/n]]\rangle_b \vdash v :: m{:}c\langle\sigma[m/n]\rangle$.
5. By 3.), 4.) and (VAL-UNIQUE), $\Gamma \vdash_m U_m[v; H[m/n]] :: \mathtt{unique}_b{:}c\langle\sigma[m/n]\rangle$.

□

6.5.4 Subject Reduction Proof

Proof. By structural induction on the shapes of e and s. This is a continuation of the proof of subject reduction in Chapter 3 and only deals with the cases for the extended constructs, some of which were omitted in the original proof.

Case (EXPR-NEW) Assume $\Gamma \vdash \langle S \mid \texttt{new}\ p : c\langle\sigma\rangle\rangle :: \texttt{unique}_p : c\langle\sigma\rangle$.

1. By (CONFIG-EXPR),
   (a) $\Gamma \vdash S$, and
   (b) $\Gamma \vdash \texttt{new}\ p : c\langle\sigma\rangle :: \texttt{unique}_p : c\langle\sigma\rangle$.
2. By 1.b) and (EXPR-NEW), $\Gamma \vdash p : c\langle\sigma\rangle$.
3. By 2.) and Lemma 3.4.2 (Well-Formed Construction), $\Gamma \vdash p$.
4. By 3.) and (STORE-TYPE-UNIQUE), $\Gamma\langle \text{free} :: \mathfrak{U}\rangle_p \vdash \diamond$ (where free is fresh).
5. By 4.) and (IN-OWNER), $\Gamma\langle \text{free} :: \mathfrak{U}\rangle_p \vdash \text{free} \prec^* p$.
6. By 4.), 5.) and (TYPE), $\Gamma\langle \text{free} :: \mathfrak{U}\rangle_p \vdash \text{free} : c\langle\sigma\rangle$.
7. By 6.) and (STORE-TYPE-OBJECT), $\Gamma\langle \text{free} :: \mathfrak{U}[n :: c\langle\sigma\rangle]\rangle_p \vdash \diamond$ (where n is fresh).
8. Let $\Gamma' = \Gamma\langle \text{free} :: \mathfrak{U}[n :: c\langle\sigma\rangle]\rangle_p$. Then, by 7.) and (VAL-POINTER), $\Gamma' \vdash \uparrow n :: \text{free} : c\langle\sigma\rangle$, where
9. By 8.), Lemma 3.4.2 (Well-Formed Construction) and (HEAP-EMPTY), $\Gamma'; n \vdash \text{nil} \gg \text{nil}$.
10. By 8.) and Lemma 6.5.7 (see next page), $\Gamma' \vdash \sigma_n^{\text{free}}(t)$ for all $f :: t \in \mathcal{F}_c$.
11. By 10.), (FIELDS) and (VAL-NUL),
    (a) $\Gamma' \vdash_n V :: \sigma_n^{\text{free}}(t)$ where
    (b) $V = f \mapsto \mathit{null}$ for all $f \in \mathcal{F}_c$.
12. By 7.) and def. of type look-up, $\Gamma'(n) = \text{free} : c\langle\sigma\rangle$.
13. By 9.), 11.a), 12.) and (OBJECT), $\Gamma'; \text{free} \vdash n \mapsto c^\sigma[V; \text{nil}] \gg n :: c\langle\sigma'\rangle[\text{nil}]$.
14. By 1.b) and Lemma 3.4.2 (Well-formed expression), $\Gamma \vdash \texttt{unique}_p : c\langle\sigma\rangle$.
15. By 8.), 13.), 14.) and (VAL-UNIQUE), $\Gamma \vdash U_{\text{free}}[\uparrow n; n \mapsto c^\sigma[V; \text{nil}]] :: \texttt{unique}_p : c\langle\sigma\rangle$.

16. By 1.a), 15.) and (CONFIG-VAL),
$\Gamma \vdash \langle S \mid U_{\text{free}}[\uparrow n; n \mapsto c^{\sigma}[V; \text{nil}]]\rangle :: \texttt{unique}_{p} : c\langle\sigma\rangle$.

Lemma 6.5.7. *If* $\Gamma \vdash \uparrow n :: p : c\langle\sigma\rangle$, *then* $\Gamma \vdash \sigma^{p}_{n}(\mathcal{F}_{c}(f))$ *for all* $f \in dom(\mathcal{F}_{c})$.

Proof. There are two cases: (a) f is defined in c or (b) f is defined in class c', a superclass to c.

Case a) By (CLASS), $E \vdash \mathcal{F}_{c}(f)$ where $E = \mathcal{P}_{c}, \texttt{this} \prec^{*} \texttt{owner}$ (E is the static type environment in the class). By (STORE-TYPE-GENERATION), $\Gamma' \vdash \diamond$ where $\Gamma' = \bullet\, \sigma^{p}_{n} \oplus \texttt{this} :: t$ and $\sigma^{p}(t) = p : c\langle\sigma\rangle$. By Lemma (Well-formed construction), $\Gamma \vdash p : c\langle\sigma\rangle$. By (TYPE) and Lemma 3.4.3, clearly $\Gamma' \vdash \mathcal{P}_{c}$. By (IN-OWNER), $\Gamma \vdash n \prec^{*} p$. Thus, clearly $\Gamma' \vdash \texttt{this} \prec^{*} \texttt{owner}$. As we can see, Γ' satisfies the orderings in the static type environment E, and therefore $\Gamma' \vdash \mathcal{F}_{c}(f)$. By Lemma 3.4.3, $\Gamma \vdash \sigma^{p}_{n}(\mathcal{F}_{c}(f))$.

Case b) Let c_1 be a direct superclass of c. Clearly, c_1 is not `Object` as it defines f. Clearly $\texttt{class}\ c \cdots \texttt{extends}\ c_1\langle\sigma'\rangle \cdots \in P$. By def. of variable look-up, $\mathcal{F}_{c}(f) = \sigma'(\mathcal{F}_{c_1}(f))$. By (SUB-CLASS), $\Gamma \vdash \uparrow n :: p : c_1\langle\sigma_1\rangle$ where $\sigma_1 = \sigma \circ \sigma'$. By the induction hypothesis, $\Gamma \vdash \sigma_1{}^{p}_{n}(\mathcal{F}_{c_1}(f))$, which is equivalent to $\Gamma \vdash \sigma(\sigma'^{p}_{n}(\mathcal{F}_{c_1}(f)))$. By Lemma 3.4.7, this is equivalent to $\Gamma \vdash \sigma^{p}_{n}(\mathcal{F}_{c}(f))$.

□

Case (STAT-LOCAL-2) Assume $\Gamma \oplus x :: t \vdash \langle S \mid t\, x = e\rangle$.

1. By (CONFIG-STAT),
   (a) $\Gamma \vdash S$, and
   (b) $\Gamma \vdash t\, x = e$; $\Gamma \oplus x :: t$.
2. By 1.b) and (STAT-LOCAL),
   (a) $x \notin \mathsf{vars}(\Gamma)$ and
   (b) $\Gamma \vdash e :: t$.
3. By 1.a), 2.b) and (CONFIG-EXPR), $\Gamma \vdash \langle S \mid e\rangle :: t$.
4. By 3.) and the induction hypothesis, if $\langle S \mid e\rangle \rightarrow \langle S' \mid U_{\text{free}}[\uparrow n; H]\rangle$, then there exists a Γ' s.t.

   (a) $\Gamma \rightsquigarrow \Gamma'$ and
   (b) $\Gamma' \vdash \langle S' \mid U_{\text{free}}[\uparrow n; H] \rangle :: t$.

5. By 4.b) and (CONFIG-VAL),
   (a) $\Gamma' \vdash S'$, and
   (b) $\Gamma' \vdash_{\text{free}} U_{\text{free}}[\uparrow n; H] :: t$.

6. By 5.a) and (STACK-GEN),
   (a) $\Gamma_1 \vdash S''$ and
   (b) $\Gamma' \vdash F \gg \Gamma_2$ where
   (c) $\Gamma' = \Gamma_1 \bullet \Gamma_2$.

7. By 5.b) and Lemma 6.5.2 (Movement), $\Gamma' \vdash_x U_x[\uparrow n; H[x/\text{free}]] :: t$.

8. By 6.b), 7.) and (VARIABLES),
   $\Gamma' \oplus x :: t \vdash F \oplus x \mapsto U_x[\uparrow n; H[x/\text{free}]] \gg \Gamma_2 \oplus x :: t$.

9. By 6.a,c), 8.) and (STACK-GEN), $\Gamma' \oplus x :: t \vdash S' \oplus x \mapsto U_x[\uparrow n; H[x/\text{free}]]$.

10. By 9.) and (CONFIG-FINAL), $\Gamma' \oplus x :: t \vdash \langle S' \oplus x \mapsto U_x[\uparrow n; H[x/\text{free}]] \rangle$.

Case (STAT-UPDATE-2) Assume $\Gamma \vdash \langle S \mid x := e \rangle$. We disregard the fact that y might contain *null*, as this case is already covered by previous proofs (*null*$[x/\text{free}] =$ *null* and destructive read obviously valid, as y already contains *null*).

1. By (CONFIG-STAT),
   (a) $\Gamma \vdash S$ and
   (b) $\Gamma \vdash x := e ; \Gamma$.

2. By 1.b) and (STAT-UPDATE),
   (a) $\Gamma \vdash x :: t$ **ref** and
   (b) $\Gamma \vdash e :: t$.

3. By 1.a), 2.b) and (CONFIG-EXPR), $\Gamma \vdash \langle S \mid e \rangle :: t$.

4. By 3.) and the induction hypothesis, if $\langle S \mid e \rangle \rightarrow \langle S' \mid v \rangle$, there exists a Γ' such that
   (a) $\Gamma \rightsquigarrow \Gamma'$ and
   (b) $\Gamma' \vdash \langle S' \mid v \rangle :: t$.

5. By 4.b) and (CONFIG-VAL),

   (a) $\Gamma' \vdash S'$ and
   (b) $\Gamma' \vdash_{\text{free}} v :: t$.
6. By 2.a), 4.a) and Lemma 3.4.1 (Extension), $\Gamma' \vdash x :: t$ **ref**.
7. By 5.b), if $\mathsf{isunique}(t)$, then $x \notin \mathsf{defs}(\Gamma')$ as this would contradict the unique names assumption. Thus, by Lemma 6.5.3, $\Gamma' \vdash_x v' :: t$ where $v' = v[x/\text{free}]$. If $\neg\mathsf{isunique}(t)$, then by Lemma 6.5.1 (Omit qualifiers), $\Gamma' \vdash_x v' :: t$ where $v' = v$.
8. By 5.a), 6.), 7.) and Lemma 3.4.9 (Variable Update), $\Gamma' \vdash S'[x \mapsto v']$.
9. By 8.) and (CONFIG-FINAL), $\Gamma' \vdash \langle S'[x \mapsto v'] \rangle$.

Case (UPDATE-FIELD-2) Assume $\Gamma \vdash \langle S \mid x.f := y\text{--}\rangle$.

1. By (CONFIG-STAT),
   (a) $\Gamma \vdash S$, and
   (b) $\Gamma \vdash x.f := e; \Gamma$.
2. By 1.b) and (STAT-UPDATE),
   (a) $\Gamma \vdash x.f :: t$ **ref** and
   (b) $\Gamma \vdash e :: t$.
3. By 1.a), 2.b) and (CONFIG-EXPR), $\Gamma \vdash \langle S \mid e \rangle :: t$.
4. By 3.) and induction hypothesis, if $\langle S \mid e \rangle \to \langle S' \mid U_{\text{free}}[\uparrow m; H] \rangle$, then there exists, Γ' s.t.,
   (a) $\Gamma' \vdash \langle S' \mid U_{\text{free}}[\uparrow m; H] \rangle :: t$ and
   (b) $\Gamma \rightsquigarrow \Gamma'$.
5. By 4.a) and (CONFIG-VAL),
   (a) $\Gamma' \vdash S'$ and
   (b) $\Gamma' \vdash_{\text{free}} U_{\text{free}}[\uparrow m; H] :: t$.
6. By 2.a), Lemma 3.4.1 (Extension) and (LVAL-FIELD),
   (a) $\Gamma' \vdash x :: p\,c\langle\sigma\rangle$,
   (b) $\sigma^p(\mathcal{F}_c(f)) = t$ and
   (c) $\texttt{this} \in \mathsf{owners}(\mathcal{F}_c(f)) \Rightarrow x \equiv \texttt{this}$.
7. By 5.a), 6.a) and Lemma 3.4.4 (Variable Look-up), $\Gamma' \vdash \uparrow n :: p\,c\langle\sigma\rangle$.

8. By 5.b.) and Lemma 6.5.3 (Move unique), $\Gamma' \vdash_{n.f} U_{n.f}[\uparrow m; H[n.f/\text{free}]] :: t$. ($n.f$ is not in Γ as it would otherwise contradict the unique names assumption.)
9. By 5.a), 7.), 8.) and Lemma 3.4.11 (Field Update), $\Gamma' \vdash (S')_{n.f} := U_{n.f}[\uparrow m; H[n.f/\text{free}]]$.
10. By 10.) and (CONFIG), $\Gamma' \vdash \langle (S')_{n.f} := U_{n.f}[\uparrow m; H[n.f/\text{free}]] \rangle$.

Case (EXPR-DREAD-FIELD) Assume $\Gamma \vdash \langle S \mid x.f\text{--} \rangle :: t$.

1. By (CONFIG-EXPR),
   (a) $\Gamma \vdash S$, and
   (b) $\Gamma \vdash x.f\text{--} :: t$.
2. By (EXPR-DREAD), $\Gamma \vdash x.f :: t$.
3. By 2.) and (LVAL-FIELD),
   (a) $\Gamma \vdash x :: p : c\langle\sigma\rangle$,
   (b) $t = \sigma^p(\mathcal{F}_c(f))$, and
   (c) $\texttt{this} \in \mathsf{owners}(\mathcal{F}_c(f)) \Rightarrow x \equiv \texttt{this}$.
4. By 1.a), 3.a-c), and Lemma 3.4.6 (Field Look-up), $\Gamma \vdash_{n.f} :: v :: t$ where $S(x) = \uparrow n$.
5. By 4.) and (VAL-*), either (a) $v = \mathit{null}$ or (b) $v = U_{n.f}[v'; H]$. In case (a), by Lemma 6.5.1 (Omit Qualifiers), $\Gamma \vdash_{\text{free}} v[\text{free}/n.f] :: t$ as $v[\text{free}/n.f] = v$. In case (b), by 4.) and Lemma 6.5.2 (Movement), $\Gamma \vdash_{\text{free}} v[\text{free}/n.f] :: t$.
6. By 1.b), Lemma 3.4.2 (Well-Formed Construction) and (VAL-NULL), $\Gamma \vdash_{n.f} \mathit{null} :: t$.
7. By 1.a), 3.a-c), 6.) and Lemma 3.4.11 (Field Update), $\Gamma \vdash (S)_{n.f} := \mathit{null}$.
8. By 5.), 7.) and (CONFIG-VAL), $\Gamma \vdash \langle (S)_{n.f} := \mathit{null} \mid v[\text{free}/n.f] \rangle :: t$.

Case (EXPR-DREAD-LOCAL) Assume $\Gamma \vdash \langle S \mid x\text{--} \rangle :: \texttt{unique}_b : c\langle\sigma\rangle$.

1. By (CONFIG-EXPR),
   (a) $\Gamma \vdash S$ and
   (b) $\Gamma \vdash x\text{--} :: t$.
2. By 1.b) and (EXPR-DREAD), $\Gamma \vdash x :: t$ **ref**.

3. By 1.a), 2.) and Lemma 3.4.4 (Variable Look-up), $\Gamma \vdash_x v :: t$.
4. By 1.b), Lemma 3.4.2 (Well-Formed Construction) and (VAL-NULL), $\Gamma \vdash_x \mathit{null} :: t$.
5. By 1.a), 2.), 4.) and Lemma 3.4.9 (Variable Update), $\Gamma \vdash S[x \mapsto \mathit{null}]$.
6. By 3.) and (VAL-*) either (a) $v = \mathit{null}$ or (b) $\Gamma \vdash_x U_x[v'; H] :: t$. In the case (a) $v[\text{free}/x] = v$ and, thus, by Lemma 6.5.1 (Omit Qualifiers), $\Gamma \vdash_{\text{free}} v[\text{free}/x] :: t$. In the case (b), by 3.) and Lemma 6.5.2 (Movement), $\Gamma \vdash_{\text{free}} v[\text{free}/x] :: t$.
7. By 5.), 6.) and (CONFIG-VAL), $\Gamma \vdash \langle S[x \mapsto \mathit{null}] \mid v[\text{free}/x] \rangle :: t$.

Case (EXPR-LOSE-UNIQUENESS) For simplicity, we present the proof of (EXPR-LOSE-UNIQUENESS) in two separate installements: one where e evaluates to a unique and one where it evaluates to *null*.

Assume $\Gamma \vdash \langle S \mid (p)\ e \rangle :: p : c\langle\sigma\rangle$.

1. By (CONFIG-EXPR),
   (a) $\Gamma \vdash S$ and
   (b) $\Gamma \vdash (p)\ e :: p : c\langle\sigma\rangle$.
2. By 1.b) and (MOVE-UNIQUE),
   (a) $\Gamma \vdash e :: \texttt{unique}_b : c\langle\sigma\rangle$ and
   (b) $\Gamma \vdash p \prec^* b$.
3. By 1.a), 2.a) and (CONFIG-EXPR), $\Gamma \vdash \langle S \mid e \rangle :: \texttt{unique}_b : c\langle\sigma\rangle$.
4. By 3.) and induction hypothesis, if $\langle S \mid e \rangle \to \langle S' \mid v \rangle$, then there exists Γ' s.t.
   (a) $\Gamma \rightsquigarrow \Gamma'$ and
   (b) $\Gamma' \vdash \langle S' \mid v \rangle :: \texttt{unique}_b : c\langle\sigma\rangle$.
5. By 4.b), (CONFIG-VAL) and (VAL-UNIQUE),
   (a) $\Gamma' \vdash S'$ and
   (b) $\Gamma' \vdash_{\text{free}} v :: \texttt{unique}_b : c\langle\sigma\rangle$ where
   (c) $v = U_{\text{free}}[\uparrow n; H]$
6. By 2.a), 4.a) and Lemma 3.4.1 (Extension), $\Gamma' \vdash p \prec^* b$.
7. By 6.) and Lemma 3.4.2 (Well-formed construction), $\Gamma' \vdash p$.
8. By 7.), either

(a) $p :: \mathfrak{B}[\_] \in \Gamma'$, or

(b) $p :: \mathfrak{R}[\_] \in \Gamma'$, or

(c) $\Gamma'(p) = m{:}c_1\langle\sigma_1\rangle$.

(Note that $p :: \mathfrak{U}[\_] \in \Gamma'$ is not possible as uniques are never in the top-level store-type.)

9. By 5.b) and (VAL-UNIQUE),

   (a) $\Gamma_1; \mathrm{free} \vdash H \gg \Gamma''$,

   (b) $\Gamma_1 \vdash \uparrow n :: \mathrm{free}{:}c\langle\sigma\rangle$ and

   (c) $\Gamma \vdash \mathtt{unique}_b : c\langle\sigma\rangle$ where

   (d) $\Gamma_1 = \Gamma\langle \mathrm{free} :: \mathfrak{U}[\Gamma'']\rangle_b$.

10. By 5.a), 6.), 8.a-c), 9.a,d), and Lemma 6.5.2, (Lose Uniqueness), $\Gamma_1\langle\Gamma''[p/\mathrm{free}]\rangle \vdash S'\langle H[p/\mathrm{free}]\rangle_p$.

11. By 9.a,c,d), (STORE-TYPE-UNIQUE) and Lemma 3.4.2 (Well-formed construction), $\mathrm{free} \notin \mathit{rng}(\sigma)$.

12. By 6.), 9.b,d), 11.) and Lemma 6.5.2 (Lose Uniqueness), $\Gamma_1\langle\Gamma''[p/\mathrm{free}]\rangle \vdash \uparrow n :: p{:}c\langle\sigma\rangle$.

13. By 10.), 12.) and (CONFIG-VAL), $\Gamma_1\langle\Gamma''[p/\mathrm{free}]\rangle \vdash \langle S'\langle H[p/\mathrm{free}]\rangle_p \mid \uparrow n\rangle :: p : c\langle\sigma\rangle$.

Case (EXPR-LOSE-UNIQUENESS) Assume $\Gamma \vdash \langle S \mid (p)\ e\rangle :: p{:}c\langle\sigma\rangle$.

1. By (CONFIG-EXPR),

   (a) $\Gamma \vdash S$ and

   (b) $\Gamma \vdash (p)\ e :: p{:}c\langle\sigma\rangle$.

2. By 1.b) and (EXPR-LOSE-UNIQUENESS),

   (a) $\Gamma \vdash e :: \mathtt{unique}_q : c\langle\sigma\rangle$ and

   (b) $\Gamma \vdash p \prec^* q$.

3. By 1.a), 2.a) and (CONFIG-EXPR), $\Gamma \vdash \langle S \mid e\rangle :: \mathtt{unique}_q : c\langle\sigma\rangle$.

4. By 3.) and induction hypothesis, if $\langle S \mid e\rangle \to \langle S' \mid \mathit{null}\rangle$, then there exists a Γ' such that,

   (a) $\Gamma \rightsquigarrow \Gamma'$, and

   (b) $\Gamma' \vdash \langle S' \mid \mathit{null}\rangle :: \mathtt{unique}_q : c\langle\sigma\rangle$.

5. By 4.b) and (CONFIG-EXPR),
   (a) $\Gamma' \vdash S'$ and
   (b) $\Gamma' \vdash \mathit{null} :: \mathtt{unique}_q : c\langle\sigma\rangle$.
6. By 1.a) and Lemma 3.4.2 (Well-Formed Construction), $\Gamma \vdash p : c\langle\sigma\rangle$.
7. By 4.a), 6.) and Lemma 3.4.1 (Extension), $\Gamma' \vdash p : c\langle\sigma\rangle$.
8. By 7.) and (VAL-NULL), $\Gamma' \vdash \mathit{null} :: p : c\langle\sigma\rangle$.
9. By 5.a), 8.) and (EXPR-VAL), $\Gamma' \vdash \langle S' \mid \mathit{null}\rangle :: p : c\langle\sigma\rangle$.

Case (STAT-BORROW) Without loss of generality, we assume that the variable x is on top of the stack. Note that borrowed variable must contain a unique value.

Assume $\Gamma \vdash \langle S \oplus x \mapsto U_x[v; H] \mid \mathtt{borrow}\ x :: \mathtt{unique}_b : c\langle\sigma\rangle\ \mathtt{as}\ \langle\alpha\rangle\ y\ \mathtt{in}\ \{\ s\ \}\rangle$.

1. By (CONFIG-STAT),
   (a) $\Gamma \vdash S \oplus x \mapsto U_x[v; H]$.
   (b) $\Gamma \vdash \mathtt{borrow}\ x :: \mathtt{unique}_b : c\langle\sigma\rangle\ \mathtt{as}\ \langle\alpha\rangle\ y\ \mathtt{in}\ \{\ s\ \}\ ; \Gamma$.
2. By 1.b and (STAT-BORROW),
   (a) $\Gamma \vdash x :: \mathtt{unique}_b : c\langle\sigma\rangle$ **ref** and
   (b) $\Gamma \oplus n :: \mathfrak{B}[\Gamma'] \vdash s \gg \Gamma \oplus n :: \mathfrak{B}[\Gamma' \oplus \Gamma'']$ where
   (c) $\Gamma' = \alpha \mapsto n \oplus y :: \alpha : c\langle\sigma\rangle$.

   Note that $y, \alpha \notin \mathsf{defs}(\Gamma)$ from 2.b).
3. By 1.a), 2.a) and Lemma 6.5.2 (Movement)
   $\Gamma \oplus n :: \mathfrak{B}[\Gamma_1[n/x] \oplus \Gamma'] \vdash S \oplus x \mapsto \mathit{null} \oplus B^b_n[H[n/x]; \alpha \mapsto n \oplus y \mapsto v]$
4. By 2.b), 3.), Lemma 3.4.1 (Extension) and (CONFIG-STAT),
   $\Gamma \oplus n :: \mathfrak{B}[\Gamma_1[n/x] \oplus \Gamma' \oplus \Gamma''] \vdash$
   $\langle S \oplus x \mapsto \mathit{null}, B^b_n[H[n/x]; \alpha \mapsto n \oplus y \mapsto v] \mid s\rangle$.
5. By 4.) and the induction hypothesis, if
   $\langle S \oplus x \mapsto \mathit{null}, B^b_n[H[n/x]; \alpha \mapsto n \oplus y \mapsto v] \mid s\rangle \to$
   $\langle S' \oplus x \mapsto v'', B^b_n[H'; \alpha \mapsto n \oplus y \mapsto v' \oplus F]\rangle$,
   then there exists a Γ_2 such that
   (a) $\Gamma \oplus n :: \mathfrak{B}[\Gamma_1[n/x] \oplus \Gamma' \oplus \Gamma''] \rightsquigarrow \Gamma_2$, and
   (b) $\Gamma_2 \vdash \langle S' \oplus x \mapsto v'', B^b_n[H'; \alpha \mapsto n \oplus y \mapsto v' \oplus F]\rangle$
6. By 2.c), 5.a) and definition of $\rightsquigarrow$,

   (a) $\Gamma_2 = \Gamma_3 \oplus n :: \mathfrak{B}[\Gamma_4 \oplus \Gamma' \oplus \Gamma_5]$ s.t.,
   (b) $\Gamma \rightsquigarrow \Gamma_3$,
   (c) $\Gamma_1 \rightsquigarrow \Gamma_4$, and
   (d) $\Gamma'' \rightsquigarrow \Gamma_5$.

7. By 5.b) and (CONFIG), $\Gamma_2 \vdash S' \oplus x \mapsto v'', B^b_n[H'; \alpha \mapsto n \oplus y \mapsto v' \oplus F]$.
8. By 5.a) and (EXPR-LVAL), $\Gamma_2 \vdash y :: \alpha : c\langle\sigma\rangle$.
9. By 7.) and def. of variable look-up,
   $(S' \oplus x \mapsto v'', B^b_n[H'; \alpha \mapsto n \oplus y \mapsto v' \oplus F])(y) = v'$.
10. By 7.), 8.), 9.) and Lemma 3.4.4 (Variable Look-up), $\Gamma_2 \vdash v' :: \alpha : c\langle\sigma\rangle$.
11. By 7.), 10.) and Lemma 6.5.6,
    (a) $\Gamma_3 \vdash S' \oplus x \mapsto v''$ and
    (b) $\Gamma_3 \vdash_x U_x[v'; H'[x/n]] :: \mathtt{unique}_b : c\langle\sigma\rangle$.
    By 2.a,b) $n \notin dom(\sigma)$, and thus $\sigma[x/n] = \sigma$.
12. By 2.a), 6.b) and Lemma 3.4.1 (Extension), $\Gamma_3 \vdash x :: \mathtt{unique}_b : c\langle\sigma\rangle$ **ref**.
13. By 11.a,b), 12.), and Lemma 3.4.9 (Variable Update),
    $\Gamma_3 \vdash S' \oplus x \mapsto U_x[v'; H'[x/n]]$.
14. By 13.) and (CONFIG), $\Gamma_3 \vdash \langle S' \oplus x \mapsto U_x[v'; H'[x/n]]\rangle$.

□

6.6 EXTERNAL-UNIQUENESS-AS-DOMINATING-EDGES

Theorem 6.6.1 (External-Uniqueness-as-Dominating-Edges)**.** *If* $\Gamma \vdash S\langle U_n[v; H]\rangle$, *then* $(\mathsf{defs}(H) \cup \{n\}) \# \mathsf{uses}(S)$.

The theorem states that if S is a well-formed stack with a hole and a unique $U_n[v; H]$ in the hole, then there are no pointers to n or H from any object in S. Thus, v is the only reference into H outside H and *is therefore a dominating edge* (see Section 6.4.1) as all paths into H from outside must contain it. (Observe that $v \in dom(H)$ by (VAL-UNIQUE).)

Proof. We prove this in two steps: a) $\mathsf{defs}(H) \# \mathsf{uses}(S)$ and b) $n \notin \mathsf{uses}(S)$.
First note that $\Gamma \vdash S\langle U_n[v; H]\rangle$ implies $n \notin \mathsf{defs}(\Gamma)$ by (VAL-UNIQUE).

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

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

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

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

□

6.7 Concluding Remarks

This chapter concludes our extensions to the Joline language. We have now presented owner-polymorphic methods, scoped regions and, finally, external uniqueness. External uniqueness is, we believe, more naturally suited to object-oriented programming, as it considers aggregate objects, allows internal aliasing without weakening the uniqueness invariant and overcomes the abstraction problem. External uniqueness is the first and only proposal to allow transfer of ownership in a system with deep ownership.

Scoped regions and external uniqueness were realised by introducing new kinds of owners in our system, and owner-polymorphic methods allow owners to be passed to subsequent stack frames. The owners introduced by scoped regions have some interesting effects on the ownership structure of a program which is discussed in the upcoming chapter.

Our extensions are orthogonal and work well together. Owner-polymorphic methods allow borrowing of external uniques without any additional borrowing constructs and does so without the added complexity or restrictions of previous borrowing proposals. Scoped regions allow the creation of heap-allocated objects that can store references to borrowed objects without risking residual aliasing once the borrowing has ceased. Following the upcoming discussion in the next chapter, Chapter 8 shows applications for our proposed constructs and some programming idioms which depend on them.

Chapter 7

Discussion

This chapter investigates properties and applications of our proposed constructs. Section 7.1 details how scoped regions and owner-polymorphic methods have turned the ownership tree into a directed acyclic graph and what this means in practice. Section 7.2 addresses the orthogonality of our proposed constructs, and describes how this is an improvement over some existing proposals.

7.1 Generational Ownership

As was explained in Chapter 3, the inside nesting relation of deep ownership forms a tree with *world* as its root. Every node in the tree denotes an object (except *world*) and every node is inside itself, its parent node and all its ancestors.

In Chapters 4 through 6, we have introduced new owners that do not correspond to objects. Owners introduced by scoped regions and borrowing blocks correspond to blocks and unique owners correspond to variables or fields. The special owner *free* denotes the *absence* of an owner. The introduction of scoped owners in combination with owner parameters to methods has some interesting effects on our system: a stack frame can have a nested subheap and objects in that subheap are allowed to reference any object on any previous frame. By a suggestion from John Potter, we call this *generational ownership* and say that each stack frame is a *generation*. Following the same terminology, intra-generation aliases can only exist from a *younger* generation to objects in an *older one*, one that is guaranteed to outlive the former.

7.1.1 Stack Frames are Generations

Owners introduced by borrowing blocks and scoped regions are tied to a stack-frame and can never become visible on earlier frames. Thus, objects belonging to a frame f can never be referenced by objects belonging to an earlier frame.

If objects on a frame cannot reference objects on a later frame, each stack frame (or block, really) effectively becomes a root in the object graph. The graph subsequently becomes a forest where each tree corresponds to a stack frame. The ban on aliasing from previous to later frames is interesting as it means that as soon as a stack frame is destroyed, all objects belonging to the corresponding generation (subgraph of the object graph) can be safely garbage collected without creating dangling pointers. This is in a sense similar to generational garbage collection [9].

As aliases cannot cross from a older to a younger generation, reasoning about aliasing becomes easier. Objects belonging to the current generation can be safely passed to methods executing in objects in older generations, guaranteed not to be statically aliased and changing objects in the current generation cannot have any effects on objects in earlier generations.

7.1.2 Ownership is a dag

The introduction of scoped owners has another important impact on the interpretation of ownership. As a scoped owner is potentially inside all owners visible in the enclosing scope, ownership is no longer a tree but a directly acyclic graph as a node can have more than one parent node. By virtue of owner-polymorphic methods, any owner might be visible in any method scope. Thus, a scoped region is inside all owners on all preexisting frames, even though the permissions to reference these objects must be explicitly passed in as owner parameters to be used. We recall the rule (IN-GENERATION) that captures this in the store-typing:

$$\frac{\Gamma \vdash q \quad \Gamma \bullet \Gamma' \vdash p \quad p \in \mathsf{defs}(\Gamma')}{\Gamma \bullet \Gamma' \vdash p \prec^* q} \text{(IN-GENERATION)}$$

This rule shows that younger generations are ordered inside older one. Any owner in the generation(s) typed by Γ' is inside any owner in the generation(s) typed by Γ.

It makes sense to order stack frames inside each other, as we have indeed done

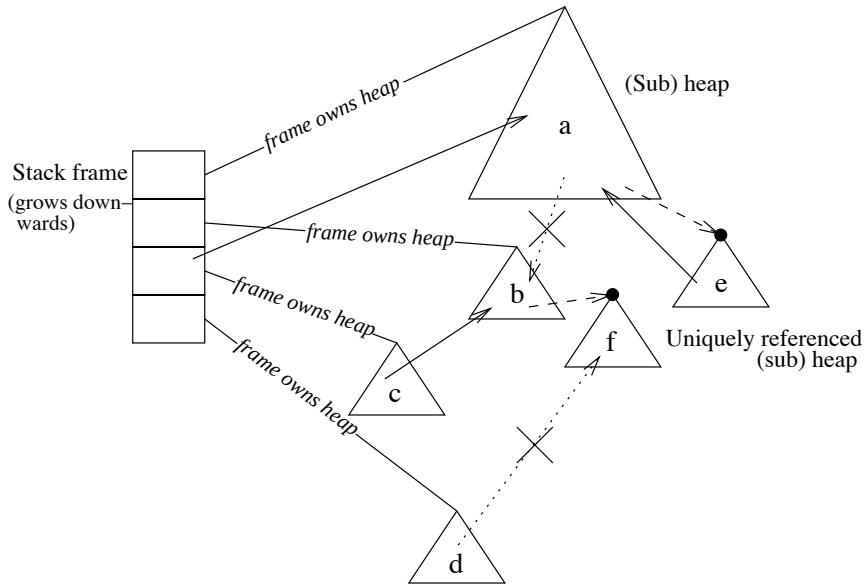

Figure 7.1: Generational ownership. $a - d$ are generations and e, f are unique subheaps. Later *generations* may refer to earlier *generations* (earlier in the alphabet, unique subheaps are not included in this definition), but not vice versa. Only one external reference is allowed for unique subheap.

in the formalisation of Joline. The top stack frame corresponds to the region *world* and the "starting statements", *s;* **return** *e*, of a program can be thought of as implicitly wrapped inside the scoped region *(world)* { ... }. The resulting model is clean and easy to understand, see Figure 7.1.

Figure 7.1 shows a downwards growing stack. Each frame is a root in the object graph and the tree nested inside the frame is denoted by a triangle. Uniquely referenced subheaps are also shown as separate triangles that only have one incoming reference. Dashed arrows denote unique references and crossed-out arrows denote references that are not valid and will never exist in the system.

7.2 ORTHOGONALITY

Our proposed constructs, external uniqueness, scoped regions and owner-polymorphic methods, are orthogonal in nature. External uniqueness and the borrowing block en-

ables unique pointers, a strong notion of aggregate, and transfer of ownership; scoped regions allows stack-local objects with permission to reference all visible objects and generational ownership; and owner-polymorphic methods allows owner-parameters, borrowed arguments and reuse of a method with different owners.

The orthogonality of our constructs allows them to be combined to make use of each other to express constructs from previous proposals. For example, we can achieve traditional borrowing [73, 92, 7, 8, 30, 32, 27, 6, 87] using a combination of our borrowing block and the borrowed references due to the owner-polymorphic methods. The result is an improvement over traditional borrowing as it does not need any additional "borrowed pointer" concept, but can be expressed directly through owners. This is cleaner, and, combined with scoped regions, allows the borrowed pointer to flow to the heap, without loosing track of the borrowing, which makes our borrowing more expressive than previous borrowing proposals.

In Chapter 9, we show how we can enable a form of borrowing that retains the uniqueness of the borrowed reference with a straightforward extension to the borrowing block keeping the owner-polymorphic methods as they are.

We now move on to describe applications of our proposed constructs.

Chapter 8

Applications

8.1 Applications for External Uniqueness

This chapter presents example applications of our proposed constructs. In particular, we show how external uniqueness can be used to enable *transfer of ownership*—moving objects between representations—and *merging representations* in the presence of ownership types. Additionally, we show how to use our borrowing construct, scoped regions and owner-polymorphic methods to encode 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 use transfer of ownership to overcome the initialisation problem, pointed out by for example Detlefs et al. [48], and allow external initialisation of an object's representation.

Many of these examples were previously impossible to encode in a system with ownership types.

8.1.1 Transfer of Ownership

Transfer of ownership is an important design pattern in concurrent object-oriented programming [82]. Ownership of an object is transferred from one object to another after which the first object must release all its references to the moved object.

Figure 8.1 shows the implementation of a token ring. The token object is passed

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

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

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

Figure 8.1: A Token ring implementation.

from one object to the next by calling the *give( )* method. Figure 8.2 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 break the owners-as-dominators property. 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 move on to other examples where transfer of ownership is beneficial.

Object Pools The use of object pools is a well-known optimisation technique. Allocating memory for an object, determining that an object is garbage, and freeing it are expensive operations. Thus, in systems where many short-lived objects are needed to perform some task, instance creation operation can be replaced by a pool of possibly initialised objects from which "new objects" are taken when needed and returned instead of disposed.

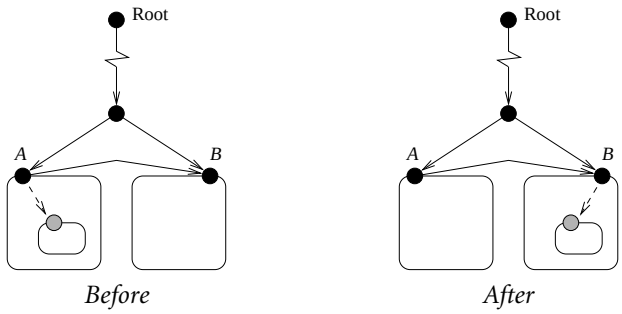

Figure 8.2: Transfer ownership of the gray object from A (left) to the sibling B (right).

In previous ownership types systems, owners are fixed for life. This is will present a problem when implementing object pools, as every object pool can only pool objects of one specific owner. This has the disadvantage of not being able to reuse objects in the pool between owners, and requiring a separate pool for each type of object pooled. The only way to avoid this is to make all objects in the pool belong to *world*, which precludes using objects from the pool in any object's representation and voids all aliasing guarantees.

With external uniqueness, we can implement object pools where the objects in the pool are referenced by unique pointers. 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.

Object pools would be particularly useful in the presence of unique borrowing (see Section 9.1), since this enables the object to be automatically returned to the pool after it has been used.

Avoiding Unnecessary Synchronisation Uniqueness can be used to avoid unnecessary synchronisation [5, 22, 23]. 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 traditional uniqueness only applies to a single object, moving the object from one thread to another might not move its representa-

tion along if there are incoming aliases to it from other parts of the program. Thus, the aggregate would be split between threads. 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.

An interesting consequence of external uniqueness is that it voids the need for synchronisation. Once a thread has entered a unique object, the unique reference to the object is temporarily destroyed and the borrowed references created as a result of the borrowing exist only on the stack of the current thread. Thus, there is no need to synchronise any subsequent method calls internal to the unique aggregate. Furthermore, as external uniqueness enables a strong notion of aggregate, moving uniques between threads cannot cause the object to be split between threads. These are powerful consequences.

Boyapati's SafeJava [27, 22] and Bacon et al.'s Guava [8] void the need for synchronisation when the receiver is unique (an unshared value in Guava's case), and for thread-local objects.

8.1.2 Merging Representations

The encapsulation of deep ownership prevents objects from merging their representation as the representations of different objects have different owners and an object's owner is fixed for life. Thus, with deep ownership, merging, for example, the sets of links of two lists requires copying, which is problematic with respect to copying strategies, sharing, performance penalties, object identities, etc. This is a severe drawback, but luckily, we can overcome it using external uniqueness.

Figure 8.3 shows the merging of two doubly-linked lists without copying using external uniqueness. The key to enable this is to make the links sibling objects in a uniquely referenced aggregate. The *append( )* method of the first list is invoked with the second list as its argument. The phases of the operation are (in order):

* The first list borrows its own head link (the entire list) using a temporary owner (here *ho*).

** The head link of the other list is moved into a variable owned by *ho*, that is, the two lists now share a common owner.

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

class List< data outside owner >
{
    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;
            }
        }
    }
}
```

Figure 8.3: Merging two doubly-linked lists. *ho* is the temporary owner of the list head while borrowed.

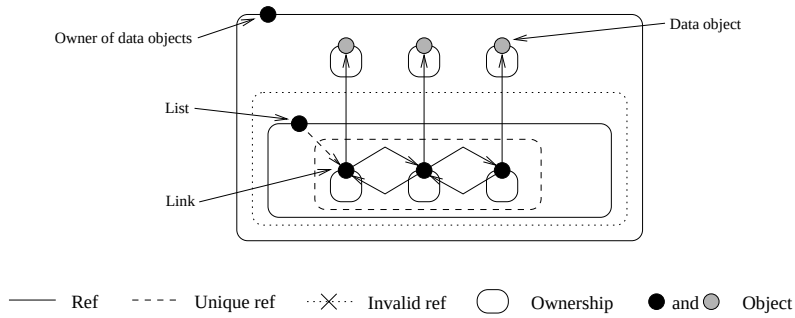

Figure 8.4: The object graph for the doubly-linked list in Figure 8.3. The dotted box denotes the possibility of any number of owners between the owner of the data objects and the list object. The dashed box denotes the ownership bound of the externally unique set of links.

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.

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

Our original Joline proposal [44] was the first system with deep ownership types to overcome this challenge, originally proposed by David Holmes after seeing the original ownership types proposal [42].

8.1.3 Simulating Borrowing

The traditional borrowing construct prevents a reference from escaping to the heap. Thus, there will be no residual aliasing of a borrowed argument to a method when the method exits. This allows borrowed arguments and receivers to be automatically reinstated, which alleviates some of the pain of programming with uniques.

Using external uniqueness in combination with owner-polymorhic methods, we can simulate traditional borrowing. Our borrowing block allows a unique reference

```
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
    }
}
```

Figure 8.5: 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( )*.

to become a regular, non-unique for the duration of the block. This reference can be passed as a borrowed argument to owner-polymorphic methods, which guarantee that no aliases to the borrowed argument survive the method call, except aliases returned or stored internally in the borrowed argument. When the borrowing block exits, the borrowed value is reinstated.

Traditional borrowing has downsides: first, it requires the addition of a borrowed pointer concept, that is neither a unique nor a non-unique pointer; second, it prevents the borrowed reference from being stored temporarily on the heap, even if the object storing it will not survive the borrowing. The reason for this is that previous proposals simply lack mechanisms in their type systems to treat borrowed references as usual non-unique references but maintain the uniqueness invariant between borrowings. Our borrowing proposal overcomes both of these shortcomings, which is explained in detail below.

```
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 > list2 = new scoped:List< temp >;
            list2.add( bb );
            ...
        }
        ...          //
    }
    ...              //
}
```

Figure 8.6: 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.

Pass a Borrowed Object as an Argument

As is shown in Figure 8.5, our borrowing block construct takes a unique reference and converts it to a non-unique reference, stored in the temporary variable *b* whose type has the temporary owner *temp*. We call *b* the borrowed variable. The contents of the borrowed variable can be passed as a borrowed argument to the method *method* by passing *temp* as a borrowed owner. This allows *method* to temporarily alias *b*'s contents.

Since *temp* did not exist before the borrowing, no previously existing external objects' types are parameterised with *temp*. Thus, no such object can store a permanent reference to the borrowed object on the heap, which is the necessary condition to allow *bb* to be reinstated with *b* at the end of the borrowing block.

Store a Borrowed Object on the Heap

Owner-polymorphic methods can use an owner parameter as an owner on objects it creates. Such objects may naturally store borrowed references on the heap. This

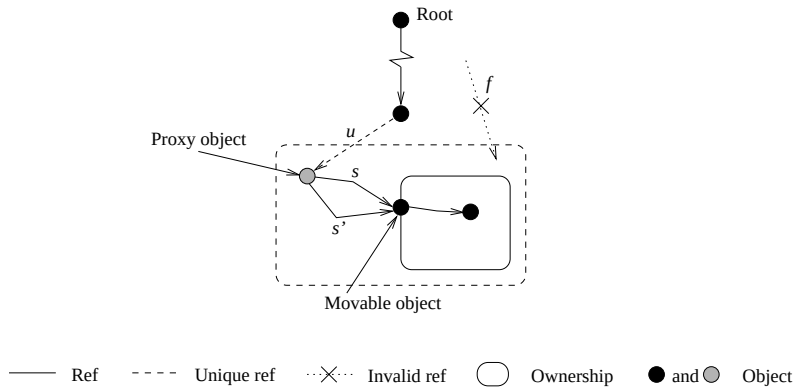

Figure 8.7: Movable aliased objects—the dashed box denotes the "virtual boundary" of the unique proxy. s, s' are sibling references to the movable object. Owners-as-dominators applies as usual. There could of course be movable objects (or rather, all sibling objects are part of the movable aliased aggregate).

does not leak the borrowed reference as the newly created objects will be siblings, or iternal to, the borrowed objects. An example of using borrowed owners in types of new objects is shown in Figure 8.6. 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. Objects owned by *scoped*, such as *list2*, will be permitted to reference *b*, but will not survive the scoped region.

Outside the borrowing block, *temp* is not visible, which means that no aliases to *b* are accessible. Thus, it is safe to reinstate the value in *b* to *bb* when the block exits.

8.1.4 Movable Aliased Object Pattern

The key behind all examples we have seen so far is that all active pointers to a unique are accounted for since there is only one. We now show how external uniqueness can be used to implement a pattern that allows for several pointers into a data structure and still have all the benefits of uniqueness. The key is to store these pointers in a proxy that is unique itself. We call this the Movable Aliased Object pattern, sometimes omitting "aliased" when it is obvious what we mean.

Figure 8.7 shows the object graph for a movable aliased object and its unique proxy, in this case, a simple tuple. It also shows a virtual ownership bound for the unique ob-

ject and its siblings. We say that bound is virtual, since it does not correspond to a 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. Accessing the movable aliased object simply requires the proxy to be borrowed.

We now show an example application for movable objects.

List with Head and Tail Links Figure 8.8 shows the code for a doubly-linked list where the set of links are encapsulated in a unique proxy. The links are thus strongly encapsulated, and can be merged with another list without copying. This cannot be handled by traditional uniqueness.

As is seen in Figure 8.8, the proxy object class *HeadAndTail* 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. Because of the presence of a tail pointer, there is no need to iterate through the list to get to the last list element as in 8.3. The resulting object graph is shown in Figure 8.9. The 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 *append()* method from Figure 8.3 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. If our language had tuple types, as Haskell [106] 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.

8.1.5 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 the owner, meaning it is impossible to create or initialise a representation object externally. This has been a limitation with many of the previous systems with deep ownership without uniqueness.

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

class HeadAndTail< data outside owner >
{
    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;
            }
        }
    }
}
```

Figure 8.8: Movable aliased objects enable head and tail pointers to unique set of links in a list. The *append( )* method allows lists to be merged without copying.

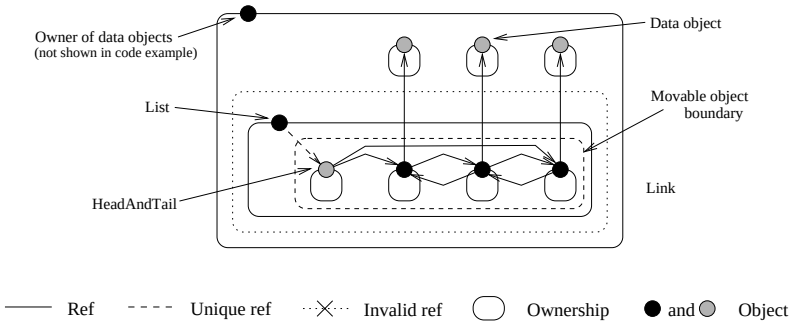

Figure 8.9: Object graph for the doubly-linked list with head and tail pointers in Figure 8.9. The dashed box denotes the virtual bounds of the owners of the unique objects. The virtual bound corresponds to the owner of *handle*.

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

In Joline, we can overcome the initialisation problem using movement and transfer of ownership. The externally initialised objects are created as externally unique 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 references from objects external to the target remain.

Figure 8.10 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 implementer of the lexer to disregard any external aliasing, which makes the implementation easier, voids the need for checks that no-one has for example moved the file pointer externally of the lexer class, and makes it easier to maintain and reason about class invariants.

Having described a few applications for our proposed constructs, we now move on to consider a few additional extensions to the Joline language, *unique borrowing*, *existential downcasts* and a way to enable iterators through a closure-like construct.

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

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

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

Figure 8.10: Overcoming the initialisation problem.

Chapter 9

Extensions

In this chapter, we present two additional extensions to the full Joline system—*unique borrowing* and *existential downcasting*—that further increase the expressiveness of the system. Unique borrowing is a minor extension that enables borrowed pointers that maintain their uniqueness. Existential downcasting enables downcasting in a style similar to SafeJava's [21], but without requiring a run-time representation of owners.

9.1 Unique Borrowing

Unique borrowing maintains the uniqueness of the borrowed variable and uniquely borrwed references can be reinstated at the end of the borrowing, regardless of how the borrowed variable was used in the borrowing block.

For simplicity, in our formalisation of Joline, we chose not support *unique borrowing*. Instead, borrowing loses uniqueness and methods cannot determine whether an incoming borrowed object is aliased or not. This weakness is shared with other borrowing proposals [73, 92, 7, 88, 27, 6].

In this section, we show how to overcome this weakness with a trivial extension to Joline, whose constituents are formalised and proven sound in other parts of the system.

9.1.1 Towards Unique Borrowing

How to achieve unique borrowing is best explained by example. The following method declaration effectively declares a method that does unique borrowing:

<b **inside** *world>* **void** *someMethod(unique[b]:Object arg) { ... }*

The borrowed parameter *arg* is a regular borrowed parameter, but the borrowed owner *b* is used not as an owner, but as a movement bound. In *someMethod( )*, *arg* cannot be statically aliased nor can it be moved into some preexisting object as it has a borrowed type. Using scoped owners or the owner *b* to create objects, objects allowed to statically alias *b* can be created allowing *arg* to be moved to the heap. However, such objects will be invalidated when *someMethod( )* exits. Thus, when *someMethod( )* exits, all references to *arg*, unique or not, will be invalidated and the run-time system can reinstate the unique value in the variable passed as an argument.

However, if *b* is used elsewhere in the signature of *someMethod( )*, the unique borrowing could no longer be verified, as an alias to *arg* could be created in an object that survives the method call, causing *arg* to be non-unique when the method exits.

There are two solutions to this problem: either we prevent the movement bound of the borrowed unique to appear more than once in the method signature, and disallow owners nested inside it; or we can simply introduce a fresh movement bound for a scope, and delay the reinstatement to a point when the scope exits. In the latter case, the fresh movement bound acts as a guard to guarantee that no aliases created to the borrowed reference remain when the scope of the bound exits, just like in a regular borrowing block.

We now show how this could be achieved using a special kind of borrowing blocks that preserve uniqueness.

9.1.2 Borrowing Blocks that Preserve Uniqueness

The key to achieving unique borrowing is to make sure that no aliases to the borrowed object exists at the time of reinstatement. We encode unique borrowing using an additional borrowing block construct that works exactly like our regular borrowing block, but maintains the uniqueness of the borrowed object and uses the borrowed owner as a movement bound. The reinstatement of a uniquely borrowed variable is "delayed" to the exit of the borrowing block, as opposed to directly after a method call. Additionally, the method call is allowed to lose the uniqueness of the borrowed argument

```
<b inside world> void someMethod( unique[b]:Object o ) { ... }

<b inside world> void other( unique[b]:Object o, b:Object p ) { ... }


unique[z]:Object x = ...;
borrow x as unique[fresh]:Object y in
{
    // x = null
    // fresh is a new owner that is only related to z (is inside)
    someMethod<fresh>(y--); // pass fresh and y to someMethod
    // y = null
}
// x is now reinstated

borrow x as unique[fresh]:Object y in
{
    fresh:Object local = ...;
    other<fresh>(y--, local); // local can swallow y in other
    // y = null
}
// x is now reinstated
```

Figure 9.1: Two examples of unique borrowing. The variable x is borrowed twice and passed as a unique argument to methods *someMethod( )* and *other( )*. After each borrowing block, it is safely reinstated.

as any such aliases will be tied to the lifetime of movement bound introduced by the borrowing block. For concreteness, we give an example.

Figure 9.1 shows an example of unique borrowing. It declares two methods. The first method will never consume its argument in such a way that it would be invalid to reinstate after the invocation. The second method could however alias o inside p (possibly losing the uniqueness of o) making it unsound to reinstate o after the invocation. The first borrowing block shows our extended borrowing construct. The borrowed variable is given a fresh movement bound and is moved into the variable y with its uniqueness maintained. Upon the entering into the block, x is nullified. We pass y as an argument to *someMethod( )* after which y is nullified. When the block is exited, we can reinstate x with its original value, or some other value in y.

The second borrowing statement is similar to the first, but illustrates that other

argument objects are safe. Any aliases to *y* saved in *local* by the method will be inaccessible when *fresh* goes out of scope. Thus, we can reinstate *x* when the block exits, just as above.

As it is the block and not the method invocation that is the unit of unique borrowing, several blocks are needed to pass a unique as uniquely borrowed to several methods. This is an inconvenient, but could probably be solved by automatically inferring the borrowing blocks.

A downside of this proposal is that there are now two borrowing blocks, with subtly different semantics—one that maintains uniqueness and one that does not. We feel that if the programmer is required to explicitly state the type of the borrowed variable, just as above, this problem is minor.

9.2 Existential Downcasting

Many statically typed object-oriented programming languages with manifest typing rely on subtype polymorphism and downcasts for standard code reuse. Methods are generally polymorphic in their parameters—an object of a subclass can be passed as argument to a method that expects a more general type. The method can then downcast the object to access the extended protocol. This practice is complicated by ownership types as owners of the target type must be considered.

Boyapati et al. [24] propose a way of doing *dynamically checked* downcasts with deep ownership where ownership information is stored at run-time. The downside of such a proposal is the overhead of keeping track of owners at run-time. If a class has four owner parameters, this means four additional pointers to the corresponding actual owners in each instance in a naive implementation. While Boyapati et al. show a way to only include owners for instances whose classes are involved in operations that make use of run-time ownership information, the cost is still high.

In this section, we propose a form of downcasting into an existential type where the owners and their relations are derived from the header of the class cast to. The existential type can be used in the same fashion as a borrowed type. The implementation, however, does not require run-time owner representation. The downside or our proposal is that the types cast to are never type-compatible with any other types in scope.

```
public boolean equals( Object o )
{
    if ( o == this )
    {
        return true;
    }
    if ( !( o instanceof List ) )
    {
        return false;
    }

    ListIterator e1 = listIterator( );
    ListIterator e2 = ((List) o).listIterator( );      // Note the cast
    while ( e1.hasNext( ) && e2.hasNext( ) )
    {
        Object o1 = e1.next( );
        Object o2 = e2.next( );
        if ( !( o1==null ? o2==null : o1.equals( o2 ) ) )
        {
            return false;
        }
    }
    return !( e1.hasNext( ) || e2.hasNext( ) );
}
```

Figure 9.2: The equals method in *java.util.AbstractList*.

9.2.1 The Importance of Downcasting

In Java, downcasting is frequently used to overcome the shortcomings of the static type system. Prior to Java 5.0, and the introduction of parametrically polymorphic classes, container classes stored data objects as instances of *Object*. As a consequence, type information of an object stored in a container was lost when the object was later retrieved and dynamically checked downcasting was essential to regain the type information.

Even in Java 5.0, downcasts are frequently used. A good example can be seen in the *equals( )* method declared in *Object*, the superclass of all objects. This method is supposed to be overridden in all classes for which structural equality is sensible. In the Java API, all *equals( )* methods have the same signature: **boolean** *equals(Object o)*. The common implementation of such a method is to check that the argument is of the

correct type; if so, cast the argument to the desired type, and then perform equality tests of the contents of the objects. Figure 9.2 shows the equals method for the class *AbstractList* from the standard Java API bundled with Java 1.4.2 (before generics was introduced in the Java language).

In a system with ownership types, the downcast in Figure 9.2, would not be possible, as the following code snippet shows:

```
<some inside world> boolean equals( some:Object o )
{
    ...
    ... ((some:List< ? >) o).listIterator( ); // problematic cast
    ...
}
```

In short, the problem above is that the owner, marked *?*, introduced by the cast of *o* to a *List* is unknown. Making *equals( )* owner-polymorphic does not solve the problem: first, it would not be clear how to map owner parameters to parameters cast to, and second; overriding of *equals( )* methods would be broken as the number of owner arguments would vary. In short, this would not solve the problem but make the system additionally complicated. Below, we introduce the concept of existential owners that overcomes the problem of downcasting to a type with unknown owners.

9.2.2 Existential Owners

The idea behind our proposal is simple: if a Java-style downcast (disregarding ownership) of an object to some class c succeedes, then we could infer the owner parameters necessary to form the new type from c's class header. We call the inferred owners existential owners, and types that use them existential types.

For example, if we can check that the class of the value of a variable typed *a:Object* is actually a list, we know that its actual type is *a:List<b>* where *b* is some owner outside *a*. Thus, we can simply add *b* to the current scope, with that nesting information. Adding owners in this fashion can never enable breaking owners-as-dominators as we must already have permission to reference the object (its owner), and all owner parameters are always outside this permission.

Existential owners completely avoid a run-time owner representation to the price of being forced to conservatively treat existential owners derived from a cast as differ-

ent from all other owners in scope, possibly even when the same variable is downcast twice (unless we can clearly see that it has not been modified since the last downcast). The syntax of existential downcasting is thus:

a:List< b > list = (a:List< b >) someObj; // someObj has type a:Object

The statement introduces *b* as a new owner that is outside *a*, inferred from the header of the *List* class. The variable declaration is optional. However, without it, every cast of *someObj* will yield a new type with fresh existential owners that will not be type compatible with *a:List< b >*.

We define existential owners and existential types thus:

Definition 9.2.1 (Existential Owner)**.** An existential owner is an owner that is introduced by a type cast.

Existential owners have no statically known relations to other owners, except for the owners in the type cast where they were introduced. Existential types are bound to their enclosing scope and thus have the same restrictions as scoped owners, see definition 5.2.1.

Definition 9.2.2 (Existential Type)**.** In our setting, an existential type is a type that uses an existential owner for one or more of its owner parameters.

Again, if *someObj* can be cast to a *List*, it must have (at least) two owners, the already known owner *a* and an owner of the data objects in the list, here *b*. As we clearly have permission to reference *a* which is inside all the "hidden" owner parameters of *someObj*, we can safely access them without breaking encapsulation. As we see it, allowing the downcasting is simply a way to overcome the (technical) difficulty of deriving what owner parameters should have been passed to the method would the desired type of the argument be known outside, something we expect to be generally impossible. Our solution preserves pure polymorphism for methods without introducing any additional complexities in the system. It also preserves abstraction as it is not visible external to the method how the method will downcast its object.

Figure 9.3 shows the use of existential downcasting to implement structural equality tests for a list using an *equals( )* method. The existential owner *b* must be introduced to type the data object *obj*, even if the *list.get(i)* call is "inlined" in the method call on line †. The owner *b* is visible in the scope from the line where it is defined to where the

```
// In List class
< a inside world > public boolean equals( a:Object arg )
{
    if ( arg == this )
    {
        return true;
    }
    if ( arg != null && arg instanceof List )
    {
        a:List< b > list = (a:List< b >) arg;
        if ( list.length( ) == this.length( ) )
        {
            for ( int i=0; i < this.length( ); ++i )
            {
                b:Object obj = list.get( i );
                if ( this.get( i ).equals< b >( obj ) == false ) // †
                {
                    return false;
                }
            }
            return true;
        }
        else
        {
            return false;
        }
    }
    return false;
}
```

Figure 9.3: Existential downcasting used in implementation of an *equals* method.

enclosing block exits. Type theoretically, downcasting to introduce existential owners is much like unpacking an existential package, which introduces a new type variable (also called a witness) [107].

Having a run-time representation of owners is orthogonal to existential downcasting. In a system with run-time representation of owners, such as SafeJava [21], downcasting into a set of *known owners* would be possible using dynamic checks. This is more powerful, but comes with the cost of keeping track of a potentially large mass of ownership information at run-time. When downcasting a *List* instance to an *ArrayList* with the same number of owner parameters in its type, our existential downcasting would not need to invent any existential owners to produce a well-formed type. Thus, in cases such as this, our solution is as expressive as SafeJava's but completely without the run-time overhead.

In short, existential downcasting preserves the expressiveness of pure polymorphism in situations where downcasting (also called reverse polymorphism [35]) is necessary to access an argument variable's extended protocol. Without existential owners, downcasting would be reduced to work only in situations with a invariant number of owners on the possible types, like in the *List* and *ArrayList* example above. As the existential types are only valid for the duration of the method (or a nested block in the method), they are effectively borrowed. Thus, the owners-as-dominators property is upheld. An interesting property of our proposal is that the code remains independent on the actual values of the owners, which is not the case for SafeJava.

9.3 Iteration Revisited

A frequently pointed-out problem with the deep encapsulation of ownership types is that it does not support iterators for lists, as iterators require an object that can access a lists representation to be externally visible. This combination is invalid in an ownership types setting.

Using objects, we can do a crude simulation of closures in Joline: An external "action object" is created with the method performing the intended iteration. The action object is then passed into the list. Using double dispatch and owner-polymorphic methods the action object is then passed a temporary permission to reference a properly encapsulated iterator. Example code for this is shown in Figure 9.4. It relies on the existence of an abstract *ActionObject* class which it extends with an implementation

of the iterate method to enable the *iterate( )* method in *List* to be reused for different kinds of iterations.

The code for the iterator class looks like any iterator, modulo the ownership annotations. It also presupposes a superclass, *Iterator*, that is used to hide the fact that the list's representation is visible in the list iterator's type. Inside the *List*, a scoped owner *iter* is used to preserve separation of the iterator and the list's representation and to capture the iterators temporary nature.

Naturally, the downside of this solution for iteration (and tasks with similar problems) is the addition of a new class to the program for each iteration that the program must perform. With the addition of closures or higher-order functions, this pain would be somewhat relieved. In any case, we avoid the more ad hoc solution employed by Boyapati [21] that allows instances of inner classes to have access to the representation of the enclosing instance, without being confined to it. For our proposed solution here, the aliases violating deep encapsulation will always be dynamic, which is safer.

For a detailed analysis of iterators and encapsulation, see work by Noble [96].

We now move on to describe our practical evaluation of our Joline system.

```
class MyCustomAction< data outside owner > extends ActionObject< data >
{
    < iter inside world > void iterate( iter:Iterator< data > iter )
    {
        // code for iteration
    }
}

class List< data outside owner >
{
    this:Link< data > first;

    < temp outside data > void iterate(temp:ActionObject< data > ao)
    {
        (iter)
        {
            iter:Iterator< data > iter = new iter:ListIterator< this, data >( first );
            ao.iterate< iter >( iter );
        }
    }
}

class ListIterator< links outside owner, data outside links >
                                                    extends Iterator< data >
{
    links:Link< data > current;

    ListIterator( links:Link< data > first ) { current = first; }

    boolean hasNext()
    {
        return current.next() != null;
    }

    data:Object next()
    {
        data:Object result = current.data();
        current = current.next();
    }
}
```

Figure 9.4: Using objects for performing iterations

Chapter 10

Practical Evaluation

10.1 The Joline Compiler

We have a prototype implementation of the compiler, which was developed together with Johan Östlund, using the Polyglot framework [100]. It compiles an extended version of Joline into byte-code that is executable on a regular Java VM. For this implementation, Joline was extended with primitive data types, standard conditionals and loops, void method returns and name-based encapsulation. All these constructs that are not in our formalisation in order to make it simpler and well-understood, and, importantly, do not in any way interact with ownership. We also added *print* and *read* routines operating on strings for trivial input and output.

Additionally, we added class variables and class methods to Joline (through a *static* keyword, just like in Java). These work as expected, though the only visible owner inside a static method is *world*. This hampers the usability of class variables, which will be discussed below. Class methods can be owner-polymorphic which relieves much of the restrictions of only having permission to reference *world*-owned objects.

In the sections below, we list a few issues with the current Joline compiler. See also Östlund's masters thesis [103] for a related discussion as well as an account of the practical usefulness of the Joline language. Some parts of this chapter are based on work together with Gustaf Cele and Sebastian Stureborg and were also reported in their masters thesis [36], an early attempt at evaluating deep ownership and external uniqueness, using manual checking instead of a compiler.

10.1.1 Class Variables

It is common to use class variables to share objects between instances of the same class. This is complicated by the fact that the only visible owners in a class scope is *world*. Thus, class variables can only hold *world*-owned objects, which is sensible but limiting. Objects shared in class variables are owned by world and can only reference objects owned by world (in addition to its representation). This will force objects to be *world*-owned even if they really should not be, and thus not protected by deep ownership.

A possible way to circumvent this problem is to assign an owner to a class object. This allows the use of the *owner* variable in the class scope, or even an additional set of parameters for the class object. This enables, for example, a class *p:Player* where *p* is the owner of the object representing the *Player* class in run-time. Thus, the class variable *worldMap* in *p:Player.worldMap* can be encapsulated inside the owner *p*, instead of world. Naturally, adding an owner to the *Player* class object will have effects on where the class can be used. The owner *p* will also bound the possible owners of player instances. If player instances were given an owner outside *p*, encapsulation of *worldMap* would be broken. Preventing this through a static check is simple. This approach is taken by Potanin et al. [109].

For flexibility, we can allow several parallel copies of a class for class objects with different owners. For example, we could have two copies of the class *Player*, *p:Player* and *q:Player*. In this case, all class variables (at least all class variables owned by *owner*) would be different for each copy of the class. Thus, *p:Player.worldMap* and *q:Player.worldMap* would refer to different fields in different copies of the *Player* class.

This model would also fit well with the possibility, in Java, of having several copies of a class in memory simultaneously, one for each class loader.

10.1.2 String Literals

The only object literal in Joline is the string literal. During the implementation, the issue was raised of how to treat the string literal in terms of ownership and types. We finally decided to make string literals create unique objects, and gave them a type that allows maximal movement, *unique[world]:String*. Thus, a string literal has the most general type possible, and is thus possible to assign to any string variable, regardless of ownership. As we use Java strings for our implementation, strings are immutables and

could thus be shared without problems. We leave such optimisations for future work.

Notably, manifest ownership, [38, 110], would enable a string to be used a type without owners, but would lose encapsulation.

10.1.3 Studies of Aliasing and Programming with Ownership

Surprisingly few practical studies of deep ownership have been performed.

In Aldrich's dissertation [2], the ArchJava language is evaluated with several large implementations. The evaluation is however mostly concerned with enforcing architectural properties, and the ownership system of the ArchJava language is shallow. It is therefore not possible to generalise from this study to our setting.

Boyapati's dissertation [21] includes a practical evaluation of SafeJava's implementation of the Run-Time Specification for Java. Eight programs in sizes from 56 LOC to 1850 LOC are ported from Java to SafeJava with, changing one LOC for roughly every 45 LOC. This study however foremost evaluates a regions system in SafeJava and tests memory management policies, not deep ownership types or uniqueness. No details on the nature of the programs implemented are provided.

Noble and Potanin's [98] study of ownership and confinement in regular Java programs finds that less than 15% of all objects have more than one pointer to them. This study is optimistic as it is based on analysing heap snapshots. Thus, it is possible that some objects deemed unique were used in a way violating uniqueness in-between snapshots. For their corpus of programs, the average depth of object nesting was around 5 or 6, although it is not possible to draw any conclusions in terms of the compatibility of deep ownership and "real-world programming" from this observation.

Recently, Hackett and Aiken [69] have studied aliasing in over a million lines of C code. Their findings are interesting, even if they may not be directly applicable to object-oriented code. They find that "[...] in real programs, aliasing has a great deal of structure reflecting the structure of the program" and that "[...] outside the data structures that use aliasing, aliasing is very rare". Notably, of 291 intentional aliasing situations found in their analysis, 68 aliases, or around 23%, were pointers inwards in a nesting structure. Note that this does not necessarily suggest that 23% of the aliasing situations would have been invalid in a system with deep ownership as their notion of nesting considers actual paths and not ownership nesting. For example, it might well

be sensible for all nodes in a binary tree to have the same owner, which would allow the inwards pointers above.

The next section reports results from our experiences of programming with deep ownership types and external uniqueness.

10.2 Case Studies

This section discusses case studies of programming with the constructs in the Joline programming language. The first two case studies are from work done together with master students at Stockholm University.

10.2.1 Case 1: Informal Ownership

For their masters thesis, Cele and Stureborg [36] posed the question "how does ownership types hamper development"?

In order to answer this question, three programs were designed and two implemented. Due to the lack of a publicly available compiler for systems with deep ownership at the time, the implementations were done in Java, and manually inspected to respect deep ownership and external uniqueness. The programs considered were: a prototype for a program for plagiarism detection using Markov Chains [89] for string comparisons; and two board games with existing designs, "Dragon Fort" and "Settlers of Catan". Only the second game was implemented. The programs were of different sizes and complexity, and were upon initial inspection determined to have a fair amount of interconnections between objects, something that we hypothesised would make the evaluation more relevant and "put more pressure" on the constructs being evaluated. In their analysis, Cele and Stureborg reports that not the size of the program, but the nature of its interconnections seemed to be the most influential factor on the successful use of ownership types.

An inspection of the ownership graph for the Settlers program reveals that 33% of all classes have its instances in *world*. Of these, 80% expose details of some supposedly encapsulated object inside some other structure, following the inner class pattern of Boyapati et al. [26]. These were however only used for reading, or could be replaced with a *unique listener pattern* (see Section 10.3.4). To us, the successful implementation of these programs, 850 LOC for the Markov Chain implementation and 5600 LOC for

Table 10.1: Pointer statistics from the program evaluated in Östlund's thesis [103]. The second percentage number on the Total row is the percentage of uniqueness when ignoring primitives.

	Unique	Non-unique	Primitive
Local variables	23	16	29
Fields	16	1	5
Class variables			3
Parameter	25		18
Return	29		25
Total	93 (49% / 85%)	17 (9% / 15%)	80 (42%)

the Settlers game (including comments but excluding the use of Java libraries), suggests that using the encapsulation of deep ownership is possible even for non-trivial programs, even though the results are inconclusive.

It is hard to draw any grand conclusions from this study, partly because of the lack of a compiler but also because parts of the program were written without respect for ownership types (the graphical user interface). Our own inspections of their programs, did not reveal any accidental violations of ownership, but this kind of manual checking is of course unreliable, especially for the 5600 LOC Settlers program. Naturally, defaulting ownership to world is always possible, meaning that a system could always be rewritten using deep ownership types, but without taking advantage of its enforcement of encapsulation.

In their concluding discussion, Cele and Stureborg raise an interesting question regarding ownership and reuse: "Will the presence of a fixed set of ownership parameters make classes harder to reuse?" The discussion about downcasting and *equals( )* method in Chapter 9 suggests that this is the case. Our owner-polymorphic methods enable reuse of a method on arguments with different owners. Existential downcasting eliminates some need for passing owners around. We expect type parametricity on classes to help as well, and patterns such as the Hide Owner pattern (Section 4.2.4). More research is necessary to understand the interplay of deep encapsulation, our proposed constructs and software reuse.

10.2.2 Case 2: A Library System

For his masters thesis, Östlund [103] implements a 1200 LOC program in Joline, a library system with different libraries, books and borrowers. This project was an assignment from a course on object-orientation that was not developed with uniqueness or ownership in mind. Our examination of this code reveals a total of 190 type declarations on variables, fields, parameters and returns. In an effort to put Noble and Potanin's positivistic uniqueness result to a test, Östlund made an effort to make as many pointers as possible unique. The results from our inspection of his resulting program is shown in Table 10.1. Around 49% of all type declarations can be made unique, and only 9% of all variable declarations denote shared objects (less than 5% if one counts static aliases only). Of the primitive variables used, some were kept in synch between objects in an effort to keep track of where a unique object belonged. For example, all books are unique, but the library needs too keep track of what books it owns—thus, the book has an int valued identifier that are copied in the catalogue of the library to which the book belongs. For non-uniques, this could have been implemented by keeping a pointer to the object and use that for comparison only. Additional inspection revealed that on all occasions, simply returning the object's hash value would have sufficed, which would have reduced use of primitive types further.

Even if Östlund's program is not very large, it shows that it is indeed possible to use the Joline language in practice. Östlund's hypothesis, extrapolated from Noble and Potanin's [98] results, that it would be possible to program exclusively with externally unique pointers, hold true for the program, which suggests that its uses of uniques does not deviate from real-world programs and that the presence of ownership did not preclude uniqueness. As the most pointers in the program are unique, it is not surprising that the strong encapsulation provided by external uniqueness does not impose any limitations on the program's structure.

The statistics for Östlund's program in Table 10.1 show declarations, not actual pointers, which of course varies with the number of instances at a given point. Nevertheless, a majority of the fields in the classes that are expected to have many instances are unique (approximately by a factor of 7:3), so for a program with a large number of such instances, the percentage of unique pointers will converge to 70%. When primitives are disregarded, the factor is around 6:1. Destructive reads were used 30 times, about 1 for every 40 lines of code.

As most pointers are unique, the level of encapsulation in the library system is

very high. The single non-unique field used is a sibling reference (that is, its owner is *owner*), and the only uses of the global *world* owner are as a movement bound, mostly on unique strings.

The use of additional integer ids for each object as described for books above could be questioned as it would perhaps normally be implemented using sharing. This is discussed in relation to the id-aliases concept in Section 10.3.3.

10.2.3 Case 3: SuDoku

Our first study of using the Joline compiler was porting a program for solving simple SuDoku puzzles to Joline. The original program was written in Ruby, and was first ported to Java without using any of the Java libraries for which there are yet no Joline counterparts. The resulting program was 390 lines of code, including a minimal list implementation. We black-box tested the program against the output of the Ruby program and found that we got the same results for the same input. We then ported the program to Joline, adding ownership parameters to classes and types and trying to make as many pointers as possible unique. Porting the program from Java to Joline took less than one hour, not including debugging and logging a few unsatisfactory error messages produced by the Joline compiler, and correcting a bug in the type-checking of field assignments.

The resulting program was 470 lines long, 80 lines longer than the original Java program, not counting the code duplication due to the lack of generics. The lack of generics forced a simple copy-and-replace operation to create a new list class for each class of objects that was stored in lists, much like manually doing the work of the C++ template mechanism. Existential downcasting would have overcome this, but it was not implemented in the Joline compiler at the time.

Of the 80 additional lines, about 65 were due to borrowing blocks, all of which could be removed by adding a trivial borrowing block inference to the compiler (further discussed in Section 10.3.5). The remainder of the code added was due to the lack of public or package scoped variables in Joline which forced code to be moved from one place to another, using the "move method" and "move field" refactoring patterns [56]. Of the 470 lines, around 70, plus an additional 40 counting beginning and end brackets of borrowing blocks, contained a Joline-specific construct, such as a type with owner parameters or an owner-polymorphic method. Owner-polymorphic methods

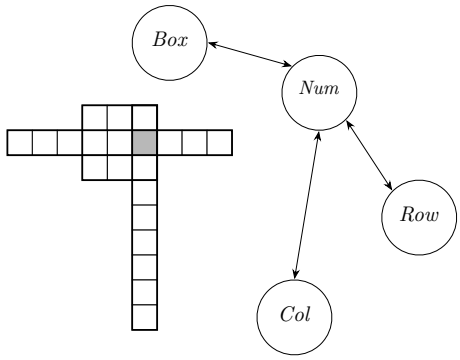

Figure 10.1: Reference Structure of the SuDoku program. The gray object corresponds to the *Num* instance, and box is the larger 3x3 structure.

were used, adding only one external borrowing block in the client code for each use to temporarily pass the entire sets of rows, columns and boxes around to synchronize the object structure during grid setup.

At no time during the porting did ownership types cause any problems. Each square on the SuDoku board was represented by a mutable *Number* instance that was shared between three "SuDoku-aware" containers. Each number also had references to each container to which it belonged. An illustration can be found in Figure 10.1. The gray square is represented in the program by the *num* object that belongs to *row*, *col* and *box* (the larger 3x3 structure).

The ownership structure that made most sense to us was to make numbers, rows, columns and boxes all siblings owned by a grid object, representing the entire puzzle. Adding the corresponding annotations to the program was straightforward.

Joline's lack of support for public variables slightly strengthened the program's encapsulation. The first attempt at fixing this compiler error was to simply create an accessor method for the variables that could no longer be accessed directly from outside the objects. As these turned out to be representation objects, the accessor method were still not accessible externally, forcing the external code to be moved inside the object. As the external object was directly manipulating another object's internals, the change was consistent with the "move method" refactoring pattern [56] and the resulting code was actually improved.

10.2.4 Case 4: Linked List

Our last study involved porting the *LinkedList* class from the Java collections framework in the Java API to Joline, including its superclasses and inner classes. It turned out, however, that quite a few things present in the collections framework, such as external iterators, conversion into arrays, creation from arrays, etc., were not possible to implement in Joline, as the language currently lacks such features. Thus, any comparisons with size and numbers of changes are uninteresting. The size of the code for the ported classes were a total of 750 LOC excluding comments. The porting took three hours over several days, as this was done while the compiler was still unstable and bugs surfaced during development. The bulk of the time was spent pondering over dependencies between pieces of code cut out for the aforementioned reasons, and how to factor out inner classes etc. in a satisfactory way. Perhaps unsurprising, adding ownership annotations was straightforward and keeping the list's representation separate from data objects easy.

Having ported the linked list to Joline, we set out to change its implementation into using unique links, to enable merging of two or more lists. As the list was implemented as a double-linked list, we had opportunity to use the Movable Aliased Object pattern described in Section 8.1.4. Wrapping the links in a uniquely referenced proxy allowed the links to remain non-unique. It was thus, not necessary to make any changes to the internal structure of the class describing the list nodes. This entire reimplementation took less than one hour.

The merge-able list totalled 860 LOC, a total of 110 lines added. The size of the small proxy class to make the links moveable aliased objects was 16 LOC excluding methods moved into it from the *LinkedList* class in order to avoid a few cases of borrowing. A few interesting refactorings resulted from this design change. The list implementation relied heavily on the use of an internal iterator supplied by the method *listIterator( )* to manipulate the links. Previously, the iterator could share the links of the list, but as these were encapsulated in an externally unique proxy, this was no longer possible. In order to create an iterator, the proxy object had first to be borrowed and supplied to the method creating the list iterator, somewhat similar to the iterator discussion in Section 9.3. (An alternative design would have been to move the iterator to where it was needed and there do the borrowing, but we chose this design to avoid having to manually move the proxy object into place after each iterator use.) This mechanism was necessary as borrowed objects in a borrowing block cannot be returned (we would

```
// Code from LinkedList—returns an internal iterator
< temp inside data > this:Iterator< temp, data > listIterator( temp:Proxy proxy )
{
    checkBoundsInclusive(index);
    this:ListIterator< temp, data > result = new this:ListIterator< temp, data >( );
    result.setup( index, size( ), proxy.first, proxy.last );
    return result;
}
// Code from AbstractSequentialList
data:Object get( int index)
{
    // This is a legal listIterator position, but an illegal get.
    if ( index == size( ) )
    {
        return null;
    }
    borrow first as temp:proxy in
    {
        temp:ListIterator< data > itr = listIterator< temp >( proxy, index );
        return itr.next( );
    }
}
```

Figure 10.2: Code from the linked list implementation.

lose track of them in our type system). Thus, the borrowing was done at the call site and the borrowed proxy passed to the *listIterator( )* as an argument, as is visible in Figure 10.2. The method was decorated with an owner-parameter for the borrowed proxy and returned a rep iterator wrapping the link nodes from the proxy. This also allowed the iterator code to remain intact. The borrowing made sure that the proxy would be safely reinstated after the iteration was performed and the method exited. As the chain of invocation never left the list object for the iterator creation, this design does not smell particularly bad, even though closures, or higher order messages could perhaps have avoided this chain of argument passing and returning. Figure 10.2 shows two methods from the ported code: *listIterator( )*, that creates and sets up the iterator, and *get( )*, that uses an iterator created by the first method to return an object from the list for a given index.

A total of 28 borrowing situations were added, around 60 lines of code, all operating on the proxy reference, the only unique reference in the list. Around 50% of these were trivial enough to be inferred by the simple inference mechanism described in Section 10.3.5.

In conclusion, even though we were forced to remove parts of the original class' implementation, the port to Joline was straightforward. When changing the implementation to use unique pointers, our Movable Aliased Objects pattern allowed the double-linked structure to remain intact and imposed only a minimum of changes. We look forward to using a more complete Joline language for similar experiments in the future.

10.3 Results

A few shortcomings, though unrelated to our proposed constructs, of the Joline compiler surfaced pretty early in our evaluation. The lack of interfaces, inner classes, arrays and exceptions made porting parts of the Java API to Joline too much work. This forced us to rewrite a few utility classes from scratch, and prevented us from porting larger existing programs to Joline, as rewriting them was easier than redesigning uses of inner classes, etc. In this section, we summarise the results from our practical evaluation.

10.3.1 Design Delicacy

The first case studies and discussions with the master students involved, suggest that the design phase of software development becomes more important, as the ownership structure of a program must also be considered when forming associations. Sometimes, an additional redesign of the ownership structure was necessary in order to add a permission or remove one that was no longer needed. Cele and Stureborg states [36]:

> "A programmer trying to 'hack up' a non-trivial system without any particular focus on design would experience great difficulties using ownership types (unless he would disregard ownership structures completely and assign ownership of everything to world)."

However, they also report:

> "We designed our systems serially, and found that as our experience with ownership types grew, our first draft of the system had a rather clear ownership structure and needed fewer changes in the design as the design process proceeded."

We conclude that more longitudinal studies must be undertaken to investigate this further.

10.3.2 The Importance of Generics and Downcasts

Very early in our first attempts at programming in Joline, we were struck by the lack of a downcast operation, something we were surprised that no-one had pointed out in papers on ownership types. We have already discussed the use of a downcast for equals methods in Section 9.2, but there are many other uses. A hidden problem with the list example for deep ownership is that unless there is a downcast operation, the list must be rewritten for each type of data objects stored in lists. Boyapati's SafeJava system [21] includes a downcast operator, but with a quite heavy run-time overhead. A better solution is to use generics, as done by Potanin et al. [109, 110]. Extending the Joline language with generics is a definite direction for future work.

10.3.3 Id-aliases

Östlund's library system implements a unique id scheme parallel to reference identity for certain objects. This enables an object to be unique, and compared against its additional id. Östlund uses this for example for copies of book—a book is unique and moved into its borrower when borrowed from the library. The library still needs to keep track of what copies it owns, and does so through a unique integer number in all instances of *Book*. Thus, returning a book to the wrong library will not succeed. This scheme is used in other places of the library program as well.

Although there is not enough evidence to support it, there may be a pattern to this—pointers are used to represent a relation, in this case some kind of ownership, not to invoke methods. In these cases, the value of the pointer is the interesting value, not the object the pointer points to.

Boyland et al. [32] presents a capability system for pointers where combinations of capabilities can be used to express constructs such as read-only and traditional uniqueness. Their system of capability combinations include a "null capability", for references

that may only be used to perform id comparison, not to read fields or invoke methods. Allowing *external id-aliases*, pointers with null capability, to a unique object would constitute an additional weakness of uniqueness. However, since these pointers are innocuous, the effective uniqueness invariants would be the same (modulo the fact that comparing a unique pointer to another pointer might return true). There seems to be no reason why id-aliases cannot be weak references (in the GC sense).

10.3.4 Unique Listener Proxy Pattern

The unique listener proxy pattern was discovered by Cele and Stureborg during their implementation of the Settlers program. It overcomes the problem of adding a listener to an external object in an event generator that does not have all the necessary owner parameters in its type. The external object interested in the events can create a uniquely referenced listener proxy, say an anonymous instance of *ActionListener*, with a back-reference to the external object to forward received events. Subtyping can be used to hide all owner parameters except owner, which is unique. The unique listener proxy can then be moved into the representation of the object generating the events, without the need for that object to have explicit rights to reference the object "wrapped" inside the listener proxy. This also allows an object to have a list of subscribers with different owners as the proxy listeners will share the common owner, determined by the event generator itself. An example of use of the unique listener proxy pattern is shown in Figure 10.3. A similar effect can also be achieved by the Hide Owner pattern in Section 4.2.4, with the exception that the listener proxy is created by the event generator itself, and that uniqueness need not be involved.

10.3.5 Syntactic Overhead

The Joline system imposes some syntactic overhead on a program. For example, in the library system, around 150 of the 1200 lines contain Joline-specific constructs, such as a variable declaration with owners, a class header with owner parameters, or a borrowing block: roughly one every eight lines. Around 35 of these are borrowing blocks, which is not surprising as most references are unique. Manual inspection shows that 25 (about 70%) of the borrowing blocks could be inferred from the program with a trivial inference algorithm. For the SuDoku program, about 80% of the code added in the porting was borrowing blocks of which every single one could have been inferred.

```
class Observer
{
    // implementation details omitted for brevity
}

class MyListener< observer outside owner > extends Listener
{
    observer:Observer observer = null;
    void setObserver( observer:Observer o )
    {
        observer = o;
    }
    void notification( )
    {
        observer.doSomething( );
    }
}

class EventGenerator
{
    this:Listener listener;
    void registerListener( this:Listener l )
    {
        listener = l;
    }
    void updateSomething()
    {
        ... // make some change
        listener.notification( );
    }
}

// a is inside b
a:EventGenerator gen = ... ;
b:Observer obs = ... ;
unique[b]:MyListener< b > lis = ... ;
lis.setObserver( obs );
gen.registerListener( lis-- )
```

Figure 10.3: The Unique Listener Proxy Pattern.

As an example, a common use of a borrowing block in the SuDoku system had the following shape:

```
borrow numarray as temp:arr in
{
    arr.atPut( pos, num );
}
```

where *temp* is the temporary owner for the borrowed reference stored in the temporary variable *arr*. The owner *temp* is never used and an automatic transformation from an expression like *var.method(args)* to

```
borrow var as tempOwner:tempVar in
{
    tempVar.method( args );
}
```

when *var* is unique seems like a trivial addition. This suggests that there is much to gain even from adding the most simple borrowing inference algorithm to the Joline compiler.

An additional syntactic overhead may be relieved if we consider a default nesting in ownership parameter declarations on class heads. In both the SuDoku case and in the library program, all owner parameters in all class declarations were declared **outside** *owner*, which thus seems to be a good target for a default. Thus, when omitting an outside declaration, we could use "**outside** *owner*" as a default. Naturally, studying a larger code body is necessary before drawing any conclusions about the practical gains of such a scheme, but it seems like a promising approach.

10.3.6 Concluding Remarks

Extending the Joline compiler to include all relevant features found in Java will allow us to port Java programs to Joline. We believe that will provide us with a superior way of evaluating deep ownership and uniqueness, and that this approach might also give some interesting techniques for refactoring into using ownership. Thus, Joline needs to be extended by at least interfaces, inner classes, exceptions, downcasting, generics and arrays. This will require some additional designs in terms of what owners of exceptions should be, whether or not to support run-time representation of owners, and what the possible owners of inner classes are.

On a side note, being forced to implement our own programs and library classes from scratch was cumbersome and forced us to deal with rather small programs which might well have affected the validity of our studies. However, implementing everything ourselves was instructive.

Having presented our experiences of programming in the Joline language, we move on to our concluding chapter.

Chapter 11

Conclusions

In this thesis we have presented Joline, a class-based object-oriented programming language with deep ownership extended with support for owner-polymorphic methods, stack-local objects with scoped regions, and externally unique pointers. We have formalised the language, and proven its soundness and the important structural properties, owners-as-dominators, inherited from Clarke's ownership types, and our own external-uniqueness-as-dominating-edges.

We have argued that our approach to uniqueness is better suited to object-oriented programming, because it considers entire aggregates and not just single objects, because it allows internal aliasing without weakening the uniqueness invariant, and because it overcomes the abstraction problem inherent in all previous proposals.

All our proposed constructs aid the programmer in dealing with aliasing. They do not directly aid in formulating mathematical proofs about a program, but as they give strong encapsulation invariants that govern possible aliasing, we believe them to be indirectly helpful. Recent research has shown that there are benefits of using ownership to simplify formalising and proving invariants about a program [39, 13].

For the practical evaluation, four case studies have been made, that suggest that the deep encapsulation and our proposed constructs have not obstructed programming to make it impossible to write "real programs" in Joline, but more research would be instructive. Furthermore, the Joline language needs to be extended further to include constructs that facilitate porting of larger applications to the Joline language.

In this last chapter, we examine our proposal with a critical mind and summarise our findings, as well as directions for future work.

11.1 CRITIQUE

Below, we look at the design of the Joline language and its practical evaluation.

11.1.1 The Joline Language

Joline is a rather complex language. In hindsight, developing a much simpler core language would easily have taken one year off of the time required to perform this thesis work. The complexity of the Joline language made is hard to formalise in a way that could not only be proved sound, but that was also easy enough for someone to follow the proofs enough to be convinced of their correctness. In particular, as movement causes changes to the store type, early versions of the Joline formalisation were extremely tricky and subtle as every subexpression caused the set of previously known fact about the layout and typing of the store to be invalidated. In the end, we are quite happy with the final formalisation as it models the nesting and uniqueness in a way that we think is intuitive and closely resembles how a programmer might think about deep encapsulation. It is possible that a system with a regular, flat heap would have been easier to prove sound, but it would not have allowed us to formalise our structural invariants in such a nice and direct a way.

11.1.2 Practical Evaluation

The practical evaluation suggests that Joline introduces a quite heavy syntactic baggage on the programmer. When effort is made to keep many references unique, methods explode with code for borrowing the uniques and reinstating them. If a unique object is called once in the beginning of a method and once in the end, a programmer might be tempted to wrap the entire method call in a borrowing block for the unique object for convenience. This has the downside of precluding simultaneous borrowings of the unique during the other parts of the method, but could also be more reliable as the unique is guaranteed to still be accessible at the end of the method. In any case. the proposed inference mechanism for wrapping method calls on uniques in implicit borrowing blocks would reduce the code bloat significantly.

Even though the programs implemented to evaluate the Joline language were rather varied, the program sizes have been fairly small, from a couple of hundred lines of code to a couple of thousand. Even though our results have been satisfactory, we would like

to extend our evaluation to much larger programs, and study the evolutions of such programs over time and how deep ownership interacts with maintenance and refactoring, an issue remains unclear. In trivial experiments, a simple refactoring as "move method" has sometimes required changes to a class' header to include additional ownership information, or the removal of owner parameters that are no longer needed. Such changes are likely to propagate to large parts of the program. A positive side-effect of this propagation is that the effect of moving a method in terms of aliasing becomes much clearer. In situations where propagating changes are undesirable, we expect the "hide owner" pattern (described on page 96) to come in handy.

Our practical evaluation did not involve enough refactoring to generalise any results from. We believe that a study of how an ownership annotated code-base is maintained and refactored over a longer period of time would prove instructive in these matters.

11.2 Summary of Conclusions

Programming with aliasing is unavoidable in contemporary object-oriented programming languages. We have presented a language that aids programming with aliasing. We allow the programmer to express strong encapsulation invariants in a clear way that is statically checkable. We can lend permissions to clients temporarily and confine permissions to a specific scope. We can express the concept of unique pointers and provide ways of programming with uniqueness that are more general and less complex than previous mechanisms. Encapsulation and uniqueness are statically checkable, and programs that violate them will not compile. Our constructs are defined for a class-based, object-oriented programming language with inheritance and subtyping.

Our practical experiences of writing Joline programs show us that our proposed constructs work well with common programming idioms, but that additional work, to ease the syntactic overhead, such as adding our proposed borrowing block inference, should be done to make programming with unique pointers more feasible. The Joline language lacks a few programming constructs that are painful to live without, but these are not really related to our constructs for alias control. We will extend Joline with such constructs and continue our evaluation on larger programs.

We believe that our implementation of external uniqueness is better suited to the object-oriented setting as it preserves abstraction, considers entire aggregates and al-

lows more lax restrictions (back pointers) without weakening the uniqueness invariant. Subsequent to our original proposal of external uniqueness, it was included in SafeJava [21], to replace its previous, flawed version of uniqueness.

11.3 Future Work

Following the extension of our Joline compiler, we will continue to evaluate the Joline language and our proposed constructs in a larger programs. We plan a range of extensions to the language, notably type inference for local variables, borrowing inference, existential downcasting, generics, interfaces and inner classes. Evaluating and porting larger programs will hopefully give insights into the interplay of our proposed constructs and maintenance and refactoring. Hopefully, we will be able to extract more useful patterns for programming with ownership-based constructs.

We will also look further into id-aliases and mechanisms for read-only or immutable pointers to see whether there are situations where encapsulation restrictions can be relieved without creating opportunities for errors.

Bibliography

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

[2] Aldrich, J. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, August 2003.

[3] Aldrich, J., and Chambers, C. Ownership domains: Separating aliasing policy from mechanism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (Oslo, Norway, Jan 2004), M. Odersky, Ed., vol. 3086 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 1–25.

[4] Aldrich, J., Chambers, C., and Notkin, D. ArchJava: Connecting software architecture to implementation. In *ICSE* (May 2002).

[5] Aldrich, J., Chambers, C., Sirer, E. G., and Eggers, S. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the Sixth International Static Analysis Symposium* (Venezia, Italy, September 1999), no. 1694 in Lecture Notes in Computer Science, Springer-Verlag, pp. 19–38.

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

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

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

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

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

[11] Banerjee, A., and Naumann, D. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM 52*, 6 (November 2005), 894–960.

[12] Banerjee, A., and Naumann, D. A. Ownership transfer and abstraction. Tech. Rep. KSU CIS-TR-2004-1, Kansas State University, October 2003.

[13] Barnett, M., DeLine, R., Fähndrich, M., Leino, K. R. M., and Schulte, W. Verification of object-oriented programs with invariants. *Journal of Object Technology 3*, 6 (June 2004), 27–56.

[14] Berard, E. V. Abstraction, encapsulation, and information hiding. Essay. Available from: http://www.toa.com/.

[15] Birka, A., and Ernst, M. D. A practical type system and language for reference immutability. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2004), ACM Press, pp. 35–49.

[16] Blair, G., Gallagher, J., Hutchison, D., and Shepherd, D., Eds. *Object-Oriented Languages, Systems and Applications*. Halsted Press, New York, New York, April 1991.

[17] Blanchet, B. Escape analysis: correctness proof, implementation and experimental results. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1998), ACM Press, pp. 25–37.

[18] Bokowski, B., and Vitek, J. Confined Types. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications* (1999).

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

[20] Booch, G. *Object-Oriented Design With Applications*. Benjamin/Cummings, Menlo Park, California, 1991.

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

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

[23] Boyapati, C., Lee, R., and Rinard, M. Safe concurrent programming in Java. In *MIT LCS/AI Student Oxygen Workshop (MIT SOW 2002)* (Gloucester, Massachusetts, July 2002).

[24] Boyapati, C., Lee, R., and Rinard, M. Safe runtime downcasts with ownership types. In *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming* (July 2003), D. Clarke, Ed., UU-CS-2003-030, Utrecht University.

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

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

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

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

[29] Boyland, J. Alias killing: Unique variables without destructive reads. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems* (Lisbon, Portugal, June 1999), At ECOOP'99.

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

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

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

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

[34] Buckley, A. Ownership types restrict aliasing. Master's thesis, Department of Computer Science, Imperial College of Science, Technology, and Medicine, Queen's Gate, London, June 2000.

[35] Budd, T. *An Introduction to Object-Oriented Programming*, 3rd ed. Addison-Wesley, 2002.

[36] Cele, G., and Stureborg, S. Ownership types in practice. Master's thesis, DSV, Stockholm University, January 2005.

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

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

[39] Clarke, D., and Drossopolou, S. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications* (November 2002).

[40] Clarke, D., Noble, J., and Potter, J. Overcoming representation exposure. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems* (Lisbon, Portugal, June 1999), At ECOOP'99.

[41] Clarke, D., Noble, J., and Potter, J. Simple ownership types for object containment. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (June 2001).

[42] Clarke, D., Potter, J., and Noble, J. Ownership types for flexible alias protection. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications* (1998).

[43] Clarke, D., Richmond, M., and Noble, J. Saving the world from bad beans: Deployment-time confinement checking. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications* (Anaheim, California, November 2003).

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

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

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

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

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

[49] Dietl, W., and Müller, P. Universes: Lightweight Ownership for JML. *Journal of Object Technology 4*, 8 (2005), 5–32.

[50] Donne, J. Meditation XVII—Devotions upon Emergent Occasions, 1624.

[51] Eilëns, A. *Principles of Object-Oriented Software Development*, 2nd ed. Pearson Education, 2000.

[52] Ernst, E., Ostermann, K., and Cook, W. R. A virtual class calculus. In *Proceedings of Principles of Programming Languages (POPL)* (Charleston, South Carolina, USA, January 2006).

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

[54] Flanagan, C., and Abadi, M. Types for Safe Locking. In *Programming Languages and Systems* (March 1999), vol. 1567 of *Lecture Notes in Computer Science*, pp. 91–107.

[55] Fluet, M., and Pucella, R. Phantom types and subtyping. In *TCS '02: Proceedings of the IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science* (Deventer, The Netherlands, The Netherlands, 2002), Kluwer, B.V., pp. 448–460.

[56] Fowler, M. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

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

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

[59] Gay, D., and Steensgaard, B. Fast escape analysis and stack allocation for object-based programs. In *9th International Conference on Compiler Construction (CC'2000)* (April 2000), vol. 1781, Springer-Verlag, pp. 82–93.

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

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

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

[63] Gosling, J., Joy, B., Steele, G., and Bracha, G. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[64] Graham, I. *Object-Oriented Methods*. Addison-Wesley, Reading, Massachusetts, 1991.

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

[66] Grogono, P., and Chalin, P. Copying, sharing, and aliasing. In *Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94)* (Montreal, Quebec, May 1994).

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

[68] Grothoff, C., Palsberg, J., and Vitek, J. Encapsulating objects with confined types. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications* (2001).

[69] Hackett, B., and Aiken, A. How is aliasing used in systems software?, November 2005. Unpublished.

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

[71] Hejlsberg, A., and Wiltamuth, S. *C# Language Specification*. Microsoft Corporation, 2000.

[72] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM (CACM) 12*, 10 (1969).

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

[74] Hogg, J., Lea, D., Wills, A., de Champeaux, D., and Holt, R. The Geneva convention on the treatment of object aliasing. *OOPS Messenger 3*, 2 (Apr. 1992), 11–16.

[75] Joyner, I. *Object Unencapsulated, Java, Eiffel and C++??* Object and Component Technology Series. Prentice Hall PTR, July 1999.

[76] Kent, S., and Howse, J. Value types in Eiffel. In *TOOLS 19* (Paris, 1996).

[77] Kent, S., and Maung, I. Encapsulation and aggregation. In *TOOLS Pacific 18* (1995).

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

[79] Kobayashi, N. Quasi-linear types. In *26th ACM Symposium on Principles of Programming Languages* (Jan. 1999).

[80] Krishnaswami, N., and Aldrich, J. Permission-based ownership: encapsulating state in higher-order typed languages. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM Press, pp. 96–106.

[81] Kulczycki, G. W. *Direct Reasoning*. PhD thesis, Graduate School of Clemson University, May 2004.

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

[83] Leavens, G. T., Baker, A. L., and Ruby, C. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, H. Kilov, B. Rumpe, and I. Simmonds, Eds. Kluwer Academic Publishers, Boston, 1999, pp. 175–188.

[84] Leino, K. R. M. Data Groups: Specifying the Modification of Extended State. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications* (1998).

[85] Leino, K. R. M., and Müller, P. Object invariants in dynamic contexts. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (2004), vol. 3086 of *Lecture Notes in Computer Science*, pp. 491–515.

[86] Leino, K. R. M., and Müller, P. Modular verification of static class invariants. In *Formal Methods (FM)* (2005), J. Fitzgerald, I. Hayes, and A. Tarlecki, Eds., vol. 3582 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 26–42.

[87] Leino, K. R. M., Poetzsch-Heffter, A., and Zhou, Y. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (June 2002), vol. 37(5), pp. 246–257. Available at http://softech.informatik.uni-kl.de/downloads/publications/pldi02.pdf.

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

[89] Manning, C., and Schütze, H. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[90] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[91] Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.

[92] Minsky, N. Towards alias-free pointers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (July 1996).

[93] Mitchell, N., and Sevitsky, G. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (January 2003), L. Cardelli, Ed., vol. 2743, pp. 351–377.

[94] Müller, P. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

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

[96] Noble, J. Iterators and Encapsulation. In *TOOLS Europe 33* (Mont St-Michel, La Belle France, June 2000), pp. 431–442.

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

[98] Noble, J., and Potanin, A. Checking ownership and confinement properties. In *4th Workshop on Formal Techniques for Java Programs* (Malaga, Spain, June 2002).

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

[100] Nystrom, N., Clarkson, M. R., and Myers, A. C. Polyglot: An extensible compiler framework for java. In *Proc. 12th International Conference on Compiler Construction* (Warsaw, Poland, April 2003), vol. 2622 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 138–152. Available from http://www.cs.cornell.edu/Projects/polyglot/.

[101] O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning abour programs that alter data structures. In *CSL, Springer Verlag, LNCS 2142* (2001).

[102] O'Hearn, P., Reynolds, J. C., and Yang, H. Separation logic and information hiding. In *Proceedings of the POPL Symposium on Principles of Programming Languages* (2004), ACM.

[103] Östlund, J. Realizing external uniqueness ... or how I learned to stop worrying (about representation exposure) and love the owner. Master's thesis, DSV, Stockholm University, December 2005.

[104] Palme, J. Protected program modules in Simula 67. *Modern Datateknik 12* (1973).

[105] Parkinsson, M., and Bierman, G. Separation logic and abstraction. In *Proceedings of the POPL Symposium on Principles of Programming Languages* (Long Beach, California, USA, January 2005), ACM.

[106] Peyton Jones, S., Hughes, J., et al. Haskell 98 — A non-strict, purely functional language. Available from `http://haskell.org`, Feb. 1999.

[107] Pierce, B. C. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[108] Pizlo, F., Fox, J. M., Holmes, D., and Vitek, J. Real-time Java scoped memory: Design patterns and semantics. In *Proceedings of the 7th IEEE International Symposium on Object-oriented Real-time distributed Computing, (ISORC)* (2004).

[109] Potanin, A., Noble, J., and Biddle, R. Generic ownership: practical ownership control in programming languages. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2004), ACM Press, pp. 50–51.

[110] Potanin, A., Noble, J., Clarke, D., and Biddle, R. Featherweight generic ownership. In *Formal Techniques for Java-like Programs (FTfJP)* (July 2005).

[111] Potter, J., Noble, J., and Clarke, D. The ins and outs of objects. In *Australian Software Engineering Conference* (Adelaide, Australia, November 1998), IEEE Press.

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

[113] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *LICS* (2002), pp. 55–74.

[114] Rumbaugh, J. R., Blaha, M. R., Lorensen, W., Eddy, F., and Premerlani, W. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[115] Shepherd, G., and Kruglinski, D. *Microsoft Visual C++ .NET (Core Reference)*. Microsoft Press, Redmond, WA, USA, 2002.

[116] Secture internet programming group, 1997. http://www.cs.princeton.edu/sip/news/april29.html.

[117] Skoglund, M., and Wrigstad, T. A mode system for read-only references in java. In *Formal Techniques for Java Programs, in Conjunction with ECOOP 2001* (Budapest, Hungary, 2001).

[118] Skoglund, M., and Wrigstad, T. Alias control with read-only references. In *Sixth Conference on Computer Science and Informatics* (March 2002).

[119] Smith, F., Walker, D., and Morrisett, G. Alias types. In *European Symposium on Programming* (Berlin, Germany, March 2000), vol. 1782.

[120] Smith, M., and Drossopoulou, S. Cheaper reasoning with ownership types. In *International Workshop on Aliasing, Confinement and Ownership in Object-oriented Programming*, D. Clarke, Ed., UU-CS-2003-030. Utrecht University, July 2003, pp. 15 – 28.

[121] Snyder, A. Encapsulation and inheritance in object-oriented programming languages. *Proceedings of OOPSLA '86, ACM SIGPLAN Notices 21*, 11 (1986), 38–45.

[122] Stroustroup, B. *The C++ Programming Language*, 3rd ed. Addison-Wesley, 1997.

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

[124] Thomas, D., Fowler, C., and Hunt, A. *Programming Ruby: A Pragmatic Programmer's Guide*, 2nd ed. Addison-Wesley, October 2004.

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

[126] van Rossum, G. Python reference manual. Report CS-R9525, Centrum voor Wiskunde en Informatica, P. O. Box 4079, 1009 AB Amsterdam, The Netherlands, April 1995.

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

[128] Walker, D., and Morrisett, G. Alias types for recursive data structures. In *Workshop on Types in Compilation* (Montreal, Canada, September 2000). Avaliable as Carnegie-Mellon University Technical Report CMU-CS-00-161.

[129] Walker, D., and Watkins, K. On regions and linear types. In *International Conference on Functional Programming* (2001), pp. 181–192.

[130] Wirfs-Brock, R., Wilkerson, B., and Wiener, L. *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

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

[132] Yates, B. N. A type-and-effect system for encapsulating memory in Java. Master's thesis, Department of Computer and Information Science and the Graduate School of the University of Oregon, August 1999.

[133] Zhao, T., Noble, J., and Vitek, J. Scoped types for real-time java. In *25th IEEE International Real-Time Systems Symposium (RTSS'04)* (2004), pp. 241–251.

[134] Zhao, T., Palsberg, J., and Vitek, J. Type-based confinement. In *Journal of Functional Programming* (2005), vol. 15(6), pp. 1–46.

Department of Computer and Systems Sciences

Stockholm University/KTH

http://www.dsv.su.se/eng/publikationer/index.html

Ph.D. theses

No 91-004 **Olsson, Jan**
An Architecture for Diagnostic Reasoning Based on Causal Models

No 93-008 **Orci, Terttu**
Temporal Reasoning and Data Bases

No 93-009 **Eriksson, Lars-Henrik**
Finitary Partial Definitions and General Logic

No 93-010 **Johannesson, Paul**
Schema Integration Schema Translation, and Interoperability in Federated Information Systems

No 93-018 **Wangler, Benkt**
Contributions to Functional Requirements Modelling

No 93-019 **Boman, Magnus**
A Logical Specification for Federated Information Systems

No 93-024 **Rayner, Manny**
Abductive Equivalential Translation and its Application to Natural-Language Database Interfacing

No 93-025 **Idestam-Almquist, Peter**
Generalization of Clauses

No 93-026 **Aronsson, Martin**
GCLA: The Design, Use, and Implementation of a Program Development

No 93-029 **Boström, Henrik**
Explanation-Based Transformation of Logic programs

No 94-001 **Samuelsson, Christer**
Fast Natural Language Parsing Using Explanation-Based Learning

No 94-003 **Ekenberg, Love**
Decision Support in Numerically Imprecise Domains

No 94-004 **Kowalski, Stewart**
IT Insecurity: A Multi-disciplinary Inquiry

No 94-007 **Asker, Lars**
Partial Explanations as a Basis for Learning

No 94-009 **Kjellin, Harald**
A Method for Acquiring and Refining Knowledge in Weak Theory Domains

No 94-011 **Britts, Stefan**
Object Database Design

No 94-014 **Kilander, Fredrik**
Incremental Conceptual Clustering in an On-Line Application

No 95-019 **Song, Wei**
Schema Integration: Principles, Methods and Applications

No 95-050 **Johansson, Anna-Lena**
Logic Program Synthesis Using Schema Instantiation in an Interactive Environment

No 95-054 **Stensmo, Magnus**
Adaptive Automated Diagnosis

No 96-004 **Wærn, Annika**
Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction

No 96-006 **Orsvärn, Klas**
Knowledge Modelling with Libraries of Task Decomposition Methods

No 96-008 **Dalianis, Hercules**
Concise Natural Language Generation from Formal Specifications

No 96-009 **Holm, Peter**
On the Design and Usage of Information Technology and the Structuring of Communication and Work

No 96-018 **Höök, Kristina**
A Glass Box Approach to Adaptive Hypermedia

No 96-021 **Yngström, Louise**
A Systemic-Holistic Approach to Academic Programmes in IT Security

No 97-005 **Wohed, Rolf**
A Language for Enterprise and Information System Modelling

No 97-008 **Gambäck, Björn**
Processing Swedish Sentences: A Unification-Based Grammar and Some Applications

No 97-010 **Kapidzic Cicovic, Nada**
Extended Certificate Management System: Design and Protocols

No 97-011 **Danielson, Mats**
Computational Decision Analysis

No 97-012 **Wijkman, Pierre**
Contributions to Evolutionary Computation

No 97-017 **Zhang, Ying**
Multi-Temporal Database Management with a Visual Query Interface

No 98-001 **Essler, Ulf**
Analyzing Groupware Adoption: A Framework and Three Case Studies in Lotus Notes Deployment

No 98-008 **Koistinen, Jari**
Contributions in Distributed Object Systems Engineering

No 99-009 **Hakkarainen, Sari**
Dynamic Aspects and Semantic Enrichment in Schema Comparison

No 99-015 **Magnusson, Christer**
Hedging Shareholder Value in an IT dependent Business society—the Framework BRITS

No 00-004 **Verhagen, Henricus**
Norm Autonomous Agents

No 00-006 **Wohed, Petia**
Schema Quality, Schema Enrichment, and Reuse in Information Systems Analysis

No 01-001 **Hökenhammar, Peter**
Integrerad Beställningsprocess vid Datasystemutveckling

No 01-008 **von Schéele, Fabian**
Controlling Time and Communication in Service Economy

No 01-015 **Kajko-Mattsson, Mira**
Corrective Maintenance Maturity Model: Problem Management

No 01-019 **Stirna, Janis**
The Influence of Intentional and Situational Factors on Enterprise Modelling Tool Acquisition in Organisations

No 01-020 **Persson, Anne**
Enterprise Modelling in Practice: Situational Factors and their Influence on Adopting a Participative Approach

No 02-003 **Sneiders, Eriks**
Automated Question Answering: Template-Based Approach

No 02-005 **Eineborg, Martin**
Inductive Logic Programming for Part-of-Speech Tagging

No 02-006 **Bider, Ilia**
State-Oriented Business Process Modelling: Principles, Theory and Practice

No 02-007 **Malmberg, Åke**
Notations Supporting Knowledge Acquisition from Multiple Sources

No 02-012 **Männikkö-Barbutiu, Sirkku**
SENIOR CYBORGS—About Appropriation of Personal Computers Among Some Swedish Elderly People

No 02-028 **Brash, Danny**
Reuse in Information Systems Development: A Qualitative Inquiry

No 03-001 **Svensson, Martin**
Designing, Defining and Evaluating Social Navigation

No 03-002 **Espinoza, Fredrik**
Individual Service Provisioning

No 03-004 **Eriksson-Granskog, Agneta**
General Metarules for Interactive Modular Construction of Natural Deduction Proofs

No 03-005 **De Zoysa, T. Nandika Kasun**
A Model of Security Architecture for Multi-Party Transactions

No 03-008 **Tholander, Jakob**
Constructing to Learn, Learning to Construct—Studies on Computational Tools for Learning

No 03-009 **Karlgren, Klas**
Mastering the Use of Gobbledygook—Studies on the Development of Expertise Through Exposure to Experienced Practitioners' Deliberation on Authentic Problems

No 03-014 **Kjellman, Arne**
Constructive Systems Science—The Only Remaining Alternative?

No 03-015 **Rydberg Fåhræus, Eva**
A Triple Helix of Learning Processes—How to cultivate learning, communication and collaboration among distance-education learners

No 03-016 **Zemke, Stefan**
Data Mining for Prediction—Financial Series Case

No 04-002 **Hulth, Anette**
Combining Machine Learning and Natural Language Processing for Automatic Keyword Extraction

No 04-011 **Jayaweera, Prasad M.**
A Unified Framework for e-Commerce Systems Development: Business Process Patterns Perspective

No 04-013 **Söderström, Eva**
B2B Standards Implementation: Issues and Solutions

No 04-014 **Backlund, Per**
Development Process Knowledge Transfer through Method Adaptation, Implementation, and Use

No 05-003 **Davies, Guy**
Mapping and Integration of Schema Representations of Component Specifications

No 05-004 **Jansson, Eva**
Working Together when Being Apart—An Analysis of Distributed Collaborative Work through ICT from an Organizational and Psychosocial Perspective

No 05-007 **Cöster, Rickard**
Algorithms and Representations for Personalised Information Access

No 05-009 **Ciobanu Morogan, Matei**
Security System for Ad-hoc Wireless Networks based on Generic Secure Objects

No 05-010 **Björck, Fredrik**
Discovering Information Security Management

No 05-012 **Brouwers, Lisa**
Microsimulation Models for Disaster Policy Making

No 05-014 **Näckros, Kjell**
Visualising Security through Computer Games Investigating Game-Based Instruction in ICT Security: an Experimental approach

No 05-015 **Bylund, Markus**
A Design Rationale for Pervasive Computing

No 05-016 **Strand, Mattias**
External Data Incorporation into Data Warehouses

No 05-020 **Casmir, Respickius**
A Dynamic and Adaptive Information Security Awareness (DAISA) approach

No 05-021 **Svensson, Harald**
Developing Support for Agile and Plan-Driven Methods

No 05-022 **Rudström, Åsa**
Co-Construction of Hybrid Spaces

No 06-005 **Lindgren, Tony**
Methods of Solving Conflicts among Induced Rules