Object Ownership & Containment

David Clarke

A dissertation submitted to the
School of Computer Science and Engineering
University of New South Wales, Australia
in fulfilment of the requirements for the degree of
Doctor of Philosophy

July 2001. Revised October 2002.

Abstract

Object-oriented programming relies on inter-object aliases to implement data structures and other abstractions. Objects have mutable state, but it is when mutable state interacts with aliasing that problems arise. Through aliasing an object's state can be changed without the object being aware of the changes, potentially violating the object's invariants. This problem is fundamentally unresolvable. Many idioms such as the Observer design pattern rely on it. Hence aliasing cannot be eliminated from object-oriented programming, it can only be managed.

Various proposals have appeared in the literature addressing the issue of *alias management*. The most promising are based on *alias encapsulation*, which limits access to objects to within certain well-defined boundaries. Our approach called *ownership types* falls into this category. An object can specify the objects it owns, called its *representation*, and which objects can access its representation. A type system protects the representation by enforcing a well-defined *containment invariant*.

Our approach is a formal one. Ownership types are cast as a type system using an minor extension to Abadi and Cardelli's object calculus with subtyping. With this formalisation we prove the soundness of our ownership types system and demonstrate that well-typed programs satisfy the containment invariant. In addition, we also provide a firm grounding to enable ownership types to be safely added to an object-oriented programming language with inheritance, subtyping, and nested classes, as well as offering a sound basis for future work. Our type system can model aggregate objects with multiple interface objects sharing representation and friendly functions which access multiple objects' private representations, among other examples, thus overcoming weaknesses in existing alias management schemes.

Sydney-Belgrade-Utrecht. July 1998-July 2001. Revised October 2002.

CERTIFICATE OF ORIGINALITY

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of a university or other institute of higher learning, except where due acknowledgement is made in the text.

Signed.
David Clarke

Acknowledgements

To my supervisors John Potter and James Noble. To John for his insightful comments, helpful guidance, and for having enough faith in me to somehow arrange that I always had money to buy books and sometimes a little food. To James for his constant wacky enthusiasm and for amazingly obtuse comments — "just use a bit to track it" — which subsequently turned out to be true, years after he first said them, and for his detailed comments on the chapters herein.

To the people of Microsoft Research Institute who helped smooth out the ideas and technical issues along the way: Michael Richmond, Ryan Shelswell, Ian Joyner, John Tidswell, Xiaogang Zhang, Geoff Outhred, and David Holmes — which pretty much covers everyone.

To the members of the Software Engineering Research group at the University of New South Wales, especially Ron van der Meyden who provided a few general hints which enabled me to hurdle a major obstacle, and Manuel Chakravarty for playing unofficial supervisor on demand and sharing my interest in functional programming.

To Sophia Drossopoulou and Alex Buckley for forcing me to improve my explanations through many confusing email discussions. To Sophia again for inviting me all the way to London to discuss Ownership Types and for her comments on Chapter 6.

To Christine Walter for encouraging me to make a detailed plan, which, although I followed it for only a week, was enough to save me from a slump which had crippled my progress.

To my housemate Bianca Deguara for putting up with my façade of an obsessive compulsive, schizophrenic, manic depressive. I blame the dissertation contained herein.

To my constant friends: to Kim Felmingham for a shared passion for books and writing, to Adi Iskandar for politics and dissidence, to Michelle Walter for always being there when I needed her, to Toby Cogley for fitness and gear, to John Domeney for his humility and interest in films, to Steve and Catherine Kelly for being fun and funny, to Jason Rutter for science talk, to Mark Dras for great lunchtime discussions and the pleasure of minding his house, to Radica Kevrešan for many emails and the invitation to visit Belgrade, to Pete Gammie for coffee, beer and cynicism, and of course to Milly-the-dog for being the exemplar of obsession.

To Boško, Julka, Radica and Stojan Kevrešan for their generous hospitality, food, pivo and rakija during my write-up in Belgrade.

Special thanks go to my parents for seeing me through school the last two and a half decades. It's high time I found gainful employment.

Finally, to the good people at Utrecht University for giving me a job.

Contents

List of Figures

Chapter 1

Introduction

1.1 Alias Management with Object Encapsulation

"Aliasing is endemic in object-oriented programming," quoth Noble, Vitek, and Potter [150]. A feature common in both the design and implementation of object-oriented programs [84, 29, 174], aliasing is indispensable; without it object-oriented languages would be crippled. Something as simple as a doubly-linked list would be impossible. When ill-used, however, aliasing breaks the encapsulation necessary for building reliable software components [130]. It is in combination with mutable state that aliasing problems arise. When an aggregate object's mutable state is externally accessible via an alias, the aggregate's state may unwittingly or otherwise change, potentially violating the aggregate's invariants. Consequently, object-oriented programs become both difficult to understand and reason about [150]. Removing one of the problem's constituents is not a feasible solution — the Observer pattern [84], to take one very common example, relies on the interaction between aliasing and mutable state. Because aliasing cannot be eliminated, it must instead be managed — according to the Geneva Convention on the Treatment of Object Aliasing [110], aliasing must be either *advertised* explicitly in syntax to aid modularity, *prevented* in a statically checkable fashion, or *controlled*, that is, isolated so that it effects only small parts of a program. Theisen [185] further suggests that an alias management scheme should be *practical*, preserving most of the expressiveness of reference-based sharing, *simple* to understand and use, *generally applicable* to all object-oriented languages, and *efficient*, requiring exclusively compile-time checks. In addition, we contend that an alias management scheme must be *precisely specified* and *sound*, that is, the annotations must unequivocally and correctly report aliasing which may occur.

To date a variety of approaches to managing aliasing have been proposed. Each addresses some combination of the following features:

- the uniqueness of object reference [109, 141, 10, 150, 32, 94, 126];
- the extent of aliasing [109, 10, 150, 125, 61, 67, 24, 144, 94];
- computational effects such as the reading and writing of objects or otherwise specified groups of objects [109, 10, 150, 125, 144, 94, 122, 185]; or
- the dependence on mutable state [150, 185].

In this dissertation, we concentrate on the advertisement and control of the extent of aliasing. In particular, we formalise schemes which provide *object* or *alias encapsulation* [150]. Such schemes restrict objects references to within certain universes (packages, classes, or objects), which generally correspond to the pre-existing units of encapsulation within the object-oriented paradigm.

One may imagine that some sort of *privacy* annotation, such as those which exist in Java [93] and C++ [72], provides the desired machinery. Privacy, however, protects only the names of fields, methods, and sometimes classes, hiding the implementation details from the clients of a class. It does not protect the objects which make up that implementation. Even Eiffel, which offers finer control over attribute export policies, does not protect the objects in the fields [138]. Formal descriptions of object-oriented languages which use bounded existential types [78, 159] or subsumption [171, 3] to model privacy and related behaviour also suffer from the same problem. In each case the objects which are considered private can be exported via some other means, for example, by a public method or using some (sub)type which is not hidden. The problem is that the privacy information is not a part of an object's type. This information can be lost and hence subverted.

Now let's examine this a little more closely. An object type is generally either a class name or a record of field and method types. Subsumption either takes a class to its superclass or allows fields and methods of record to be omitted. Although subtyping can hide the details of the implementation type, it can also, however, allow the type of the hidden data to be converted into a publicly visible type (such as `Object` in Java), which allows the encapsulated objects to be exported. Instead of protecting the type and other details of how the private parts, or *representation*, are implemented, we must also protect the objects which make up the implementation. The key to achieving this is to *annotate types with protection information which is preserved through*

subtyping. Indeed, a number of proposals which appear in the literature adopt this approach:

- *Confined types* [24] modifies class types using package-level information to prevent objects from certain classes from being accessed outside a package;

- *Universes* [144] modifies object types using class-level information to prevent certain objects from being accessed outside of certain classes; and

- *Flexible Alias Protection* [150], *Universes* [144], *Ownership Types* [61] (and this thesis) modify object types using object-level information to prevent certain objects from being accessed outside certain other objects.

All of these approaches adopt some form of *ownership* and some form of *containment*. Firstly, a collection of universes is used to partition the realm of types and thus partition objects. This form the basis for ownership. The universes can be considered to be the owners of objects. Secondly, the universes are nested in some way, whether it be the shallow nesting corresponding to a partition, perhaps with some global element added, or something deeper. This nesting is then used to control which references are allowed. This is the basis for containment.

What is lacking from most of the alias management proposals above, and indeed from many appearing in the literature, is a proper semantic description of the annotations employed and the properties they entail. This makes it impossible to verify the claims made about each system. Our approach heeds this semantic gap and takes measures to close it. We work within a type theoretic framework by extending Abadi and Cardelli's object calculus with a notion of ownership. We use the resulting formalism to explore the properties of a few models of containment, thus providing a type theoretic description of object encapsulation. Although Abadi and Cardelli's object calculus lacks classes, these can easily be encoded within the calculus, and we do so for the extensions presented here. The general approach we take enables correctness proofs in a general setting, but also offers a framework for a principled exploration of different containment models and more advanced alias management schemes. Thus we lay foundations so that the future should see a more private `private` (See Heller [101]!)

1.2 Contributions

This dissertation makes the following contributions:

We adopt a model of containment which enforces a globally specified containment property over object graphs. This uses a locally defined constraint to determine the validity of a reference from one object to another. This constraint can be incorporated into a type system. Before doing so, we generalise the containment model by separating objects from their owners by introducing *contexts* as the units of ownership. More flexibility is gained by separating an object's owner (which governs access to the object) from its representation context (which governs which objects the object can access).

We introduce a system of *ownership types* which is distilled from *flexible alias protection* [150] and incorporates our containment model. Firstly we use a statically defined collection of contexts, and then, by introducing an operation which allows contexts to be created dynamically, we allow each object to have its own representation context and thus own its own private implementation objects. We also present a number of variations within these models. The underlying calculus is based on Abadi and Cardelli's object calculus, and the type system uses mostly standard type-theoretic constructs, although a number of innovations were devised to overcome certain difficulties to enable a sound system to be constructed. We also prove properties of such as soundness of the type system and that objects resulting from a well-typed expression satisfy the constraints underlying the containment model.

We identify features and examples which could not be handled in previous work and show how they appear in our calculus. These include subtyping and extensions to allow object-oriented idioms such as iterators, objects with multiple interfaces, friendly functions which access multiple objects' encapsulated objects, and the initialisation of an object's representation with externally specified objects.

Finally, we apply the abovementioned developments to a class-based programming language which resembles Java.

The result is a practical, flexible, and powerful formal system which not only models many of the features of other systems presented in the literature, allowing for the efficient implementation of complex abstractions while preventing the undesirable effects of aliasing, but it also provides a sound basis for future developments in encapsulation-based alias management, and for other schemes which employ object encapsulation.

1.3 Outline

The dissertation is structured as follows.

Chapter 2 reviews the problem of aliasing in object-oriented programming, before focusing mainly on proposals which address this problem with some form of alias encapsulation. We also review the theoretical material on which our work is based.

In Chapter 3 we present our model of containment. Starting with a graph-theoretic description of containment, we derive and generalise an invariant which enforces this condition, and is defined in such a way that it can be incorporated into a type system.

Chapter 4 presents a key extension to Abadi and Cardelli's object calculus, which enables a model of containment with a fixed collection of object owners. It is not only expressive enough to capture the essence of some proposals in the literature, it also serves as a stepping stone to the main calculus studied herein.

The formal development peaks in Chapter 5. By introducing an operation which creates object owners at run-time, and by using existential types to hide the representation contexts of objects, we produce a calculus which models object ownership where each object can own its own collection of private objects. We demonstrate the soundness of the calculus and prove that well-typed programs satisfy the containment invariant developed in Chapter 3.

Chapter 6 presents examples illustrating the calculus at work, including an extension of a class-based programming language with ownership types. A deficiency in the expressiveness of the calculus is also noted and a number of solutions are outlined.

Chapter 7 concludes and discusses future directions which the work presented here could be taken.

Appendix A contains the uninteresting details of both important and non-trivial results.

Appendix B contains a proof relating our containment invariant to the graph theoretic property it enforces.

Chapter 2

Background

Our overall aim is to formalise an alias management scheme for object-oriented programming languages based on object encapsulation. In its strongest form the units of encapsulation are aggregate objects and the encapsulated objects constitute an aggregate's private representation. Before we achieve this end, we establish a suitable setting by briefly describing the features of object-oriented programming and the work done in their formalisation. We focus in particular on Abadi and Cardelli's object calculus [3], which we adopt as the formal basis for our work. Essential to our proposal, but lacking in these formalisms, are the notions of aggregation and object encapsulation. We discuss these further, establishing our view of objects as aggregates, and describe the relationship with encapsulation.

Next we focus on the problems of aliasing, which are only manifest in the presence of computational effects, particularly the modification of mutable state. Aliasing is an important but problematic issue across a broad spectrum of computer science. Its spectrum forms an important source of ideas, problems, techniques and solutions, which we review in some detail, before focusing on the alias management schemes previously proposed for object-oriented systems, including the direct precursors to this work Flexible Alias Protection [150] and the original Ownership Types system [61].

2.1 Object-Oriented Programming

Object-oriented programming is a wide spread programming paradigm which focuses on the development of reusable software. Underlying object-oriented programming are key notions such as *objects*, *classes*, *inheritance*, and *dynamic binding* which

work together to achieve this goal [137]. There are roughly three approaches to object-oriented programming, each taking a different focus: prototype-based, where the object is the focus, class-based, where class is the focus, and approaches based on multimethods, where the method or dynamic binding is the focus. We now cover these aspects.

Objects and Prototype-based Programming Computation in an object-oriented program involves the manipulation of *objects*. An object consists of some internal state and a collection of methods which are pieces of code that operate on it. The internal state consist of a collection of *instance variables* or *fields* which contain primitive data such as integers or references to other objects. Methods are evaluated by *method selection*, sometimes called *message send*, which indicates which method to evaluate for a given *target* object. For example, consider the following two-dimensional point object, corresponding to the point $(10, 21)$:

Internal State	
x	10
y	21
Method Suite	
bump	*code for bump*

The fields of this object x and y store the values 10 and 21 respectively. If this object is stored in variable p, then the code $p.bump$ selects the *bump* method and executes its code. This could alternatively be described as sending the *bump* message to p. The code for the *bump* method can modify the contents of fields x and y. Keeping the data and the operations together like this is a key feature of object oriented programming which, arguably, enables better modelling of real world phenomena [134]. Cardelli calls this approach *objects-as-records* [46].

Programming languages can be based purely on objects, allowing the construction, modification, cloning and extension of objects directly. These are dubbed *prototype-based* programming languages [128, 191, 182, 149]. One of their key operations is method update, which allows the method of an object to be changed. Abadi and Cardelli argue that prototype-based languages are more fundamental than other object-oriented programming languages, because many object-oriented features such as classes can be modelled with just objects [3]. We will see how prototype-based programming languages behave when we examine Abadi and Cardelli's object calculus in

Section 2.2.1. Abadi and Cardelli's object calculus is the formalism which underlies the work contained in this dissertation.

Classes, Inheritance and Class-based Programming Objects are created using *classes*. A class contains both the fields the generated objects will have and the code for its methods. In a typed programming language, the types of the fields and the method's formal parameters and return values are specified and used to ensure that the field or method selected is present in an object. For example, the following is the class of points in 2-d space:

```
class Point {
  Point(int x_init, int y_init) { x = x_init; y = y_init; }
  int x
  int y

  bump() { x = x + 1; y = y - 1 }
}
```

The method *bump* updates the object in-place. Thus objects are stateful and object-oriented programming languages are imperative.

Within a method the target object of the method invocation can be referred to using the keyword `self` [89] or `this` [93, 72], or using a variable designated expressly for that purpose [165, 44]. Thus the *bump* method could have been written equivalently as

```
self.x = self.x + 1; self.y = self.y - 1.
```

An object can also use the reference to *self* to pass itself to other methods.

Each class also contains one or more methods for creating elements of that class — these generally take the name of the class, such as `Point` in the example above. An object of a class is created by calling the constructor with some arguments which are used to initialise the new object's fields. For example, the code `new Point(10,21)` would create a point object resembling the one depicted above. A constructor can be called any number of times, with potentially different arguments, and each time a new object is created. Classes can be thought of as templates for creating objects.

Mainstream object-oriented programming languages such as C++ [72], Java [93], Eiffel [138], and Smalltalk [89] are based on classes.

In object-oriented programming the principal means for reusing code is *inheritance* or *subclassing*.[1] Rather than writing each class from scratch, inheritance allows one class to *inherit* the features of another class, in the process *extending* the class with new fields and methods, and *overriding* any methods whose functionality is not appropriate for the new class. For example,

```
class ColouredPoint extends Point {
  ColouredPoint(int x_init, int y_init, Colour c_init)
    { super(x_init,y_init);
      colour = c_init }
  Colour colour

  bump() { super.bump();
           colour.change() }
}
```

This code declares a new class representing coloured points. It states that its implementation extends or inherits the implementation of `Point` by adding a new constructor and a new field, `colour` which is an object from class `Colour`, and by overriding the method `bump`. This new method firstly calls the `bump` method from class `Point`, using `super.bump()`, then calls the method `change` on its own colour field. This class is shorter than it would have been if inheritance was unavailable. And the benefits multiply when a program is more that a few lines of code. We call `Point` a *superclass* of `ColouredPoint`, and, dually, `ColouredPoint` a *subclass* of `Point`.

In general inheritance provides more than just code reuse.[2] Subclassing induces a relation between classes which allows objects from the subclass to be used where an object from the original class is expected. For example, a function `flipX(Point p)` which takes a point and flips it around the x-axis (using `y = -y`) can also take an object from class `ColouredPoint` as its argument. This is called *subtype polymorphism*.

A class need not inherit from just one class. Multiple inheritance allows the inheritance of attributes from more than one class. While this introduces more opportunities for reuse, it introduces problems when there are conflicts and duplications among the

[1] Subclassing refers to process of incrementally extending classes with additional features, whereas inheritance is the sharing of features between the existing and new classes. The distinction is not important here.

[2] Meyer presents a taxonomy of various kinds of inheritance [137], but programming languages cannot generally express these distinctions.

attributes [137]. These problems have forced Java to include only single inheritance [93], though it also has interfaces which give the benefits of subtyping without the problems of multiple inheritance. Some authors even go as far as questioning whether inheritance is even worth having at all. [195]. But these issues are beyond our concern.

Dynamic binding The final key feature of object-oriented programming is *dynamic binding*. This means that the method which is actually called when a method is selected depends on the class of the receiver, rather than on the type of the variable as specified in the code.

For example, consider the following method.

```
doBump(Point p) {
  p.bump();
}
```

The type of the variable `p` is `Point`. Assume that the object `cp` is from class `ColouredPoint` and that we make the call `doBump(cp)`. Under dynamic binding the method which will be called when statement the `p.bump()` is evaluated is `bump` method from `ColouredPoint`.

Dynamic dispatch is important, because it selects the method which has been specialised in the class of a given object. This creates more opportunities to reuse code. Prototype-based takes the specialisation that dynamic binding provides even further, because it allows the collection of methods for individual objects to be updated.

In both cases just mentioned, the method selected under dynamic binding depends on the object receiving the message. A different approach is taken by object-oriented programming languages based on multimethods. These select a method based on the dynamic type of all arguments rather than just the receiver of a method. Castagna studies their formal properties [54], and we discuss them no further.

Aggregation, Ownership and Containment This thesis deals with alias protection in a manner which parallels the notion of *aggregation*. Briefly, aggregation is the construction of aggregate objects from smaller objects, which themselves may be aggregates. These objects are then considered to be a part of the aggregate. From this we derive a notion of *ownership*: an aggregate object owns the objects which are a part of it. Aggregation induces a nesting relationship among objects. To preserve the

invariants of an aggregate, we wish to prevent access to the objects inside an aggregate from outside of the aggregate. This is called *containment*. We cover these issues in more detail throughout this thesis.

The approach taken in this thesis is formal and based on the existing foundations of object-oriented programming languages.

2.2 Foundations of Object-Oriented Languages

The seminal work of Cardelli [45, 46] precipitated the study of the foundations of object-oriented languages. Foundational studies in object-oriented programming aim to provide abstract accounts of how object-oriented languages are and how they could or should be. Research deals with the specification of static type systems for various object-oriented features such as classes, objects, inheritance, subtyping, overloading, self-type, and so forth; with type checking and inference; and with the dynamic behaviour of object-oriented systems, such as the semantics of late-binding. This research has uncovered flaws in programming languages, such as those in Eiffel's ambitious but unsound type system [63], and has suggested fixes [63, 177], as well as guiding safe extensions to existing programming languages [153, 34, 165, 79].

Formal models appear at various levels of abstraction, deal with particular programming languages, individual features, or combinations and extensions thereof. Models are based on recursive records [53, 64, 71, 35], existential types [159, 108], primitive objects [3, 76], primitive classes [28, 78], overloaded functions [54], F-Bounded Polymorphism [43], or process calculi [113, 112], just to present a sample. Comparisons of various approaches exist [37, 77]. Features investigated include binary methods [36], multimethods and overloading [54, 56], mixins [81, 27], state [71], inheritance [64, 79, 164, for example], virtual types [116, 166, 189], method update/overriding/specialisation [3, 76, 75], object extension [76, 129, 26], primitive objects [2, 4, 3], privacy [171], classes [81, 28, 35], inner classes [117], MyType [35], and so forth. Programming languages which have been treated in detail include Java [70, 146, for example], Eiffel [63, 177], Smalltalk [197], Cecil [55, 56], Objective Caml [165], and Modula-3 [2]. A handful of books on the area also exist: Palsberg and Schwartzbach's *Object-Oriented Type Systems* [155]; the collection *Theoretical Aspects of Object-Oriented Programming* [98]; Adadi and Cardelli's *Theory of Objects* [3]; and Castagna's *Object-Oriented Programming: A Unified Approach* [54].

As far as we can determine none of these works explore aggregation, containment,

or ownership, nor do they address alias management. Perhaps the reason is that these notions restrict the programs which can be written, as they impose additional safety criteria beyond type safety, whereas the research mentioned above aims for as much expressiveness as possible give the features under consideration while retaining type safety.

Ownership and containment are largely orthogonal to the features considered in the research cited above. These features concern only objects and we are concerned with the structure of object graphs. Thus we would like to describe them in a formalism which has just objects. For this reason we select Abadi and Cardelli's object calculus [3] as our starting point. Other formalisms tend to have too much type machinery in their most basic form. In its most primitive form, the object calculus has just one kind of type, that of objects, and three syntactic constructs, objects, method selection, and method update. It is expressive enough to encode features such functions, classes, etc [3]. Taking another formalism as our base would have required extensions similar to the ones we make, presenting similar challenges along the way.

We now review Abadi and Cardelli's object calculus in more detail.

2.2.1 Abadi and Cardelli's Object Calculus

Abadi and Cardelli's object calculus [3], which we will refer to as "the object calculus", even though there are others, is a foundational calculus which provides semantics and typing rules for many object-oriented concepts such as objects, classes, prototypes, self type, inheritance, binary methods, and so on. Rather than presenting a single calculus with a single type system, the exhaustive *Theory of Objects* [3] presents a wealth of variations of the same basic calculus. The calculi increase in complexity: untyped, first-order typed, imperative, second-order typed, and higher-order typed. The more complex versions model more object-oriented features more accurately. We situate the final calculus developed in this thesis somewhere in the middle, choosing an imperative calculus with some second-order features, namely parameterised types.

The object calculus has objects not classes as its most primitive construct. Classes can be modelled using objects. The two basic operations are method select and method update, which makes the object calculus prototype-based, rather than class-based. The untyped, functional object calculus is the simplest presented in Chapter 6 of *A Theory of Objects* [3]. It has the following syntax:

$$
\begin{array}{rcll}
a,b & ::= & x & \textit{variable} \\
 & | & [l_i = \varsigma(x_i)b_i^{\,i\in 1..n}] & \textit{object formation } (l_i \textit{ distinct}) \\
 & | & a.l & \textit{field select/method invocation} \\
 & | & a.l \Leftarrow \varsigma(x)b & \textit{field update/method update}
\end{array}
$$

An object is a collection of methods with labels l_1 to l_n. Each method has the form $\varsigma(x)b$, where the parameter x is the self parameter used to refer to the target object within the method body b. The semantics are defined in part using the following reduction relation, where $o \equiv [l_i = \varsigma(x_i)b_i^{\,i\in 1..n}]$:

$$
\begin{array}{rcll}
o.l_j & \rightsquigarrow & b_j\{^{o}/_{x_j}\} & (j \in 1..n) \\
o.l_j \Leftarrow \varsigma(y)b & \rightsquigarrow & [l_j = \varsigma(y)b, l_i = \varsigma(x_i)b_i^{\,i\in(1..n)-\{j\}}] & (j \in 1..n)
\end{array}
$$

The notation $b_j\{^{o}/_{x_j}\}$ denotes a capture-avoiding substitution of o for variable x_j in expression b_j — this can be defined by noting that the only place where variables are bound is in the syntax $\varsigma(y)b$ which restricts the scope of y to the expression b.

Method invocation $o.l_j$ proceeds by selected the body b_j corresponding to the method label l_j, then substituting the object o for the self parameter x_j in b_j, and continues by reducing the this expression.

Method update $o.l_j \Leftarrow \varsigma(y)b$ reduces to a copy of the target object where the method in l_j is replaced by $\varsigma(y)b$. It results in a copy of the original object because the calculus we are considering now is functional.

The definition of reduction is completed by allowing subexpression within an expression to be reduced.

Here is an example reduction. The expression $[l = \varsigma(x)x.l \Leftarrow \varsigma(y)x].l$ reduces as follows:

$$
\begin{array}{rcl}
[l = \varsigma(x)x.l \Leftarrow \varsigma(y)x].l & \rightsquigarrow & [l = \varsigma(x)x.l \Leftarrow \varsigma(y)x].l \Leftarrow \varsigma(y)[l = \varsigma(x)x.l \Leftarrow \varsigma(y)x] \\
 & \rightsquigarrow & [l = \varsigma(y)[l = \varsigma(x)x.l \Leftarrow \varsigma(y)x]]
\end{array}
$$

Common object-oriented constructs can easily be modelled within the object calculus. Fields are modelled as methods which are values that make no reference to the self parameter. Field update employs method update, where again the new method is a value which makes no reference to the self parameter. A function is modelled as an object containing a field to store the argument to the function. Methods with arguments are modelled as methods which return a function, and so forth. A more complete account can readily be found in *A Theory of Objects* [3].

Typing this small fragment is quite simple, as there is only one type:

$$A,B \quad ::= \quad [l_i : B_i{}^{i\in 1..n}] \qquad \textit{object type } (l_i \textit{ distinct})$$

This is the type of an object with methods l_1 to l_n having types B_1 to B_n, respectively. The type system depends on a typing environment E which gives the types for variables appearing free in the expression being typed. The typing rules for expressions are:

$$\frac{\text{(Val Object)} \;\; \text{where } A \equiv [l_i : B_i{}^{i\in 1..n}] \qquad E, x_i : A \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i:A) b_i{}^{i\in 1..n}] : A}$$

$$\frac{\text{(Val Select)} \qquad E \vdash a : [l_i : B_i{}^{i\in 1..n}] \quad j \in 1..n}{E \vdash a.l_j : B_j} \qquad \frac{\text{(Val Update)} \;\; \text{where } A \equiv [l_i : B_i{}^{i\in 1..n}] \qquad E \vdash a : A \quad E, x : A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A) b : A}$$

(We have modified the untyped syntax slightly to include a type for the method's self parameter.)

The rule (Val Object) states that an object of type $[l_i : B_i{}^{i\in 1..n}]$ is well-formed when each method body b_i has type B_i when typed against an environment extended with the self parameter x_i of type $[l_i : B_i{}^{i\in 1..n}]$. The rule (Val Select) states that selecting the method l_j from an object of type $[l_i : B_i{}^{i\in 1..n}]$ produces a result of type B_i. Finally, the rule (Val Update) checks that the new method has the expected type in the same way methods are checked in (Val Object). The result has the type of the object being updated.

Subtyping will be dealt with in subsequent chapters. In its most basic form, it allows us to forget some of the methods present in an object.

In an imperative object calculus objects are held in a store which is a map from locations to objects. The results of computations are locations, the only values in the imperative version of the calculus presented thus far. To add imperative features the syntax is extended as follows:

$$\begin{array}{rcll} a,b & ::= & \ldots & \\ & | & \iota & \textit{location} \\ \\ \sigma & ::= & \emptyset & \textit{empty store} \\ & | & \iota \mapsto [l_i = \varsigma(x_i) b_i{}^{i\in 1..n}], \sigma & \textit{location-object binding} \end{array}$$

The semantics of this imperative object calculus can be expressed in a number of ways. Abadi and Cardelli originally used a store which contained methods rather than objects [3]. Gordon, Hankin and Lassen present a number of equivalent semantics in different styles using a store containing objects [90]. We adopt their big-step, substitution-based operational semantics. Yet another presentation of the imperative object calculus (with concurrency) integrates stores and expressions [92].

The semantics is defined as a relation between an initial configuration and a final configuration, $(\sigma_0, a) \Downarrow (\sigma_1, \iota)$, which states that evaluating expression a with store σ_0 produces result ι with store σ_1, when the computation terminates. The relation is defined by the following rules:

(Subst Location)

$$\frac{}{(\sigma, \iota) \Downarrow (\sigma, \iota)}$$

(Subst Object)

$$\frac{\sigma_1 = \iota \mapsto [l_i = \varsigma(s_i) b_i{}^{i \in 1..n}], \sigma_0 \qquad \iota \notin \mathsf{dom}(\sigma_0)}{(\sigma_0, [l_i = \varsigma(s_i) b_i{}^{i \in 1..n}]) \Downarrow (\sigma_1, \iota)}$$

(Subst Select) where $j \in 1..n$

$$\frac{(\sigma_0, a) \Downarrow (\sigma_1, \iota) \quad \sigma_1(\iota) = [l_i = \varsigma(s_i) b_i{}^{i \in 1..n}] \quad (\sigma_1, b_j\{{}^{\iota}/_{s_j}\}) \Downarrow (\sigma_2, \iota')}{(\sigma_0, a.l_j) \Downarrow (\sigma_2, \iota')}$$

(Subst Update) where $j \in 1..n$

$$\frac{\begin{array}{c}(\sigma_0, a) \Downarrow (\sigma_1, \iota) \quad \sigma_1(\iota) = [l_i = \varsigma(s_i) b_i{}^{i \in 1..n}] \\ \sigma_2 = \sigma_1 + \iota \mapsto [l_i = \varsigma(s_i) b_i{}^{i \in 1..j-1, j+1..n}, l_j = \varsigma(s) b]\end{array}}{(\sigma_0, \iota.l_j \Leftarrow \varsigma(s) b) \Downarrow (\sigma_2, \iota)}$$

When a method is selected, its location (or object id) is substituted into the self parameter. Compare with the functional version which substitutes the original object for the self parameters. Method update updates the object in-place, rather than constructing a new object, as in the functional version.

The expression $[l = \varsigma(x) x.l \Leftarrow \varsigma(y) x].l$ evaluates as follows:

$$\frac{(\emptyset, [l = \varsigma(x) x.l \Leftarrow \varsigma(y) x]) \Downarrow (\sigma_0, \iota) \qquad \dfrac{\dfrac{(\sigma_0, \iota) \Downarrow (\sigma_0, \iota)}{(\sigma_0, \iota.l \Leftarrow \varsigma(y) \iota) \Downarrow (\sigma_1, \iota)}}{(\sigma_0, \iota.l) \Downarrow (\sigma_1, \iota)}}{(\emptyset, [l = \varsigma(x) x.l \Leftarrow \varsigma(y) x].l) \Downarrow (\sigma_1, \iota)}$$

where $\sigma_0 = \iota \mapsto [l = \varsigma(x)x.l \Leftarrow \varsigma(y)x]$ and $\sigma_1 = \iota \mapsto [l = \varsigma(y)\iota]$.

We use this style of operational semantics for the remainder of the thesis, but in a slightly different form which facilitates simpler proofs. But now we return to our main thread and discuss aliasing.

2.3 The importance of not being aliased

The interference between aliasing and mutable state has been a problem across the entire programming spectrum, from specification logics to optimisation in compilers. Aliasing is an issue in almost all programming languages; but it is in object-oriented programming where the suffering is most severe. Underlying many of the approaches which address this problem is the idea that memory can be partitioned into distinct areas and the observation that references into different areas cannot be aliases for the same object.

While the specification of object-oriented programs is possible, and indeed new tools and techniques are being invented all the time, such as Hoare and Jifeng's trace model which adopts ideas from process algebra [106], Abadi and Leino's sound specification logic for Abadi and Cardelli's object calculus [5], Hensel, Huisman, Jacobs and Tews approach based on co-algabra [103], one overriding criticism is that while these approaches are feasible in principle, they all suffer, because even on simple examples the constraint sets become large due to possible aliasing [5]. While proof assistants help manage this complexity [103], the greatest improvements seem to come when exploiting some partitioning of the memory.

In equi-named papers, Wills [196] and Utting [192], present techniques for reasoning in the presence of aliasing. Wills collects all objects which can affect the value of an object into the same conceptual entity, thus excluding all others objects, whereas Utting uses explicit local stores to simplify his reasoning. Leino uses data groups to collect together variables which may be modified during a method's evaluation, thus excluding all others. This is done in such a way that specifications are well-behaved in the presence of inheritance [125]. At a more fundamental level, though now moving away from object-oriented programming, is Reynolds' *Syntactic Control of Interference* (SCI) for Algol-like languages [167], which was the original proposal that allowed a programming language to state which variables did not interfere. SCI was unfortunately unsound in the presence of β-reduction, but was fixed [169] and simplified [154] many years later. Very recently logics for reasoning about shared mutable

data structures were demonstrated and validated [170, 31, 41, 118]. One of the exciting tools underlying this work is the *Logic of Bunched Implications* [118] which allows a store to be partitioned so that only the part modified by a statement need be considered when reasoning about the statement. The logic allows dangling pointers into parts of the store unaffected by the statement. A *frame axiom introduction* rule admits local reasoning: specifications in terms of store fragments can be lifted to larger stores enclosing those fragments. It is early days yet for Bunched Implications but the future looks bright, even though the programming languages underlying their specifications lack objects and subtyping and thus fall short of full-blown object-oriented languages.

A type system provides an abstract specification of program behaviour. It can be more expressive than the one presented to the programmer. Rather than adorning the programming language proper, types can annotate intermediate languages to express memory allocation and deallocation and aliasing properties [188, 66]. In the *Regions Calculus* memory is partitioned into *regions* which are allocated and deallocated in a stack-based manner. The results of evaluation are stored in regions and a type system ensures that regions which go out of scope can be safely deallocated, since the values the region contains will never be accessed. Early purveyors of these ideas include Baker [13], Lucassen and Gifford [132], and Talpin and Jouvelet [184], but their use has been pushed to a practical level in the ML-Kit which provides a Standard ML implementation without garbage collection [187, 188, 19]. The underlying inference algorithm, which is based on Hindley-Milner type inference [139], has been described in painful detail [186] and can be improved using set-based constraints [9]. One of the key features of the regions calculus is that it allows dangling pointers into deallocated memory to be safely present in terms, since the type system guarantees that the pointers are never dereferenced. This made the original proof of safety very complicated. Subsequent simplifications of the proof include treatments using an operational semantics stratified into high and low level semantics which each deal with different behavioural aspects [40], the π-calculus [201] and the polymorphic lambda calculus [17]. Ideas underlying regions have also been used explicitly to speed up dynamic memory allocation in C [86], and by O'Callahan and Jackson [151] in program understanding tools for reverse engineering.

More sophisticated systems which express more than a simple stack-based partition of memory include Crary, Morrisett and Walker's Calculus of Capabilities [66], Smith, Walker and Morrisett's [179] and later Walker and Morrisett's [193] Alias

Types. The type system underlying the Calculus of Capabilities can express non-lexically scoped region lifetimes, whereas in Tofte and Talpin's regions calculus, the regions lifetimes are lexically scoped, as well some simple alias information. Alias Types are expressive enough to represent pointer aliasing in the presence of recursive data structures, with explicit coercions allowing a form of subtyping. Interestingly these type systems seem also to allow the partitioning of stores on a par with the partitioning used in the work on Bunched Implications [118]. These type systems allow the verification of program properties such as whether a particular object is garbage. They are used as compiler intermediate languages and are very detailed, and thus are not themselves candidates for programming language extension.

Yates [199] specifies a regions system for encapsulating memory in Java, but it does not include regions for an object to contain its representation in, as we do here. His system starts with only a finite number of regions, but allows expressions to evaluate in new, local regions as in the original regions calculus [188]. This system seems to be less powerful than Greenhouse and Boyland's Object-Oriented Effects System, a system we discuss in more detail later in this chapter [94], though Yates does, however, present a complete formal development including a soundness proof.

The partitioning of memory is also important for security. Leroy and Rouaix rely on the implicit partitioning derived from the type-theoretic properties of typed applets [127], whereas Zdancewic, Grossman and Morrisett's Principals exploits the unknown nature of abstract types to demonstrate their separation properties [200]. Cardelli, Ghelli and Gordon introduce the new notion of a *group* which is a type that can be dynamically created. Apart from providing an account of the regions calculus [201], these have also been used to guarantee certain security properties in the π-calculus [49] and in the Ambient Calculus [48].

The final area where the partitioning of memory is important is in the alias analysis phase of a compiler's analyses. The literature covering this area is immense and it would be too much of a diversion to review it here. There are a number of interesting examples. Diwan, McKinley and Moss use the type hierarchy in the object-oriented language Modula-3 to infer that certain aliases cannot occur, using the idea that certain classes have non-overlapping instances. Their analysis says nothing about when references are of the same type [68]. Escape Analysis [87, 21, 21] infers when objects are allocated, used and done with within a method body, so that such objects can be allocated on the stack. The analysis ignores the deep structure present in aggregate objects. Genius, Trapp and Zimmermann [88] use a generalisation of Balloons, an

alias management scheme to be discussed in Section 2.5, as the target of a particular compiler analysis which exploits some of the implicit nesting between objects and places objects of the same type together in the heap and improves locality of reference. Unfortunately, the language they analyse does not include inheritance and their technique suffers whenever a class is reused.

A different approach to allowing the compiler to infer the aliasing properties is to specify in the syntax of the programming language hints which allow the partitioning or aliasing present in a program to be more easily inferred. Klarland and Schwartzbach do this using a second order monadic logic as the basis for their analyses of aliasing in dynamic pointer-based data-structures [121], and Hummel, Hendren and Nicolau [111] include annotations of a similar expressiveness to describe the shape of their pointer data structures. These proposals follow Hendren and Goa's *design for analysability* dictum [102], which requests that programming languages include hints to the compilers to assist in their efficient implementation. The proposals of this form are best for describing pointer data structures, but are less suitable for object-oriented programming, if design patterns are to be any indication of object-oriented practice [84], since the structure of object-oriented programs seems to be less predictable.

Many of the alias management proposals discussed later in Section 2.5 combine the ideas described above: they capture the partitioning of objects in a manner which parallels the existing encapsulation boundaries such as objects, classes, and packages; and they make the partitioning explicit in the programming language. Before describing these proposals we explore the problem of aliasing in relation to aggregate objects in object-oriented programs.

2.4 Aggregate Objects and the Effect of Aliasing

An object is too small to implement complex abstractions on its own. Rather it is the interaction and cooperation of collections of objects which gives object-oriented programming its power. An object refers to another using its *identity* [137], and the collection of objects which exist in a program form a graph, called the *object graph*. Rather than being a loosely connected collection of objects forming some arbitrary graph, there is generally some underlying structure grouping objects together. This structure can be derived from some notion of aggregate object. One view is that "the state of an object is not fully specified by just its variables, but also upon the states to

which these variables refer [110]." Thus an aggregate object could be the collection of all objects reachable from a given object. But this ignores the distinctions made when an object system is designed. Some objects are considered to be a *part of* an object's implementation, for example, an `Engine` is part of a `Car`, whereas other objects are merely *used* for some service which they provide, such as a `Service Station` object. The relationship between an object and the objects it uses is called *association* and the relationship with the objects which are a part of its implementation is called *aggregation* [29, 174, 30].

We are most interested in the latter relationship. An object together with its implementation is referred to as an *aggregate object*, or simply an *aggregate*. The objects which are a part of its implementation are considered to be *inside* the aggregate. We consider an object to be the *owner* of its implementation objects. We call these implementation objects its *representation*. Thus an aggregate consists of an object plus its representation. Generally, the representation is accessible only to the aggregate and not beyond the aggregate's boundary.

This view of aggregation suggest that objects have a single interface.[3] While it is possible for an object's class to use multiple inheritance to inherit features from multiple sources, the object will still present a single (monolithic) interface to its clients. In some instances this view is too limited. We adopt the stance that *an aggregate object can have multiple interfaces, each of which can be a separate object, which may itself be another aggregate*. Each interface belongs to the aggregate and has access to the aggregate's implementation objects, but is also accessible outside of the aggregate.

An example will help illustrate these ideas.

Consider the instance of Java's `Vector` class depicted in Figure 2.1 [93]. A `Vector` consists of an `Array` which is a part of the implementation of the `Vector` aggregate. It is not directly accessible to objects not in the aggregate. The only way objects are added to and removed from the `Array` is through the interface `Vector`. The `Data` (which would have type `Object`) are from our point of view not a part of the implementation of the `Vector`, thus not a part of the aggregate. The `Enumeration` object which iterates through elements of the `Vector` has access to the `Array`, but itself is accessible by the external objects which can access the `Vector`. It acts as another interface to the `Vector` aggregate. Multiple `Enumeration` objects can exist using this idea, something which is impossible using monolithic interfaces. Overall

[3] By interface we do not mean Java's interface, rather we mean an entry point for accessing a collection of objects.

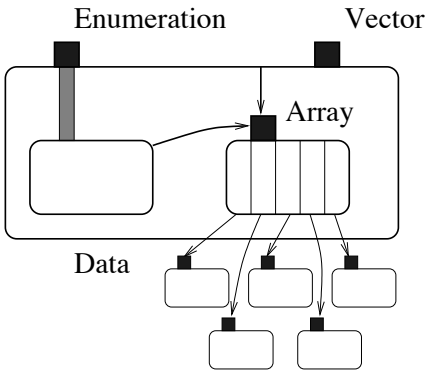


Figure 2.1: Java's `Vector` as a aggregate with multiple interfaces

we consider the `Vector` object, the `Array` object, and all the `Enumeration` objects to be a part of the `Vector` aggregate.

Noble discusses various styles of iterator, including the one just mentioned, and the extent to which they preserve encapsulation [147]. Baker expresses perplexity that they are required in object-oriented languages at all [15]. We must stress however that iterators are but one important example.

This model of aggregate object closely resembles *components* [181], of which prime examples are CORBA [161], Microsoft's COM [172], and JavaBeans [136]. These systems are however less concerned with preserving the distinction between the objects which are inside an aggregate and otherwise.

2.4.1 Aliasing

When two objects refer to another we have aliasing. Aliasing is essential for implementing linked data structures such as doubly-linked lists, and for many object-oriented idioms such as the Observer design pattern [84]. A number of classifications of the kinds of references exist. The first distinction is determined by the lifetime of an reference. A *static* reference is stored either in an object's field or in a global variable. Static references tend to exist in the heap. *Dynamic* references exist through local variables and method arguments or return values. Dynamic references exist on the stack and are ephemeral because they cease to exist when the particular stack frame vanishes upon method exit [110]. Static aliases are deemed to be a more serious concern [110], whereas dynamic aliases are acceptable since they are essential for implementing real programs [10].

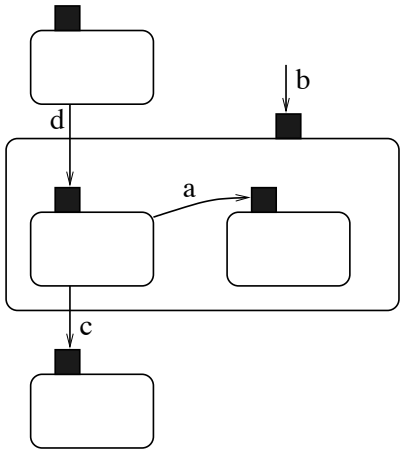


Figure 2.2: (a) Internal, (b) external, (c) outgoing, and (d) incoming references

In the presence of aggregation a different categorisation of references is possible. A reference can be categorised based on which aggregate its source and target belongs to. Consider the objects in Figure 2.2. References which are used within an aggregate's implementation are called *internal* (a), whereas references to an aggregate are called *external* (b). Internal references are crucial to understanding how an aggregate works, whereas external references generally do not affect the aggregate's implementation [150]. (This categorisation of a reference depends upon which aggregate is being considered — reference (a) can be considered internal in the larger rounded box, as well as external to the right hand nested object.) Internal and external references are considered *benign* because they do not cross the boundary of the aggregate, and are therefore "mostly harmless [6]."

Outgoing references (c) are those which originate from an object inside an aggregate to an object outside the aggregate's boundary. Diagrammatically, these are references which go outwards crossing a line representing an aggregate's boundary. *Incoming references* (d) are those going the other way, from an object outside an aggregate to an object inside an aggregate. Both outgoing and incoming references can be problematic. Outgoing references are often treated with care because the external state to which they refer is beyond the boundary of the aggregate, and this state can therefore be changed independently of the aggregate. Incoming references can modify the internal state of an aggregate, without using the aggregate's interface. Since these can violate an aggregate's invariants without the aggregate's knowledge these are the most serious [150].

2.4.2 Encapsulation

The idea of distinguishing the objects inside an aggregate from those not inside and then protecting the internal objects from external access is an example of *encapsulation*. This concept has been known since the halcyon days of structured programming, and has become an integral part of the object-oriented patois. Alan Snyder defines encapsulation as "... a technique for minimizing inter-dependencies among separately written modules by defining strict external interfaces [180]." Motivating this definition is the desire to allow any part of a software system to change without affecting the remainder of the system.

In object-oriented programming, the interfaces in Snyder's characterisation correspond to the method signatures of classes. Encapsulation is achieved by hiding the implementation of the classes using the visibility restrictions on the fields and methods of a class, or of the classes in a package [93, 72], or using export policies as in Eiffel [138]. These visibility annotations provide *information hiding* rather than encapsulation, and contribute little to solving the aliasing problems addressed in this thesis. The reason is simply that an object stored in a `private` field can by error or oversight be exported using a `public` method or be passed into a parameter, a global object, or some other object which may otherwise expose it. Perhaps diligent use of a particular privacy regime may achieve encapsulation [123, for example], but such a scheme can not be easily checked simply because the privacy information is not a part of an object's type.

Thus encapsulation requires not only that one module is protected from code changes in other modules, but also that objects in one aggregate are protected from changes in others [110]. Rumbaugh and friends have the right idea: "encapsulation consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects [174, page 7]." We refer to the encapsulation of the objects which implement the aggregate as *object encapsulation* or *containment*. Unfortunately, current object-oriented languages cannot provide an enforceable notion of object encapsulation, and hence the notion of aggregate object is in the programmer's mind. The problem is that the representation of an object can escape its encapsulation boundaries via aliases, because an object's type does not contain information concerning which objects are private and where the boundary lies.

Aliasing breaks encapsulation, because incoming and outgoing references can violate the invariants of an aggregate object, because such references are beyond the

control of the aggregate object. The presence of incoming references is called *representation exposure*. Outgoing references can suffer from the problem of *argument dependence*. We examine these in the next two sections.

2.4.3 Representation Exposure

Representation exposure is a well-known problem which occurs when the objects forming the internal implementation or *representation* of an aggregate are accessible outside of the aggregate [130, 110]. Exposing representation has been recognised as bad programming practice. This is facilitated by inadequate programming language design which makes it easy to subvert an aggregate's interface and access its representation. Incoming references to the internal state can be used to change the aggregate's implementation and violate its invariants without the aggregate being aware. The result is that one loses the benefits the programming languages abstractions provide, namely encapsulation. Software components become tightly coupled, less reusable, and changes to one piece of code often requires changes to the others. Implementations become fragile. Unfortunately, there is no sure way of preventing representation in current object-oriented programming languages. This is simply because type systems are unable to express the fact that an object is representation.

2.4.4 The Effect of Effects

Aliasing alone is not problematic, as any red-blooded functional programmer knows [18]. It is when combined with computational effects that problems arise. In this case a change induced upon an object through one alias can be observed by the other aliases. This behaviour can be a deliberate choice, as for example a part of the Observer pattern [84], but it could be the result of unknown or unwanted aliasing. In such a case, aliasing in the presence of mutable state can violate the assumptions about a reference to an object's state.

Consider the simple example in Figure 2.3.

After running through the `while` loop, the value in `mycounter` will be incorrect. This is because the `Counter` object it holds is aliased by variable `sues_counter`, which inadvertently is incremented elsewhere in the code, away from the `while` loop above. Thus each time `mycounter.count` is displayed the output is unexpected. Improving encapsulation as suggested above would reduce the chance that such an alias can exist.

```
class Counter {
   int count = 0;
   void inc() { count++; }
}
Counter mycounter = new Counter();
...
Counter sues_counter = mycounter;
...
while(stuff_to_count()) {
    mycounter.inc();
    display(mycounter.count);
}
...
bool stuff_to_count() {
    ...
    sues_counter.inc();
}
```

Figure 2.3: Aliasing is Problematic

```
class Key {
    int val = 0;
    int hashcode() { return val; }
    void munge() { val = val + 1; }
}
class HashTable {
   Data[] buckets = new Data[107];
   void add(Key key, Data data) {
       buckets[key.hashcode() % 107] = data;
   }
   Data fetch(Key key) {
       return buckets[key.hashcode() % 107];
   }
}

// losing data in a hashtable
HashTable h = new HashTable();
Key mykey = new Key();
Data my_precious_data = ...;
h.add(mykey, my_precious_data);
...
mykey.munge();
...
Data retrieved_data = h.fetch(mykey);
```

Figure 2.4: Argument Dependence

A different approach is to allow the alias `sues_counter`, but ensure that it cannot modify the object, either directly or via one of its method calls. Such a reference is called *read only*. In this case the method call `sues_counter.inc()` would be invalid.

These two approaches do not eliminate all aliasing problems. Consider the example in Figure 2.4 which illustrates another problem with aliasing, namely the *dependence on mutable state*.

The position of an object in the array underlying the `HashTable` depends on the hashcode of the `Key` object. The `HashTable` assumes that the hashcode does not change, since it is used to determine the location in which to store elements when storing and when fetching values from the underlying array. In the code given, `my_precious_data` is stored in the hashtable, but it cannot be retrieved because the `munge` method of the `Key` object changes the value of its hashcode. Consequently, `retrieved_data` will not be the same as `my_precious_data`.

The `HashTable` ceases to function correctly due to a change beyond its control. The result is in an unwanted coupling between the hashtable implementation and the implementation of the external objects (`Key`s) it uses. The innocuous read of the hashcode is therefore potentially dangerous because it *depends on mutable state*.

Now let's generalise to aggregate objects. An outgoing reference refers to state which is beyond the control of an aggregate. If the aggregate's invariants depend upon the state of the objects the outgoing references refer to, then the aggregates invariants may be violated without it being aware since the external state can change beyond the aggregate's control [150].

An approach to dealing with this problem relies on the virtues of purely functional programming languages, in which there is no dependence on mutable state [18]. In a purely functional language values can be aliased indiscriminately without compromising safety. Adapting ideas of Grogono and Chalin [95], Noble et al. pursue this idea, in the context of object-oriented programming [150]. Rather than adopting pure value objects, they observe that changes to an object are invisible to aliases which depend only on the immutable part of that object. An object can then be sliced into its mutable and immutable parts, and some of its methods can be written to be independent of the object's mutable state. Certain references to an object can then be restricted to use only those methods which cannot observe the changes in the object.

To make the `HashTable` work properly, therefore, we require that its hashcode does not change. That is, the `hashcode()` function must be independent of mutable state. In the `Key` object either `val` would be immutable and then `munge()` would be

invalid, or the `hashcode()` method would have to be implemented some other way.

2.4.5 Fundamental Difficulties

Before progressing further, it is worth emphasising that the aliasing and its interaction with mutable state is both a feature and a problem with object-oriented programming [110, 95, 150, 84]. Any proposal dealing with these issues must exclude aliases in a way which does not disrupt the virtues of object-oriented programming. That is, the features cannot be excluded, only managed.

2.5 Alias Management

We have described how aliasing in object-oriented systems is essential but problematic. Alias management schemes address this "problem" by supplying the programmer with annotations which are used to control either the amount of aliasing in a program or the effects of aliasing when combined with mutable state. We classify the kinds of annotations into four categories:

- alias encapsulation
- effects control
- uniqueness
- borrowing.

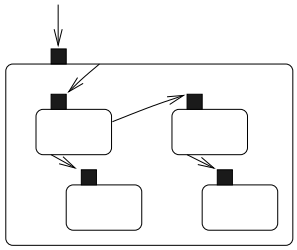
Many alias management proposals combine a number of these.

Before focusing on some of the proposals which appear in the literature, we will discuss what these aspects entail.

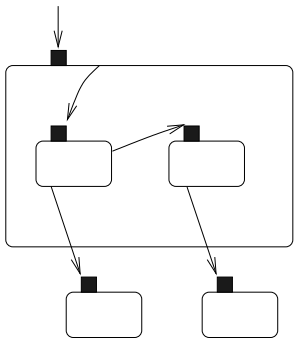
Alias Encapsulation

Alias management schemes which employ alias encapsulation prevent references to the inside of an aggregate from its outside. That is, they exclude incoming references. The kind of encapsulation provided varies from system to system. They can be roughly categorised in three ways, in increasing order of flexibility: *full alias encapsulation*, *flexible alias encapsulation*, and *fractal alias encapsulation*. The first two categories assume that the aggregate has only one interface and the third is the extension to include aggregates with multiple interfaces:

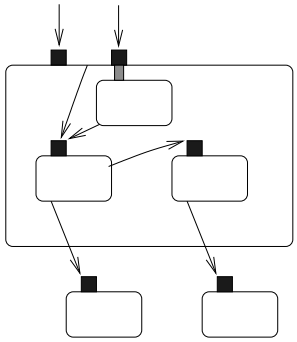
- *full alias encapsulation* — as well as excluding incoming references, this model also prevents outgoing references. This means that the only references to and from parts of an aggregate are contained within the aggregate, except for the external reference to the single interface object. Under this model it must be assumed, for example, that the data stored within a container is a part of the container's representation. Operations such as destructive read or copy assignment are required to move data in and out of a fully encapsulated aggregate. A picture follows.



- *flexible alias encapsulation* — in addition to the references that full alias encapsulation allows, this model allows outgoing references. Under this model the interface object is a dominator in the graph theoretic sense, that is, all external access paths to objects within the aggregate must pass through the interface object. This model allows, for example, the data stored within a list to simultaneously be an element of another list. A picture follows.



- *fractal alias encapsulation* — this model extends flexible alias encapsulation by allowing multiple interface objects to an aggregate. One characterisation of this is that the interface objects are cut-sets, that is, all external access paths to the objects within the aggregate must pass through one of these interface objects. This model supports data structures with iterator objects which can traverse the links implementing the data structure. A picture follows.



These distinctions appeared in an unpublished manuscript by Noble et al., though the first two appear implicitly within the description of Flexible Alias Protection [150].

Under this categorisation the system presented in this thesis is fractal, though we do present a way of carving out a subsystem which enforces full alias encapsulation.

Effects Control

A method may be annotated to report the degree to which it reads and writes its arguments and the target object. The state of a program may be partitioned into regions and the annotations can report which regions are read or written. This allows more modular definitions. When annotations are soundly checked by the compiler, then the statement that certain effects may occur can be viewed equally as a statement of the effects which cannot occur. Effects annotations can be used to limit the behaviour in a program, which can be used when reasoning about the program [94].

Uniqueness

A reference to an object is unique when it is the only reference to that object. Some alias management systems use this idea to some degree. Knowing that reference to an object is unique means that one can safely *move* it from one place to another and know that no other object will be affected by the move. Token passing algorithms rely on this idea. Unique tokens can be used to control the access to concurrent data structures [123].

There have been many suggestions to use uniqueness in object-oriented programming [100, 16, 109, 10, 141]. One other advantage of uniqueness is that when a unique reference is discarded, the object to which it refers can be deleted without troubling the garbage collector [16]. However, effective use of uniqueness requires additional

programming language support, including operations such as destructive read [109], copy assignment [10], or swapping [100] to safely move references from one place to another. Interestingly, Boyland avoids problems with these operations using a static analysis which checks that variables and fields holding unique references are never used, unless they are updated, after the transfer [32]. Minsky [141] considers the difficulties when objects hold unique references to objects. Combined with the unintuitive operations required to support uniqueness, this suggests that uniqueness may not be entirely compatible with object-oriented programming.

A lightweight approach to uniqueness is to use it just when an object is created. In this case a new object is created with no static references to it, just the implicit reference holding the result of a constructor call. Noble et al. calls this *freedom* and Leino *virginity*. Free objects can be assigned into an object's representation with the guarantee that there are no aliases to the given object, since objects are born free [150]. Leino and Stata uses this idea to address problems in specification [126].

Borrowing

Some methods provide a service such as printing, displaying or sorting which does not require a reference to its argument objects to be retained. When such a method has completed, no aliases to the method's arguments remain in the target of the method. Only dynamic aliases exist for the duration of the method call [110]. The objects to which these refer are said to be *borrowed*.

The qualifiers `uncaptured` [110] and `borrowed` [32] have been used to indicate borrowed references. Boyland uses borrowing in conjunction with uniqueness to allow unique references to be passed to methods, temporarily giving up their uniqueness, without the burdens of tracking linearity [32]. Borrowing can also be achieved using read-only variables which cannot be assigned [109] .

A more advanced view of borrowing would allow borrowed references to be stored in the fields of objects which exist only for the duration of the stack frame. Such a constraint would be difficult to formally state and prove. Notions of parametricity may help [168], though in the presence of mutable state and subtyping this may be difficult.

The Geneva Convention

The Geneva Convention on the Treatment of Object Aliasing [110] classifies techniques which deal with aliasing into four groups: *alias detection* — static or dynamic diagnosis of potential or actual aliasing; *alias advertisement* — annotations to help modularise the checking of aliasing properties of methods; *alias prevention* — constructs that disallow aliasing in a statically checkable fashion; and *alias control* — methods which isolate the effects of aliasing. In terms of these categories, alias encapsulation is a form of alias prevention, effects control is a form of alias control, uniqueness annotations represent a form of alias prevention of a different flavour to alias encapsulation, and borrowing is an alias advertisement scheme which corresponds to stating that the arguments of a method are not captured by the method's body, but also an alias prevention scheme since no aliases can be created.

2.5.1 Systems of Alias Management

In the following we review proposals which deal with aliasing and effects in object-oriented programming. We annotate each system to indicate whether it uses alias encapsulation (A), effects control (E), uniqueness (U), or borrowing (B).

Islands [109] (A, E, U, B)

Hogg's *Islands* proposal was the first alias management scheme for object-oriented programming. Not only did Hogg's paper characterise the problem (along with the Geneva Convention [110]), but every element which appears in the proposal appears in some form in each of its descendants.

An *island* is the transitive closure of all objects reachable from a particular object called the *bridge*. Messages to and from objects in an island must use the bridge, making the bridge a single access point to an island. Having a single entry point facilitates proving properties of objects. Islands may be nested, but they enforce only full alias encapsulation. Objects may move between islands using a destructive read operation which nullifies all other references to the itinerant object.

Islands rely on some invariants based on the alias modes `read`, `unique`, and `free` which annotate interfaces. `Read` states not only that an object can only be read, but also that it cannot be assigned to a field. This allows a distinction between static and dynamic aliases. `Read` mode is used to implement side-effect free functions and hence borrowing. Mode `free` annotates objects which have no static references and `unique`

annotates those which have a single static reference. Destructively reading a `unique` reference gives a `free` reference. A unique reference to a bridge object is the only external reference to the entire island reachable from that bridge. Hence the entire island can be reasoned about without reference to the remainder of the program.

The paper [109] presents Islands in a language without types, claiming that types are not needed because run-time checking could guarantee safety of the annotations. These claims are not substantiated. Furthermore, the semantics of destructive read is unintuitive and using it would require a programmer using Islands to make a paradigm shift. Nevertheless, this work is important because it recognises the problem of statically controlling aliasing and defines a system for full alias encapsulation.

Balloons [10, 11] (A, U, B)

Balloons implement another full alias encapsulation scheme. The advantage with this scheme is that only a single annotation on a class is required. The annotation `balloon` indicates that objects of the annotated class must satisfy the *balloon* invariant; this states that there is at most one reference to the object, that this reference, if it exists, is external to the balloon, and there are no incoming references into the balloon.

Free balloons, those which are not stored in a field, can simply be assigned to a field. Other balloons require the more costly copy assignment operation, which copies the entire balloon before assignment. The semantics underlying balloons are simple, but checking that the correct assignment operation is used and thus that the balloon invariant holds is not trivial. Balloons can also be arbitrarily borrowed, since any attempted assignment will be by copy assignment which will preserve the original balloon's invariants.

The single annotation and rather simple conceptual model are at the heart of Balloons' appeal. Unfortunately there are problems. The copy assignment operation implies that object identity is no longer stable, requiring a change in a programmer's thinking. Copy assignment is also potentially expensive. The complex abstract interpretation required for checking is unlikely to produce coherent error messages for a user, and we believe that this would be sensitive to small changes in a program. Both these problems would make Balloons difficult to use.

The main deficiency with Islands and Balloons is that they adopt an all or nothing approach to aliasing which excludes forms of aliasing useful in object-oriented programming. The remainder of the proposals do not adopt full alias encapsulation,

avoiding both copy assignment or destructive read, and gel more closely with current practice in object-oriented programming.

Flexible Alias Protection [150] (A, E, U)

Flexible Alias Protection (FLAP) focuses on protecting aggregate objects. It was the first system to offer flexible alias encapsulation. The objects accessible to an aggregate are categorised as either *representation* or *arguments*. Representation objects are those not accessible outside the aggregate, and argument objects are externally accessible objects but treated as immutable by the aggregate. For flexibility and composability, arguments can be partitioned to reflect the different roles they play within an aggregate object. When used in its strictest form, with all external objects treated as immutable, an aggregate object will be unaffected by changes to external objects, except those which use the aggregate's interface.

FLAP offers a number of features, which are described below. Each of these has a corresponding aliasing mode which annotates the types of fields and variables. In addition to the aliasing modes there are also *roles* which are used to logically partition object space. Initially there is a single partition for objects which can be accessed from anywhere in the system. Each object also has a partition in which its representation is stored. A class (which may be generic) has roles allied with its type variables so that classes can be used in different places, with different instantiations of the role parameters. The roles correspond to the owners of the entities of that role (though Flexible Alias Protection did not use the term owner).

Freedom The `free` annotation indicates that an object has only one dynamic and no static references to it. This is generally used for the objects returned by constructors. Such objects can then be assigned anywhere without problem, assuming that they are assigned only once.

Representation The `rep` annotation indicates which objects are a part of an object's representation. Such objects can be modified, and stored in and retrieved from internal containers, but never exported from the object to which they belong. The containers in which the representation can be stored are also representation, so these cannot be exported either.

Value Objects The annotation `val` is applied to classes which implement values. Such objects cannot be changed and are said to have *Value* type [133, 119, 95]. Java's `String`, `Integer`, and `Float` objects are perfect examples. Because they are immutable, values can be freely shared without adverse effect, like values in a functional programming language [156]. A minor problem with value objects in an object-oriented programming language is that different objects representing the same value have different identity. To avoid any adverse implications of this, Grogono and Santas [97] and Baker [14] suggest that values should be compared using equality rather than identity, in the spirit of referential transparency. FLAP *does not* adopt this idea, though it easily could.

Argument Independence FLAP introduced the idea of the independence of mutable state, as discussed above. The fields of an object which do not change after an object has been initialised are called *immutable*. Only these fields and the immutable parts therein can be accessed through so-called *immutable interfaces*.

The mode `arg` is used to refer to fields and variables which can only access objects through their immutable interface. Such objects can be treated as values. Even though the objects referred to through an `arg` reference may be referred to elsewhere through their full interface, any modifications which occur cannot be observed through the immutable interface.

Checking argument independence is simple. Firstly fields are annotated as mutable or immutable. Immutable fields can only be assigned to when the object is initialised. The methods of objects in immutable fields can be invoked, and this may change their state, but the field itself can never be updated. Methods in the object's immutable interface can access only immutable fields through their immutable interfaces, as well as access itself through its own immutable interface.

FLAP suggests that `arg` be used on an aggregate's arguments, that is, on all outgoing references. This is called *argument independence* since an aggregate's integrity will be protected against changes which occur in its arguments caused by aliasing. Thus an aggregate's invariants cannot be violated due to changes in the mutable state of its arguments. Limiting the dependence on mutable state earns the predictability of immutability without losing the abstraction and modelling power offered by mutable aggregate objects.

The "Loophole" – `var` The final mode is `var`. This mode has two forms which have rather different semantics. In both cases `var` mode means that there is no restriction on which part of an object's interface may be used. It is used to refer to mutable objects. When `var` mode appears without a role parameter, this means that the objects are accessible system wide. When `var` mode appears with a role parameter, then the objects assigned that role come from whichever partition is bound to the parameter, without any restriction on which part of its interface may be used.

Noble et al. state that mode `var` introduces a loophole which allows a regular object-oriented program to be embedded into a language with Flexible Alias Protection. It does not provide any protection for the thing which it annotates. As loopholes go this is not so bad, because `var` cannot be used to violate the invariants underlying the other modes.

Flexible Alias Protection made the following contributions to alias management:

- the realisation that every object an aggregate refers to is not part of its representation. Some objects are external to an aggregate;
- a modular, parameterised specification which can be written as a type system;
- strong constraints on representation access; and
- the concept of independence of mutable state.

A less strict version of independence of mutable state seems to be more practical: rather than requiring that an object not depend on the mutable state of external objects, it may be better to require that *the invariants of* an object do not depend on the mutable state of external objects. This idea has not been considered by Noble et al., nor is is considered in this thesis, but it does deserve further investigation.

Ownership Types [162, 61, 59, 58] (A)

Ownership types (OT) were originally devised to understand the core encapsulation mechanism of Flexible Alias Protection, but are now of interest in their own right. Beginning with the observation that the modes `arg` and `val` of Flexible Alias Protection are related to effects control, and are thus orthogonal to the structural restrictions imposed by the other modes, ownership types adopted only `rep` and the roles parameters. Together these are called *ownership contexts*, or just *contexts*. The roles parameters are called *context parameters*.

The first modification OT made to the remains of FLAP was the addition of a context called `norep` which corresponds to no object's representation and thus to objects which are accessible system-wide. This context corresponds to `var` without a role from FLAP.

In FLAP it was impossible to give a type to self. A major contribution of OT was the realisation that the role/context associated with a class could be interpreted as the *owner* of elements of that type. The keyword `owner` was introduced to serve this purpose. The `owner` allowed additional benefit of being able to build arbitrarily linked data structures where all the links in the data structure have the same owner and therefore are afforded the same amount of protection.

The form of encapsulation ownership types, and hence FLAP, offered is known as *owners-as-dominators*. The context `norep` can be interpreted as the root object in an object graph, and each `rep` context as the object corresponding to the given instance, that is, to `self` or `this` of the object in which it appears. Parameters refer to the object corresponding to the context to which they are ultimately bound. This means that each type, and hence each object, was owned by another object. The type system statically enforces the owners-as-dominators property which states that all access paths from the root of the system to an object pass through the owner of that object. That is, owners are dominators. This is a very strong contraint on access to an object's representation, namely that the owner of the representation is a single access point for external access to that representation.

Ownership types were originally defined for a class-based language resembling Java without inheritance [61]. Inheritance was added [62] demonstrating that preserving the ordering between contexts in the type system was essential to preserve the owners-as-dominators property. The formulation was overly complex and never published. We redress this issue in Chapter 6.

A more thorough review of this ownership types system is presented in Chapter 6, where we describe an encoding in terms of the calculus presented in Chapter 5. As a part of Chapter 6, we also describe how inheritance and subtyping can be regained.

Structural issues underlying ownership types were studied in [163]. The underlying dominance constraint was simplified to a local condition between objects, given the nesting between objects implicit in the containment structure. This local condition will be used as the basis of our type system, and is discussed in more detail in Chapter 3. Under the guidance of Potter, students Hill [104] and Santibáñez [176] developed visualisation tools which examined the ownership structure implicit in object-oriented

programs based on similar ideas.

We have also applied ownership to an untyped prototype-based programming language which allows a more sensible clone operation and offers a solution to the *prototype corruption* problem [148].

Boris Bokowski implemented ownership types using his CoffeeStrainer tool [22] and issued a short critique [23]. Similarly, Alex Buckley made another implementation as a part of his Master's thesis [39]. Buckley's implementation included a few extensions such as anonymous contexts in methods, which allow borrowing, and context polymorphic methods. We also implemented flexible alias protection as an additional Pizza topping [153]. The semantic difficulties encountered in that venture precipitated the need for the theoretical work presented in this thesis.

Data Groups and Extended Static Checking [125, 67, 73] (E)

The Extended Static Checking for Java (ESC) project [73] has constructed lightweight program verifiers for discovering errors in Java programs based on partial specifications of the software's behavior.

Leino uses *data groups* to allow the specification of mutable state in a way that is compatible with subclassing, simple to use, and sound. A data group represents a collection of an object's fields. A `modifies` clause states which data groups may be modified. Apart from documenting the behaviour of an object-oriented program, data groups can be used to infer non-interference properties These can then be used to reason about a program's behavior [125]. The limitation in this approach, as described, is that the data groups are statically defined, and therefore, fixed.

In another paper [67], Detlefs, Leino, and Nelson trace the problem of representation exposure to what they call *abstract aliasing*, which is when an unexpected side effect occurs between two values, but the dependency — one of their modular specification constructs — linking those two values is out of scope. Driven by pragmatic concerns, they also present a convincing case that borrowing and outgoing references are essential for programming, and that it should be possible to initialize representation with objects created externally to the owner of the representation. For this reason we endeavour to handle these features in this thesis.

JAC [122, 185] (E)

JAC (Java with Access Control) abandons explicit alias encapsulation models and relies exclusively on effects control. Annotations express the access rights a method has to its arguments and to the fields of the target object during a method's evaluation. The access rights include `readonly`, `readimmutable`, which prevents the dependence on mutable state, and `readnothing`, for pure functions. These annotations are transitive, that is, `readonly`, for example, means than no write occurs in the entire transitive state of the entity which has this annotation.

These mechanisms can be used to organise a program according to one of the alias encapsulation schemes, by for example, allowing representation to be written and non-representation to only be read (as in Universes which follow), but the annotations in this case can only be suggestive of the structural constraints. Indeed Theisen proposes extending the model with some form of ownership as future work. Static checking of the validity of the annotations can be achieved using Java's type system and a trick with interfaces [185]. JAC has the benefit of being both conceptually simple and easy to check.

The Object-Oriented Effects System [94] (A, E, U, B)

Greenhouse and Boyland's *Object-Oriented Effects System* is a modular approach to specifying the effects which may occur during a method's evaluation. It is the first such system proposed for an object-oriented language. The effects include the reading and writing of regions which denote some collection of the mutable state. They form a hierarchy, the top of which is called `ALL` and represents all mutable state. New regions may be declared within the body of a class. A part of their declaration includes a parent region. This is used to construct the hierarchy. Regions may be class or object/instance specific, allowing for a precise specification of the potential effects that methods exhibit. The effects information is used to determine the validity of certain program transformations.

The hierarchial regions induce a subeffect relation, namely that a read or write on a region can be considered to be a read or write on any region which contains that region. This avoids the problems with effects specifications and subclassing following the solution in Leino's Data Groups proposal [125].

The Object-Oriented Effects System also includes unshared fields, Greenhouse and Boyland's version of uniqueness. This creates some difficulty with checking the

validity of their annotations. Perhaps this is why Greenhouse and Boyland use a control flow analysis rather than a type system to specify their analysis.

Related and, in our mind largely compatible, work by Boyland includes borrowing [32]. The annotation `borrowed` indicates objects which can be passed to a method but not assigned within the method. Apart from reducing aliasing, borrowing helps preserve the `unshared`ness of fields. While this adds flexibility, it unfortunately also requires complex flow analysis to check. A type-theoretic treatment would be more palatable in our opinion.

Universes [144] (A, E)

Universes is a representation protection scheme for a Java-like language derived from our ownership types. It offers instance-based protection, as well as some protection based on classes. Together these would implement a (limited) form of flexible alias protection were it not for the pervasive `readonly` mode which ignores encapsulation restrictions. This annotation is used to implement additional interfaces to an object, so long as they are read only, and thus Universes implement a limited form of fractal alias protection. `Readonly` mode is also used to implement friendly functions which can access the representation of different objects, so long as all but one member of the friendship are read only. Whether this limitation is too restrictive in practice remains to be seen.

Types may be annotated with either the keyword `rep`, to indicate representation which is confined to a particular instance, or the keyword `readonly`, for unconfined, read only references. Type checking is based on the following table which determines the actual type of a field, given the type of a target. More precisely, it converts a type as it appears in the class of the target type to a new type depending on the target type's annotation.

		Field Type		
		C	`rep` C	`readonly` C
Target Type	D	C	`rep` C[†]	`readonly` C
	`rep` D	`rep` C	*undefined*	`readonly` C
	`readonly` D	`readonly` C	`readonly` C	`readonly` C

The first cell accommodates the behavior of Java, namely that the type written in the program text is the type of an object.

The second cell, marked with a †, indicates what can be considered dubious behavior. It allows access of the representation stored in a field, though the resulting representation cannot be assigned, using other rules in their type system.

The second row allows a non-rep field of a rep to be accessed, but it is treated as rep — this is the key to the protection in their scheme: a simple syntactic trick to make all things accessible from representation also representation.

The second cell of the second row disallows the representation's representation from being accessed — its type is undefined.

The bottom row states that any type accessible from a read only type is also read only. Similarly, the last columns states that the type of a field declared read only field is read only.

The type rules use this table to determine valid field access, assignment and method call. However, the Universes formalisation suffers from being non-standard and hard to follow. The table specifies one kind of behavior, which is then made irrelevant by later rules in their type system.

The language underlying universes lacks both nested classes and static fields and methods. It is clear, at least to us, that the trick used by the table fails in the presence of these features. Had the type system adopted an approach based on existing type-theoretic constructs, these problems may have been avoided. The type system presented in this thesis avoids the pitfalls of *ad hoc* formality.

Confined Types [24] (A)

Bokowski and Vitek's *Confined Types* impose static scoping restrictions on dynamic object references to enforce confinement properties in Java programs. Objects from classes annotated with the keyword `confined` cannot be accessed outside the package in which the class is defined. Confined types provide per object protection where the units of encapsulation are packages.

To facilitate their protection scheme with minimal disturbance to ordinary Java programs they require the notion of an *anonymous method*. Anonymous methods are those which cannot export self. They allow confined types to derive from classes which are not confined, with the proviso that the derived class only uses anonymous methods from the super class. This prevents self from escaping the domain of confinement.

Confined types also use privacy on classes which are confined and methods which access confined types to further enforce the confinement.

The Confined Types proposal also considers pervasive classes such as `Thread` and `Throwable`. These cannot be confined because their dynamic use is not confined to any package.

Related Proposals

Eiffel offers *expanded types* which are expanded in-place within the fields of an object, rather than being a reference to a heap-allocated object. Objects of expanded type cannot be aliased [138]. Unfortunately, there are a number of significant problems when considering expanded types as a means of encapsulation. The fields of the expanded type are not protected, unless they are also expanded. Expanded types cannot be recursive.

In a similar vein C++ offers *value objects*. These can be placed on the stack or in an object's field, but their address can be taken. They suffer from similar problems as expanded types, but they can be aliased, even to the point where an alias will exist longer than the value object in the case of stack-allocated value objects, and thus can result in dangerous dangling pointers [72]. They do not solve encapsulation problems, though they can make some programs a little quicker.

Kent and Maung [120] propose a dynamic encapsulation scheme for Eiffel. Their protection annotations use the names `protected` and `private` borrowed from C++ to indicate notions which in hindsight can be seen to resemble `owner` and `rep` in Ownership Types. Unfortunately their system requires too much dynamic checking, and its description is somewhat imprecise. We believe that their proposal has been subsumed by Ownership Types.

Simons [178] employs an ownership policy to guide memory management using the Orthodox Canonical Form. In particular he addresses the transfer of object ownership and the reduction in memory usage via sharing and execution time by delaying copying. While useful in a language such as C++, many of the problems would cease to exist in a programming language with garbage collection and value objects.

Dong and Duke [69] extend object-oriented specifications with a "geometry" of containment. But by making everything an object refers to contained within that object, and by not distinguishing outgoing references, they are forced to introduce some rather odd geometries. Furthermore, their proposal does not easily account for idioms such as an aggregate object with multiple interfaces.

Bryce and Razafimahefa [38] use access control mechanisms which are dynamically checked to prevent aliasing in a security context. Underlying their protection

model, called *object spaces*, are distinct spaces to which objects are assigned. Protection is determined by which object spaces have access to which others. The mechanisms have the flexibility of being dynamic, at the cost of dynamic checking.

Chan, Boyland, and Scherlis [57] present a complex set of annotations called *promises* which express a number of program properties, such as uniqueness, immutability, read and write effects, and so forth. Their aim is to improve the understanding of a program's behaviour to enable, for example, program transformations. This proposal can be seen as a less structured precursor to Greenhouse and Boyland's Object-oriented effects system [94].

Genius, Trapp, and Zimmermann [88] describe an algorithm which recognises certain containment relationships between classes (and hence objects) using the class dependency graph. With this information they improve the locality of memory access in the implementation of an object-oriented language. Unfortunately, their proposal is hobbled whenever a class which they recognise as being contained is reused. The examples for which they present performance figures are trivial.

Comparison

We conclude this chapter with a comparison of the major systems discussed above.

Islands [109] includes effects annotations, but offers only full alias encapsulation. It most likely cannot be checked soundly, since it is specified for an untyped programming language. It must rely on run-time checking. Islands also depends on the nonstandard destructive read operation.

Balloons [10] like Islands offers only full alias encapsulation, but it uses instead a nonstandard copy assignment operation for moving objects between balloons. It has the advantage that only a single annotation is needed to indicate a balloon class. It suffers because it has two assignment operations and requires a complex, non-modular abstract interpretation procedure to determine whether the correct assignment is used and thus that the underlying invariant is preserved.

The remaining systems, which were subsequently devised, abandoned the nonstandard operations and the full encapsulation model, relying exclusively on standard object-oriented references semantics.

Of all the systems discussed above, only Flexible Alias Protection [150] and Ownership Types [61] have the flexibility which parameterisation give. On the other hand, systems without parameterisation do not suffer the syntactic baggage which comes with parameterisation. Both OT and FLAP can be checked modularly.

JAC [122, 185] is purely based on effects, but can be used idiomatically to implement a form of safe aggregate object, by making certain references read only. JAC cannot enforce the distinction between external and internal references, but it can be checked modularly.

The Object Oriented Effects System [94] implicitly has some notion of aggregate object, though it deals mainly with reporting effects. It also has uniqueness and borrowing, which all proposals apart from Islands and Balloons lack. The proposal takes special pains to work smoothly in the presence of inheritance. It is largerly orthogonal to the approach taken by Flexible Alias Protection and Ownership Types, and can be checked modularly.

Universes [144] seems to be a simplified Ownership Types system, while adding some other features. It does not include parameterisation, but gains some ground with an all pervasive `readonly` mode. Universes can be checked modularly, though the techniques employed do not appear to extend to include inheritance, static fields, or nested classes.

Confined Types [24] require only a single annotation on a class and on some methods, and can be checked modularly and statically. They provide only package level protection, but the authors argue that this is sufficient for their application (security). Confined Types are specified for a complete programming language, namely Java.

The work from the Extended Static Checking [73] project differs significantly from the other work described. This work does not consider aggregate objects directly, since it stems from system specification and verification. The constructs ESC introduces address problems in their specifications, whereas the other approaches are based on programming languages and address problems which arise through programming. A marriage would be beneficial to all parties concerned. Müller's recent thesis takes a step in this direction [143].

By casting ownership types into a type-theoretic framework, this thesis aims to take the first steps towards providing a foundation for describing, comparing, and hopefully reasoning about the alias management schemes just discussed and future alias management proposals, in particular, those based on alias encapsulation. It is time now to begin.

Chapter 3

A Model of Containment

Underlying the type systems presented in this thesis is a model of object containment which exploits nested object ownership to provide a simple invariant to constrain access to an aggregate object's private representation.

Section 3.1 takes the first step and presents a simple model based on *dominators* which we call the *owners-as-dominators* model. Section 3.2 exploits the inherent nesting between object owners and formulates a condition defined between pairs of objects which determines whether one can have a reference to the other. This condition can easily be incorporated into a type system which includes subtyping. Section 3.3 discusses the limitations of this model, in particular, that the containment property it enforces precludes some common object-oriented idioms. Section 3.4 generalises the containment invariant to produce the model of containment which we adopt for the remainder of the thesis. We then explore some possible uses of this model. Section 3.5 presents a diagrammatic notation which, under two reasonable assumptions, contains sufficient information so that the validity of a reference is determined by a simple graphical condition. Section 3.6 offers a comparison with Java's nested classes and their visibility in order to justify that our model is natural. Finally, Section 3.7 concludes this discussion, leading into the formal part of the thesis.

3.1 Representation and Ownership

In the previous chapter we described the notion of aggregation and how it can be defeated by representation exposure. In other words, the notion of aggregation is not enforceable since the objects which constitute an aggregate's private implementation, *i.e.*, its representation, can be accessed by objects which are not considered to be part

of the aggregate. Current type systems are unable to prevent representation exposure, because they cannot distinguish between objects which are part of the representation and those which are not, and thus cannot protect those which are.

The first step toward resolving this is to recognise that some form of *ownership* is required to determine whose representation is being protected. The approach we take initially is to consider an aggregate object to own its representation. From this we can define an object's representation as the objects it owns.

For now we assume the following properties of ownership:

- every object has a single owner which is either another object, or, for objects which are the representation of no object, the root of the system; and
- an object's owner is fixed for the object's lifetime.

The first requirement is to keep the type system simple, though it clearly precludes any representation shared between different objects, something which may be desirable for some applications. The second enables us to develop a static type system, since the owner will become a part of an object's type. Changing owner during evaluation would amount to changing type, but this is unlikely to be sound in an imperative calculus — certainly not in any simple manner. In addition, the amount of ownership is potentially unbounded, since the number of objects is unbounded and every object can potentially own other objects. This needs to be handled carefully when developing a static type system, but we reserve this discussion until Chapter 5.

Let's briefly consider a simple example. A linked `List` data structure consists of a number of `Link` objects which refer to `Data` objects. The `Link` objects implement the `List` object, so we consider them to be the `List`'s representation. On the other hand, we do not consider the `Data` objects which the `List` holds to be a part of the `List`'s representation, because they can be accessed by objects which are not a part of the `List`'s implementation, presumably including the owner of the `List`. The diagram in Figure 3.1 illustrates the idea.

The view that the `Data` elements are not a part of the `List`'s representation was first put forward as a part of Noble, Vitek and Potter's Flexible Alias Protection [150]. This differs from Hogg's Islands [109] and Almeida's Balloons [10]. These two proposals consider the `Data` as a part of the `List`'s representation, and provide additional operations for moving or copying objects into and out of such data structures.

There are a number of consequences of our simple model. Firstly, an object can refer to objects which it owns as well as objects it does not, implying that there is a

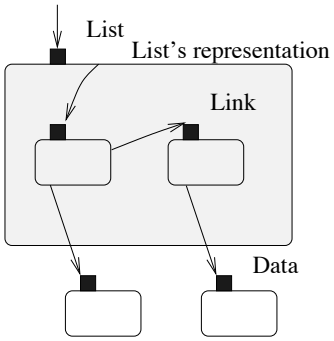


Figure 3.1: `Data` is not representation

distinction between being an *owner of* and having a *reference to* an object. Secondly, the representation need not be restricted to the values in an object's fields and can be an arbitrary linked data structure. Furthermore, an object's owner need not be the object which creates it (though this is not apparent in the example).

An aggregate should be treated as a single entity and its representation should be accessed only via the interface to the aggregate, that is, through the owner object. The owner acts as a single entry point to an aggregate, because the remainder of the system must use its interface to effect changes upon its aggregate. Modifications to an aggregate which circumvent the owner via aliases to its representation can violate the aggregate's invariants, without the aggregate being aware of the changes. The existence of such aliases is called *representation exposure* and it is this problem that we wish to avoid.

We now state a property in terms of access paths to objects which amounts to saying that aggregate objects are accessed through their interfaces and that there is no representation exposure. This particular formulation was first characterised by Potter, Noble and Clarke [163] and we call it *representation containment*:

> *All paths from the root of a system to an object must pass through that object's owner.* ($\star$)

This property ensures, for example, that a `List`'s `Link` objects are not exposed outside of the `List`, as this would require a path to some `Link` which did not include the `List` which owns it. The property does allow, for example, the two different `Lists` to share the same `Data`, but not share any `Links`.

This property is equivalent to stating that owners are *dominators* [7] for access paths from the root of the system to the objects they own [163]. For this reason we

call this the *owners-as-dominators* model.

A problem with property $(\star)$ is that it is defined globally over entire paths, rather than being localised to particular objects and references. In this form it would be difficult to incorporate into a type system. In a moment we make some additional observations about aggregate objects which enable us to define a local condition for determining whether a reference between two objects is valid. The resulting condition is equivalent to $(\star)$.

The representation of an aggregate can be considered to be inside the aggregate. But an object may be an aggregate and itself be part of another aggregate. For example, an aggregate can have a `List` as part of its representation, which in turn has `Link` objects as its representation. Objects are therefore nested and it makes sense to talk about the inside or outside of an object. From this we can define a notion of containment: an object is contained within the aggregate which it belongs to and it cannot be accessed from outside that aggregate, or more simply, representation must be protected from outside of its owner.

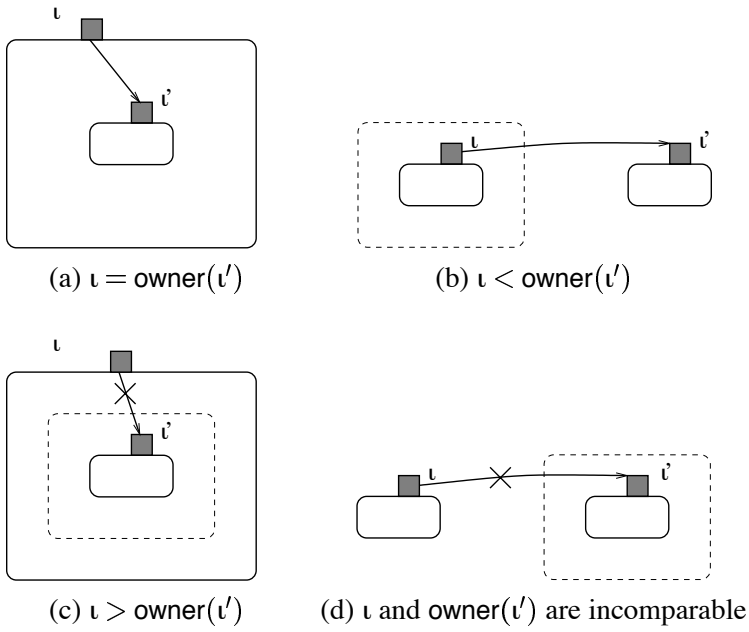
In the next section we formalise this more precisely and provide a more workable, but equivalent, definition of representation containment.

3.2 The Owners-as-Dominators Model

The nesting mentioned in the previous section induces a partial order between objects. From the fact that every object is inside its owner, we straightforwardly obtain the relation $\iota < \mathsf{owner}(\iota)$ for all objects ι. This is actually a tree, since an object's owner must exist when the object is specified. It is called the *ownership tree* [163]. (This tree corresponds to the *dominator tree* [8], where the owners are dominators.) The relation in which we are interested is the reflexive, transitive closure of the ownership tree, which we will denote by $\leq$ for the current discussion.

We will now define a condition between pairs of objects which, given an ownership tree, determines whether a reference can exist from one object to the other.

Consider Figure 3.2 in which object ι refers to object ι'. References in (a) and (b) do not break the containment invariant — these references will not create alternative access paths to an object from the root of the system which do not pass through the owner. On the other hand, the references depicted in (c) and (d) do access an object's representation from outside of the object. The constraints underlying (a) and (b) can be packaged together to form a *containment invariant* which governs whether one



(a) $\iota = \mathsf{owner}(\iota')$ (b) $\iota < \mathsf{owner}(\iota')$

(c) $\iota > \mathsf{owner}(\iota')$ (d) ι and $\mathsf{owner}(\iota')$ are incomparable

Figure 3.2: Possible relationships ι and $\mathsf{owner}(\iota')$ for a reference from object ι to ι'. The nesting of objects (dark squares) corresponds to the ownership tree. Representation containment has a simple graphical interpretation: references are not allowed to cross from the outside of a rounded box to its inside.

object can access another [163]:

$$\iota \to \iota' \Rightarrow \iota \leq \mathsf{owner}(\iota').$$

The relation $\to$ captures that an object refers to another, in some sense, and thus denotes the object graph. $\Rightarrow$ is logical implication. This invariant states that the reference can exists only if the source of the reference is inside the target's owner.

Because this condition is defined locally between pairs of objects, it can determine whether a reference is valid without needing to examine the entire object graph, as the condition $(\star)$ seems to suggest. This locality makes it more amenable to incorporation into a type system. Indeed, a less elegant variant of it was employed in our first ownership type system [61]. The above statement of the containment invariant was discovered while considering the ownership structure underlying arbitrary object graphs [163], and was developed in conjunction with John Potter and James Noble. A proof that the underlying object graphs satisfy the property that object owners are dominators can be found in Appendix B.

Unfortunately this model is too rigid. We will now indicate some of its limitations, before generalising it to obtain a more practical model.

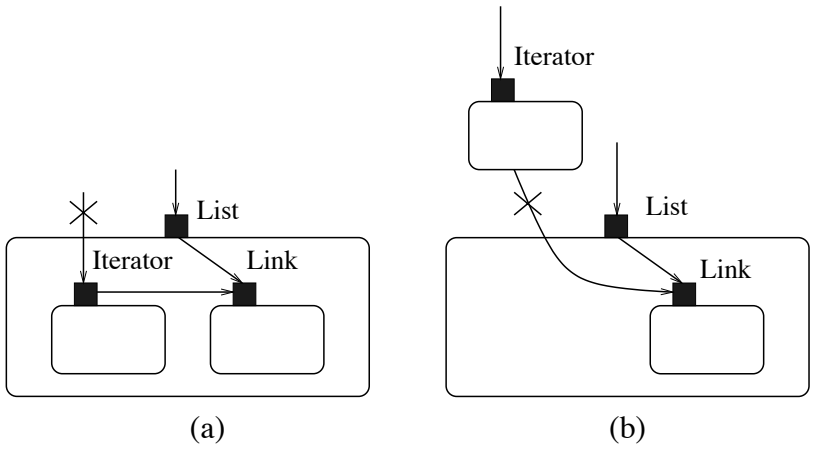


Figure 3.3: Invalid iterators in the owners-as-dominators model.

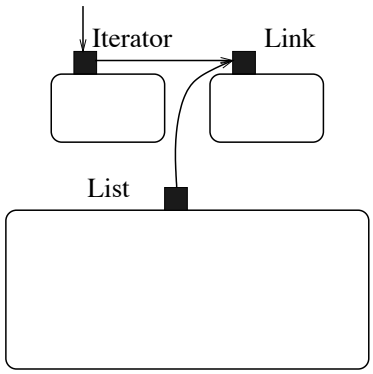


Figure 3.4: Iterators in the owners-as-dominators model must expose representation.

3.3 Limitations of Owners-as-Dominators

Although the owners-as-dominators model provides a simple notion of representation containment, it is, unfortunately, too restrictive to implement common programming idioms such as external iterators [147] without sacrificing representation containment. It is not possible, for example, for a `List` object to own its `Link` objects and allow the `Links` to be simultaneously referred to by an externally accessible `Iterator` object. Figure 3.3 illustrates that this is invalid since it requires two access paths to the representation. Coding this example in the owners-as-dominators model requires the `Link` objects to be exposed to direct external access, as shown in Figure 3.4. Noble, Vitek and Potter propose this solution [150], but it is unsatisfactory.

Iterators are not the only example that reveals the inadequacies of the owners-

as-dominators, but it is sufficient to illustrate that we need to extend our model to incorporate aggregate objects which have multiple interface objects accesses the aggregate's representation.

One obvious possible solution is to extend the owners-as-dominators model to allow objects to have multiple owners, and thus allow the `Iterator` and the `List` object to share the representation `Link` objects. This approach, however, falls short for a number of reasons:

- It is syntactically burdensome.
- It is difficult to statically track the owners an object may have, since this number may not be known when the object is created. This is the case with the `Iterator` example.
- It excludes examples such as an `Iterator`-like object which has its own representation.

For these reasons we chose to take another direction.

The key to generalising the model is to break the coupling between objects and their owners, which are also objects. To separate ownership from objects we introduce a different domain called *contexts* as the units of ownership. For now we can assume that contexts and their ordering form a tree, though in Chapter 4 we present examples which use a more complex ordering.

Each object has an *owner context* which abstractly represents its owner. The objects with the same context as their owner context are considered to have the same owner. Each object also has a *representation context* which is considered to be the owner of its representation. The objects which are a part of some object's representation are those which have the object's representation context as their owner. We see that the model employs a level of indirection. Each object has two contexts: an owner context, indirectly giving the objects which own it; and a representation context, indirectly giving the objects which it owns.

In the owners-as-dominators model, the position of an object's owner in the ownership tree constrains which objects it can access, namely, those which are inside the owner. Dually, the position of the object in the ownership tree constrains which objects it can access, namely, those whose owner it is inside. Thus there are two positions in the tree which govern object access. We can take a small leap of faith and use the two contexts introduced above to achieve a comparable effect in the new model. The

position of the object's owner context can constrain the objects which it can access, and the object's representation context can constrain which objects it can access.

Now let's bring all of the details together.

3.4 The Containment Invariant

We assume that contexts form a partial order $(C, \prec:)$. This partial order may be derived from a tree, a forest, or dag. It may have a maximal element which, when present will be denoted ε. The order relation $\prec:$ is called *inside*. The converse relation $:\succ$ is called *contains*.

Each object ι is assigned two contexts, an owner and a representation context, given by functions $\mathsf{owner}(\iota)$ and $\mathsf{rep}(\iota)$. The owner context can be thought of as where the object's interface resides in the ownership tree. It governs which other objects can access the object. The representation context is the owner of an object's representation. It also acts as a capability governing which objects an object can access. The *containment invariant* determines when an object can refer to another. It states:

$$\iota \to \iota' \quad \Rightarrow \quad \mathsf{rep}(\iota) \prec: \mathsf{owner}(\iota')$$

where $\iota \to \iota'$ mean that ι *refers to* ι'. In addition, we require that $\mathsf{rep}(\iota) \prec: \mathsf{owner}(\iota)$, so that an object can access itself. Other restrictions are possible and we describe some in a moment.

Contexts can, for example, represent the collection of packages in a Java program, where the inside relation represents the nesting between the package. In this case the set C would contain a maximal element to denote the root package, that is, the package without a name, which would become the owner context of objects accessible to any other object. This collection of contexts can be used to guarantee that values of certain types cannot be directly accessed outside of a given package. Confined Types use this idea [24], except that only one level of nesting is used. Similarly, contexts could represent classes where $\prec:$ captures inner class nesting and again a greatest element could correspond to some ubiquitous system-wide context. Values of types can be restricted so that they are not accessed outside of a given class. Universes take this approach [144], but again using only one level of nesting. And of course, contexts can also correspond to objects, as the discussion up to this point has suggested.

But contexts need not be restricted to the existing abstraction boundaries in an object-oriented programming language. Perhaps an orthogonal hierarchy of contexts could be established separately from the classes and packages, or perhaps encompassing a different level of granularity. Another possibility is that the contexts could represent the names (or IP addresses) of computers on a network, and then values of certain types could be restricted from being accessible beyond a certain network. The inside relation could model the subnet relation, for example.

More abstractly, we can impose additional restrictions on the abstract containment model to obtain different degrees of containment. For example:

- to model an object ι which does not have its own representation, set $\mathsf{owner}(\iota) = \mathsf{rep}(\iota)$;
- to model two objects ι and ι' sharing representation, set $\mathsf{rep}(\iota) = \mathsf{rep}(\iota')$. This precludes them from having their own private representation;
- to model iterators, where the owner of the container ι and the iterator ι' are the same, set $\mathsf{rep}(\iota') \prec: \mathsf{rep}(\iota) \prec: \mathsf{owner}(\iota) = \mathsf{owner}(\iota')$. Thus ι' has access to representation of ι, but ι can be accessed wherever ι' can be. In addition ι' has its own representation which cannot be accessed by ι.
- to a system without containment, set $\mathsf{owner}(\iota) = \mathsf{rep}(\iota) = \varepsilon$ for all objects ι. When applied to individual objects this allows us to embed ordinary unprotected objects into any model which has ε;
- to regain our original owners-as-dominators, give each object a unique representation context, and ensure that for all objects that the representation context is directly inside the owner context. The last condition effectively states that

  $$\forall \iota. \neg \exists \iota'. \mathsf{rep}(\iota) \prec \mathsf{rep}(\iota') \prec \mathsf{owner}(\iota),$$

  where $a \prec b \mathrel{\widehat{=}} a \prec: b \wedge a \neq b$.

  In this case, the dominance property $(\star)$ is regained.

We explore some of these in greater detail in Chapter 6.

3.5 A Diagrammatic Containment Notation

This section reviews a diagrammatic notation for describing object graphs in the presence of representation containment. We have already been using it, but there is one

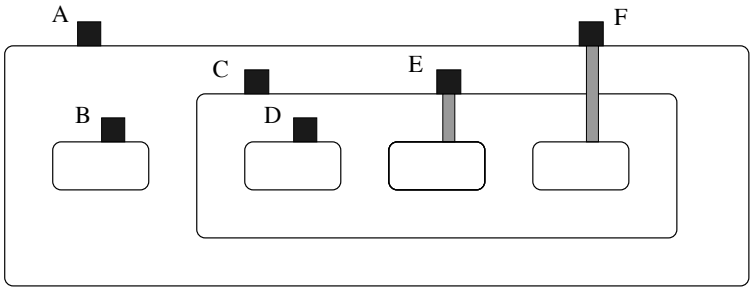


Figure 3.5: A Nest of Objects and Contexts

element we have not yet utilised. The notation accurately captures all the constraints between objects and contexts and enables a simple graphical test for determining whether a reference is valid.[1]

The diagrams rely on two reasonable assumptions about the containment model they represent. Firstly, we assume that no two objects share the same representation context, and that contexts are nested. That is, they form a tree or forest.

Now consider the objects and contexts represented in Figure 3.5, and the references added in Figure 3.6. The diagrams in these figures can be interpreted as follows:

1. Dark squares represent object identity, and are thus the target of references.
2. The rounded boxes denote contexts.
3. The nesting of rounded boxes corresponds to the nesting of contexts, with the inner boxes being "inside" outer boxes.
4. There may be a context ε which encloses a diagram corresponding to the maximal context.
5. The owner of an object is the context immediately enclosing it.
6. Each object has a representation context. Either:
   (a) the object sits on the outer side of the boundary of its representation context; or
   (b) the object is joined by a thin grey box (conduit) to its representation context. This case occurs when an object's owner and representation contexts are not adjacent in the ownership tree.

[1] For the most part, these diagrams are due to Ryan Shelswell, though we hesitate to call these *Shelswell Diagrams*.

The representation context also acts as the source of references contained in the object's fields.

7. Since representation contexts are unique per object, we consider the object and its representation context as a single entity.
8. An object's representation is the collection of objects *directly* inside the representation context.
9. The position of an object's owner contrains which objects can refer to it.
10. Dually, the object's representation context constrains which objects it can refer to. When the grey conduit is present, the object can access more objects, or dually, be accessed by more objects.
11. The simple rule governing the validity of references is:

    No reference can cross a context boundary from the outside to the inside.

The diagram in Figure 3.5 corresponds to the following data: context set $C = \{\varepsilon, A, B, C, D, E, F\}$ with inside relation given by $B, C \prec: A \prec: \varepsilon$ and $D, E, F \prec: C$; and owner and representation contexts for each object:

object	owner	rep
A	ε	A
B	A	B
C	A	C
D	C	D
E	A	E
F	ε	F

Object C could be a List object, D could be a Link object, and E could be an Iterator object. Furthermore, A could be a data structure implemented using a private List (C). It could be that the data structure A provides an iterator and that the List's iterator F is sufficient for that purpose, so it is used directly.

Figure 3.6 depicts various references which are allowed and one which is not. The valid ones either cross no context boundary or do so from inside to out. The invalid one, from B to D, crosses from outside to in and is therefore invalid. Considering the underlying data for this diagram, we see that $\mathsf{rep}(\mathtt{B}) = B \not\prec: \mathsf{owner}(\mathtt{D}) = C$, and thus the containment invariant is not satisfied.

We could extend the diagrams to allow the sharing of representation contexts between objects, but then things start getting messy, so we refrain from doing so.

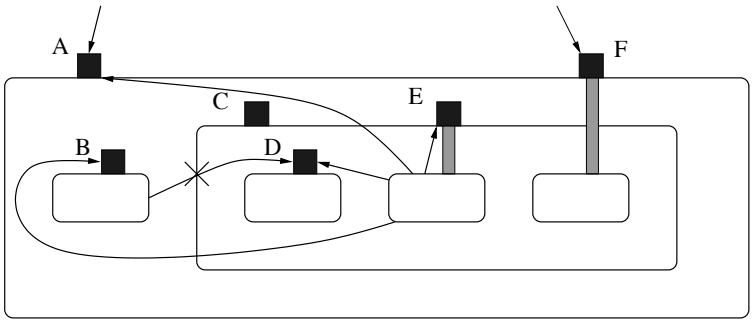


Figure 3.6: References must not cross a context boundary from the outside to the inside

3.6 Nesting and Privacy

Our containment model can be seen as a natural extension to the nesting and visibility of inner classes in Java (and Simula and Beta before that), except that the visibility restrictions here extend across the whole subtype hierarchy, rather than just to specific class names. That is, we restrict object type visibility not class name visibility. In Java there are only three choices concerning the visibility of a class name: it can be restricted to the surrounding context (class or package); to the outermost context but one (package); or can be unrestricted [93]. A more complete scheme allows the degree of privacy to be specified up to any surrounding class, by naming the outermost class which can see any given name.

Consider the following nest of classes which include precise visibility annotations. The annotation `public` specifies no restriction, whereas `private-up-to` specifies the outermost class which can see the name of the annotated class.

```
class A is public {
    class B is private-up-to A { }
    class C is private-up-to A {
        class D is private-up-to C { }
        class E is private-up-to A { }
        class F is public { }
    }
}
```

Visibility is governed by the nesting between classes. In fact, the nesting between objects in Figure 3.5 corresponds to the nesting between objects produced by these

classes. Thus `A` can see everything publicly visible as well as everything visible inside itself. Class `C` can see everything publicly visible, everything visible in `A`, and everything visible inside itself — which in this nest of classes is every class. Similarly `D`, `E`, and `F` can see every class, though `B` cannot see `D` because it is contained within `C`.

Another way of looking at this is that the visibility annotation on an element of this nest of classes corresponds to the position of the owner context in the ownership tree. Thus class `A` is visible everywhere and `B` and `C` are visible only inside `A`. Class `D` is only visible inside `C`, class `E` is only visible inside `A`, but `F` is visible everywhere.

All of this follows from the containment invariant, where the representation context is the position of a class in the nesting, and the owner context is the position of its visibility annotation.

This discussion is intended mainly as an analogy, but in Chapter 4 we exercise our calculus by encoding the above nest of classes and the constraints just discussed. Then in Chapter 6, we push the possibilities even further giving a number of different interpretations of the above syntax, so that the visibility restrictions apply to individual objects.

3.7 Conclusion

We have presented a simple model of containment based on object ownership using abstract units of ownership called contexts. Each object has an owner context, denoting its owner, and a representation context, denoting the owner of its representation. Object access is then governed by three things: the representation context of the source of a reference, the owner of the target object, and the nesting of contexts.

In the following two chapters, we extend the objects of Abadi and Cardelli's object calculus with an owner and representation context and present a type system which enforces the containment invariant between objects. We call the resulting types *ownership types*. The resulting calculus offers a static type system which indicates object ownership and provides a flexible means for limiting object visibility and controlling a system's object graph structure.

Chapter 4

Finitary Ownership Types

In the previous chapter we introduced the notions of ownership and containment, and indicated how these can be used to constrain object graph structure. In this and the next chapter we develop type systems which incorporate these notions and enforce the desired containment invariant. The system we present in this chapter adopts a minimal extension to Abadi and Cardelli's object calculus to illustrate how the containment invariant can be enforced by a type system. Apart from being interesting in its own right, this chapter also serves as a stepping stone to the more sophisticated calculus which follows in the next chapter and to the approach to proving its properties.

For the calculus presented in this chapter, ownership is based on a fixed and finite number of contexts, where the contexts and their ordering are prespecified for each expression. No contexts can be created during the evaluation of the expression. Finitary ownership is sufficient to approximate the containment imposed by Bokowski and Vitek's Confined Types [24] and Müller and Poetzsch-Heffter's Universes [144]. The former uses Java's packages as the units of ownership, whereas the latter uses classes (among other things). In both cases only a limited amount of nesting is exploited: both use some root context under which all packages or classes are nested. Rather than restrict ourselves in this manner, we develop the type system to allow arbitrary nesting between contexts.

The type system uses *permissions* [80] to control the well-formedness of expressions and types. To justify and explain this choice, Section 4.1 discusses permissions, elsewhere called *capabilities* [66], and their relationship with the well-known *type-and-effects* discipline [183]. Next in Section 4.2 we present the syntax for our calculus which includes two small extensions to a first-order variant of Abadi and Cardelli's object calculus: to each object we add an owner and a representation context; and

to methods we add arguments to distinguish between evaluation which occurs inside and outside of an object. The type system, which includes subtyping, appears in Section 4.3, and the dynamic semantics of the calculus in Section 4.4. Then in Section 4.5 we state the most important properties of the calculus, in particular, the soundness of its type system, highlighting the important consequence that object graphs underlying well-typed programs satisfy the containment invariant in Section 4.6. Section 4.7 presents some examples, including ones which capture the essence of the Confined Types [24] and Universes [144] proposals.

As the current calculus will be subsumed by that of the next chapter, including the proofs of its properties, we state its properties without proof.

4.1 Effects and Permissions

The type-and-effects discipline revolves around type systems which in addition to typing expressions also report computational effects which may occur during the evaluation of the expression. Such a type system can also be used to limit the effects which occur by ensuring that the effects reported fall within a prespecified limit [66]. We discuss this shift of perspective in terms of a type-and-effects system to motivate the style of type system employed herein.

Computational Effects Effects that occur during the evaluation of an expression include the creation, reading, and writing of memory locations, as well has other behaviour such as the raising of exceptions, thread creation, the acquisition of locks, and communication [145]. Even recursion has been considered a computational effect [82]. It seems that any behaviour of interest can be considered to be an effect.

A *type-and-effects system*, or just *effects system*, is a type system annotated with *effects* which summarise behaviour that may occur when an expression is evaluated [145]. An typical example judgement such as $E \vdash e : \tau \,\&\, \psi$ states, firstly, that in typing environment E the expression e has type τ, and also that when e is evaluated it will produce *at most* effect ψ, where ψ is some term in the language of effects. Such effects systems conservatively approximate the behaviour. An expression may not produce all the effects which a sound effects system determines, but it never produces more.

Effects are generally ordered and a subsumption rule is often included to increase the size of the effect that the type system calculates an expression may produce. A

simple rule of this form is:

$$\text{(Effect Sub)}\quad \frac{E \vdash e : \tau \,\&\, \psi \quad \psi \subseteq \psi'}{E \vdash e : \tau \,\&\, \psi'}$$

When an expression involves multiple subexpressions, the effect of the entire expression includes the effects of computing the subexpressions. For example, the following type rule for a **let** expression simply combines the two effects its subcomputations produce:

$$\downarrow \quad \text{(Effect Let)}\quad \frac{E \vdash e_1 : \tau_1 \,\&\, \psi_1 \quad E, x : \tau_1 \vdash e_2 : \tau_2 \,\&\, \psi_2}{E \vdash \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 : \tau_2 \,\&\, \psi_1 \cup \psi_2}$$

In this case the effects an expression produces is derived from the effects produced in its subexpressions, as the downwards arrow indicates.

From Effects to Permissions An effects system calculates the effects which may occur when an expression is evaluated. Shifting perspective slightly, we could specify an effect and then use the effects system to check that the effect produced by the expression under consideration does not exceed the specified effect. This prespecified effect can then be considered to be a *permission* [80] or *capability* [66] describing the maximum behaviour the expression is permitted to exhibit. If the behaviour exceeds the permission, then the expression does not type check. Thus permissions can be used to constrain effects.

Rather than having the type system calculate the effect that an expression may produce using the effects of its subexpressions, we can reformulate the inference rules to make it clear that the effects which a subexpression may exhibit are constrained by the effects we allow the expression to exhibit. For example, the above rule for **let** is reformulated as:

$$\uparrow \quad \text{(Almost-Permission Let)}\quad \frac{E \vdash e_1 : \tau_1 \,\&\, \psi \quad E, x : \tau_1 \vdash e_2 : \tau_2 \,\&\, \psi}{E \vdash \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 : \tau_2 \,\&\, \psi}$$

In this case we consider the permission information being pushed up the derivation tree, from an expression into the subexpressions to constrain their behaviour, as the arrow indicates.

To remind us that the permission is pre-supposed, rather than calculated or derived, we can place it on the left-hand side of the turnstile ($\vdash$) with the other assumptions. The **let** rule now becomes:

$$\text{(Permission Let)}\qquad \frac{E;\psi \vdash e_1 : \tau_1 \quad E, x : \tau_1; \psi \vdash e_2 : \tau_2}{E;\psi \vdash \textbf{let}\, x = e_1\, \textbf{in}\, e_2 : \tau_2}$$

Of course subsumption would be modified accordingly:

$$\text{(Permission Allow)}\qquad \frac{E;\psi \vdash e : \tau \quad \psi \subseteq \psi'}{E;\psi' \vdash e : \tau}$$

The final aspect of this shift of perspective from effects to permissions deals with effects masking. In an effects system, masking corresponds to forgetting some effects which can occur in an expression and just reporting a subset [184]. Masking is usually employed at some boundary, such as that introduced by a `letregion` block [188].

One of our contributions is the adaptation of this idea to the setting of encapsulated objects. In our system, an object has a boundary between its inside and outside. The objects an object can access differ from the objects which can access it. More permission is required inside the object. The additional permission, which governs access to the representation, is not required by those wishing to access the object. This is our new perspective on masking — an object masks the permissions required to access the contents of its fields.

We now begin our account of ownership types.

4.2 A Calculus with Finite Ownership

The object calculus on which we base our calculus is a variant of Abadi and Cardelli's first-order, typed, imperative object calculus [3]. It is simple enough to illustrate the mechanisms underlying ownership types, but powerful enough to model containment based on a finite number of contexts, as in Confined Types [24] and, in part, Universes [144].

There are two main extensions we make to the object calculus. Firstly, to capture the nesting between objects and the consequent restrictions, we add contexts as discussed in the previous chapter. The containment of representation induces a different

view inside an object than from outside of it, in particular, certain objects accessible inside the object are not necessarily accessible from outside the object.

The second extension is designed to support the first. Methods accept arguments and the type system ensures that all parameters are supplied when a method is selected. This allows us to distinguish between evaluation which occurs inside an object and that which occurs outside.

We will now illustrate. Consider the following object:

$$[d = 4, calc = \varsigma(self, param)param + self.d]$$

The method *calc* takes a single formal parameter *param*; the one named *self* is of course the self parameter. When this method is selected a single parameter must be supplied. The following reduction sequence illustrates that this is sufficient to capture the notions of evaluation inside and outside of an object.

$$[d = 4, calc = \varsigma(self, param)param + self.d].calc\langle 10\rangle \tag{4.1}$$
$$\rightsquigarrow \{10 + [d = 4, calc = \varsigma(self, param)param + self.d].d\} \tag{4.2}$$
$$\rightsquigarrow \{10 + 4\} \tag{4.3}$$
$$\rightsquigarrow \{14\} \tag{4.4}$$
$$\rightsquigarrow 14 \tag{4.5}$$

Here we artificially use braces "{ }" to enclose that part of the evaluation which occurs on the inside of an object. On the line (4.1) the method *calc* is called with actual parameter 10, which was evaluated outside of the object. When the method is selected, 10 is substituted for the formal parameter and it crosses from outside of the object to its inside. Evaluation on lines (4.2) and (4.3) take place inside the object, resulting in the value in (4.4). In the transition to line (4.5) the result 14 escapes the object boundary from the inside to the outside.

So there are three places where values cross the boundary of an object, as arguments of a method and in return values, and of course in method update. The type system must ensure that enough permission is held both inside and outside of an object for these transfers to occur.

Note that the object calculus cannot make this distinction. Methods which take parameters are encoded using functions. The function closure may be applied, which is fine, but it could also be installed as part of another object, which is not acceptable from our perspective because the closure may contain a reference and thus an access point to representation which the object should not have access to.

p	$\in$	CONTEXT	$=$	C
K	$\in$	PERMISSION	$::=$	$\langle p \rangle$
			$\mid$	$\langle p\uparrow \rangle$
A,B,C	$\in$	TYPE	$::=$	$[l_i : \Theta_i{}^{i\in 1..n}]^p_q$
Θ	$\in$	METHODTYPE	$::=$	$A \mid A \to \Theta$
x,s	$\in$	VAR		
ι	$\in$	LOCATION		
u,v	$\in$	VALUE	$::=$	x
			$\mid$	ι
a,b	$\in$	EXPRESSION	$::=$	v
			$\mid$	o
			$\mid$	$v.l\langle\Delta\rangle$
			$\mid$	$v.l \Leftarrow \varsigma(s : A,\Gamma)b$
			$\mid$	**let** $x : A = a$ **in** b
o	$\in$	OBJECT	$::=$	$[l_i = \varsigma(s_i : A_i,\Gamma_i)b_i{}^{i\in 1..n}]^p_q$ *(l_i distinct)*
Γ	$\in$	PARAM	$::=$	$\emptyset \mid x : A,\Gamma$
Δ	$\in$	ARGVAL	$::=$	$\emptyset \mid v,\Delta$
σ	$\in$	STORE	$::=$	$\emptyset$
			$\mid$	$\sigma,\iota \mapsto o$
t	$\in$	CONFIG	$::=$	(σ,a)
			$\mid$	WRONG

Figure 4.1: The Syntax

Remark 4.1 *Some variation of the additional syntax, $\{\ \}$, could be added to the calculus to highlight the difference between inside and outside during evaluation. Indeed this would be necessary had we not chosen to present our semantics in a big-step style which hides such intimate details.*

4.2.1 Syntax

Figure 4.1 gives the syntax for *objects*, *permissions*, *types* (including *method types*), *values*, *expressions*, and *configurations*. We describe each in turn.

Objects We adopt the model discussed in the previous chapter and modify objects accordingly to include both an owner and representation context. Objects now have the syntax $[l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i\in 1..n}]^p_q$. The owner context is p and the representation context is q. Each s_i is the parameter used in the method body b_i to refer to the current instance of the object, where A_i is its type. The Γ_i are the formal parameters to the method. These are a list of term variables with their types.

Permissions for Object Access A permission denotes a collection of contexts. They are used to constrain the objects and locations accessible in an expression. An object $[\ldots]^p_q$ is accessible given permission K only if p is one of the contexts in K. Permissions only constrain objects and locations based on the top level owners, that is, for example, object $[\ldots l = [\ldots]^{p'}_{q'} \ldots]^p_q$ is accessible only if p is accessible. Access to the object in field l would require additional permission, namely some permission containing context p'. The representation context q governs which contexts are accessible in the methods of this object.

There are two kinds of permission: $\langle p \rangle$ and $\langle q\uparrow \rangle$. The permission $\langle p \rangle$ allows access to the single context p, that is objects with owner p. We refer to this as a *point* permission. Permission $\langle q\uparrow \rangle$, called an *upset* permission, denotes the set $\{p \mid q \prec: p\}$.

The subpermission relation corresponds to subset on the context sets underlying each permission. This is used to allow expressions which require a certain permission to be valid whenever a larger permission is provided.

Example 4.2 *Let* $C = \{\mathsf{Bob}, \mathsf{Alice}, \mathsf{World}\}$*, where* $\mathsf{Bob} \prec: \mathsf{World}$ *and* $\mathsf{Alice} \prec: \mathsf{World}$*. Using this data we have that* $\langle \mathsf{Bob} \rangle \equiv \{\mathsf{Bob}\}$ *and* $\langle \mathsf{Bob}\uparrow \rangle \equiv \{\mathsf{Bob}, \mathsf{World}\}$*. Access to objects owned by* Bob*, for example, is restricted. The only objects which can access an object owned by* Bob *are those expressions which have at least permission* $\langle \mathsf{Bob} \rangle$*.*

Types At this stage types, TYPE, include only object types. The type $[l_i : \Theta_i{}^{i\in 1..n}]^p_q$ lists the names and types of the object's methods, as well as the owner context p (superscript) and the representation context q (subscript). Objects of this type can only be accessed in expressions possessing at least permission $\langle p \rangle$. Similarly, this type can only be constructed when at least $\langle p \rangle$ is given with the typing environment.

Method Types As mentioned at the start of this section, we wish to distinguish between evaluation inside and outside an object. To do this we require methods to take all of their arguments at once. The syntactic category METHODTYPE, for method types, supports this constraint. The return type of a method must be a TYPE. The arguments of a method have types from the TYPE syntactic category; each gives rise to a METHODTYPE, $A \to \Theta$, where Θ is a method type. For example, a method which has formal parameters of types A and B and returns a value of type C has method type $A \to B \to C$. Note that the elements of METHODTYPE are not in TYPE.

Expressions Expressions are presented in a variant of the *named form* [175]. This amounts to the requirement that the result of (almost) every evaluation step be bound to a variable which is subsequently used to refer to the result of this computation. While this form does not change the expressiveness of the calculus, it simplifies the statement of its semantics and the proof of its properties.

The language is imperative, since aliasing problems are only apparent in the presence of mutable state. Objects evaluate to locations, ι, which are subsequently used to refer to the object in the store. Locations are the only possible result of computation, although they do not appear in the expressions a programmer writes.

Method selection, $v.l\langle\Delta\rangle$, takes a collection of actual parameters, Δ, which are a sequence of values.

Method update, $v.l \Leftarrow \varsigma(s : A, \Gamma)b$, replaces the method labelled l in the object v with that defined by $\varsigma(s : A, \Gamma)b$. Again s is the self parameter and Γ are the formal parameters to the method. As usual, a method can be treated as a field if $s \notin FV(b)$, b is a value (in this calculus, this means that b is a variable), and additionally that $\Gamma = \emptyset$ [3]. Thus we do not distinguish between fields and methods.

Let expressions, **let** $x : A = a$ **in** b, are used for local declarations and to link computations together.

To keep our system simple, we omit functions which can be encoded as methods.

Example 4.3 *Assume the contexts from Example 4.2. The following expression creates an object residing in context* Bob*, and then assigns it to a field of an object which is owned by* World *(and accessible to potentially all objects):*

$$x \;\widehat{=}\; \textbf{let } pr = [\ldots]^{\mathsf{Bob}}_{\mathsf{Bob}} \textbf{ in } [priv = \varsigma(s : A)pr]^{\mathsf{World}}_{\mathsf{Bob}}$$

Because the representation context of the object x is Bob*, it has access to* Bob *objects. The object x does not have access to any objects owned by* Alice*. Thus the following*

is illegal:

$$\textbf{let } ins = [\ldots]^{\mathsf{Alice}}_{\mathsf{Alice}} \textbf{ in } x.priv \Leftarrow \varsigma(s : A)ins$$

Stores and Configurations The store, σ, maps locations to objects. Locations are created when objects are evaluated. A configuration represents a snapshot of the evaluation. It consists of an expression to be evaluated, a, and a store, σ. The configuration WRONG corresponds to the result of stuck evaluation, that is, evaluation which cannot proceed because, for example, an object has no method corresponding to the selected method name.

4.3 Types for Ownership

How the type system enforces the containment invariant Expressions are typed against a permission. This permission restricts which contexts can appear in the top-level owner position of objects and locations. Locations and objects whose owners are not in the permission cannot be accessed. The containment invariant governs this access, using an object's representation context to constrain which contexts an object can access.

Recall that the containment invariant states a necessary condition for a reference from ι to ι' to exist:

$$\iota \rightarrow \iota' \;\Rightarrow\; \mathsf{rep}(\iota) \prec\!: \mathsf{owner}(\iota').$$

We consider a reference for ι to ι' to exist whenever $\iota \mapsto [l_i = \varsigma(s_i : A)b_i{}^{i\in 1..n}]^p_q$ is a location-object binding in some store and ι' is a location appearing in one of the method bodies b_i, where ι' is not within some other object expression. Noting that $\mathsf{rep}(\iota) = q$, the containment invariant can be transformed to give an upper bound on the objects ι can refer to:

$$\{\iota' \mid q \prec\!: \mathsf{owner}(\iota')\}. \tag{4.6}$$

Using the definition of $\langle q\uparrow\rangle$ from above, (4.6) becomes:

$$\{\iota' \mid \mathsf{owner}(\iota') \in \langle q\uparrow\rangle\}. \tag{4.7}$$

Now consider the object at location ι' which has owner p'. Access to this object requires permission $\langle p'\rangle$, which corresponds to the singleton set $\{p'\}$. Thus ι' is in the set (4.7) if and only if

$$\langle p'\rangle \subseteq \langle q\uparrow\rangle. \tag{4.8}$$

This condition is enforced by our type system. To see how, consider the following simplified version of our object typing rule (which is yet to come):

$$\frac{E, s_i : A; \langle q\uparrow\rangle \vdash b_i : C_i \quad \forall i \in 1..n}{E; \langle p\rangle \vdash [l_i = \varsigma(s_i : A) b_i{}^{i\in 1..n}]^p_q : A} \text{ (Val Object-Simplified) (where } A \equiv [l_i : C_i{}^{i\in 1..n}]^p_q)$$

The conclusion states, among other things, that the permission required to access this object is $\langle p\rangle$, where p is the object's owner. Similar permission is required to access a location. The premises each state that the permission governing access in the method bodies b_i is $\langle q\uparrow\rangle$, where q is the object's representation context. The objects and locations accessible, therefore, are at most those whose owner satisfies condition (4.8). Therefore, the only locations accessible in a method body are those permitted by the containment invariant.

Apart from giving types to the appropriate constructs in the expected manner, the other rules in the type system also propagate or preserve the constraints we require. In particular, subtyping does not allow the owner information present in a type to be lost — omitting this property precluded subtyping from our original Ownership Types proposal [61].

The type system presented in this chapter contains a number of artifacts which are present to support the calculus presented in Chapter 5. We do not simplify these to make the transition from this type system to the next easier.

The Details The type of an expression depends on a *typing environment*, E, which maps program variables and locations to their types. The typing environment is organised as a sequence of bindings, where $\emptyset$ denotes the empty environment:

$$E \quad ::= \quad \emptyset \mid E, x : A \mid E, \iota : A$$

The syntax for method formal parameters, Γ, is just a subset of this syntax, which allows them to be treated as environments in the type rules.

We define the type system using the judgements described in Figure 4.2. Judgements concerning constructs with no free variables do not require a typing environment in their specification. All judgements concerned with types and expressions are formulated with respect to a permission K.

To simplify the handling of environments in what follows, we employ the notation such as $x : A \in E$ to extract assumptions such as $x : A$ from a typing environment E.

$E \vdash \diamond$	*E is a well-formed typing environment*
$K \subseteq K'$	*K is a subpermission of* K'
$K \vdash A$	*A is a well-formed type given K*
$K \vdash A <: B$	*A is a subtype of B given K*
$E; K \vdash a : A$	*a is a well-typed expression of type A in E given K*
$K \vdash \Theta$ *meth*	Θ *is a well-formed method type given K*
$E; K \vdash (\Theta)(\Delta) \Rightarrow C$	*The actual parameters* Δ *match method type* Θ *whose return type is C in E given K*
$E \vdash \sigma$	σ *is a well-formed store in E*
$E; K \vdash (\sigma, a) : A$	(σ, a) *is a well-typed configuration with type A in E given K*

Figure 4.2: Judgements

Implicit in this notation is the assumption that typing environment E is well-formed, that is, $E \vdash \diamond$.

The function $\mathsf{dom}(E)$ extracts the variables bound in a typing environment.

Definition 4.4 (Domain of an Environment) *The domain of an environment* $\mathsf{dom}(E)$ *is defined as:*

$$\begin{aligned} \mathsf{dom}(\emptyset) &\mathrel{\widehat{=}} \emptyset \\ \mathsf{dom}(E, \iota : A) &\mathrel{\widehat{=}} \mathsf{dom}(E) \cup \{\iota\} \\ \mathsf{dom}(E, x : A) &\mathrel{\widehat{=}} \mathsf{dom}(E) \cup \{x\} \end{aligned}$$

The function $\mathsf{dom}(\,)$ *also applies to* Γ.

The definition of free variables, $FV(\,)$, determines the scoping of variables; bound variables are those which appear in a expression but are not free. With this definition substitution can be defined in a straightforward manner [3, 105].

Definition 4.5 (Free Variables)

$$\begin{aligned} \mathrm{FV}(x) &\mathrel{\widehat{=}} \{x\} \\ \mathrm{FV}(\iota) &\mathrel{\widehat{=}} \emptyset \\ \mathrm{FV}(v.l\langle\Delta\rangle) &\mathrel{\widehat{=}} \mathrm{FV}(v) \cup \mathrm{FV}(\Delta) \\ \mathrm{FV}(\emptyset) &\mathrel{\widehat{=}} \emptyset \\ \mathrm{FV}(v, \Delta) &\mathrel{\widehat{=}} \mathrm{FV}(v) \cup \mathrm{FV}(\Delta) \end{aligned}$$

$$\frac{}{\emptyset \vdash \diamond}\ (\text{Env } \emptyset) \qquad \frac{K \vdash A \quad x \notin \mathsf{dom}(E)}{E, x : A \vdash \diamond}\ (\text{Env } x) \qquad \frac{\langle p \rangle \vdash [l_i : \Theta_i^{\,i \in 1..n}]_q^p \quad \iota \notin \mathsf{dom}(E)}{E, \iota : [l_i : \Theta_i^{\,i \in 1..n}]_q^p \vdash \diamond}\ (\text{Env Location})$$

Figure 4.3: Well-formed Environments

$$\begin{aligned}
\mathrm{FV}(v.l \Leftarrow \varsigma(s : A, \Gamma) b) &\mathrel{\widehat{=}} \mathrm{FV}(v) \cup \mathrm{FV}(\varsigma(s : A, \Gamma) b) \\
\mathrm{FV}(\textbf{\textit{let}}\ x : A = a\ \textbf{\textit{in}}\ b) &\mathrel{\widehat{=}} \mathrm{FV}(a) \cup (\mathrm{FV}(b) \backslash \{x\}) \\
\mathrm{FV}([l_i = \varsigma(s_i : A_i, \Gamma_i) b_i^{\,i \in 1..n}]_q^p) &\mathrel{\widehat{=}} \bigcup_{i \in 1..n} \mathrm{FV}(\varsigma(s_i : A_i, \Gamma_i) b_i) \\
\mathrm{FV}(\varsigma(s : A, \Gamma) b) &\mathrel{\widehat{=}} \mathrm{FV}(b) \backslash (\{s\} \cup \mathit{dom}(\Gamma))
\end{aligned}$$

We now present the type system.

Well-formed Environments The rules in Figure 4.3 define the well-formedness of typing environments. Adding a variable to an environment requires that it not be already present and that its type can be well-formed given some permission (Env x). The permission does not matter at this point, except for ensuring that the variable's type is well-formed, but it will be required when x is used in an expression. (Env Location) specifies that locations have object type.

Well-formed Permission and Subpermissions All permissions in PERMISSION, that is $\langle p \rangle$ and $\langle p\uparrow \rangle$ for each $p \in C$, are valid. Figure 4.4 defines the subpermission relation, $K \subseteq K'$ in the obvious manner given the set-theoretic description of permissions earlier in this chapter.

A point permission is a subpermission of the corresponding upset permission by rule (SubPerm p). The rule (SubPerm $\prec$:) lifts the nesting relation between contexts to upset permissions. This is based on the following reasoning: if $\langle p\uparrow \rangle \subseteq \langle q\uparrow \rangle$, then $p' \in \langle p\uparrow \rangle$, which implies that $p \prec: p'$. Thus $q \prec: p'$, hence $p' \in \langle q\uparrow \rangle$.

The subpermission relation is also reflexive and transitive, by (SubPerm Refl) and (SubPerm Trans).

Example 4.6 *Continuing Example 4.2, we have* $\langle \mathsf{Bob} \rangle \equiv \{\mathsf{Bob}\}$, $\langle \mathsf{World} \rangle \equiv \{\mathsf{World}\}$, *and* $\langle \mathsf{Bob}\uparrow \rangle \equiv \{\mathsf{Bob}, \mathsf{World}\}$. *Hence,* $\langle \mathsf{Bob} \rangle \subseteq \langle \mathsf{Bob}\uparrow \rangle$, *and also* $\langle \mathsf{World} \rangle \subseteq \langle \mathsf{Bob}\uparrow \rangle$.

$$\frac{p \in C}{\langle p \rangle \subseteq \langle p\uparrow \rangle}\ (\text{SubPerm } p) \qquad \frac{q \prec: p}{\langle p\uparrow \rangle \subseteq \langle q\uparrow \rangle}\ (\text{SubPerm } \prec:) \qquad \frac{}{K \subseteq K}\ (\text{SubPerm Refl}) \qquad \frac{K \subseteq K' \quad K' \subseteq K''}{K \subseteq K''}\ (\text{SubPerm Trans})$$

Figure 4.4: Sub-permission

$$\frac{\langle q\uparrow \rangle \vdash \Theta_i\ meth \qquad \forall i \in 1..n \quad q \prec: p}{\langle p \rangle \vdash [l_i : \Theta_i{}^{i \in 1..n}]^p_q}\ (\text{Type Object})\ (l_i \text{ distinct}) \qquad \frac{K \vdash A \quad K \subseteq K'}{K' \vdash A}\ (\text{Type Allow})$$

Figure 4.5: Well-formed Types

Well-formed Types The rules in Figure 4.5 define well-formed types. The well-formedness of types depends upon a permission, which is the permission required to access values of that type.

The justification for (Type Object) was given earlier. The method types of an object type must be well-formed given the permission $\langle q\uparrow \rangle$, where q is the representation context. This limits the form of method types. The permission of an expression wishing to access an object of this type must be at least $\langle p \rangle$. The condition $q \prec: p$ implies that $\langle p \rangle \subseteq \langle q\uparrow \rangle$ which will ensure that an object can access itself.

The rule (Type Allow) states that types which are well-formed against some permission are well-formed for any larger permission.

Method Types Method types resemble function types. The type rules are presented in Figure 4.6. Together the rules (Type Return) and (Type Arrow) allow a method type to be a function with zero or more arguments, where each argument and the return type are ordinary types.

Well-formed Subtyping The subtyping rules are given in Figure 4.7. By the rule (Sub Object), an object type is a subtype of one containing additional methods, but neither the owner nor the representation context may vary. Method types are invariant, though this could be changed in a straightforward manner by adding variance annotations [3]. Reflexivity and transitivity of subtyping follow trivially from this

$$\frac{K \vdash A}{K \vdash A\ meth}\ \text{(Type Return)} \qquad \frac{K \vdash A \quad K \vdash \Theta\ meth}{K \vdash A \to \Theta\ meth}\ \text{(Type Arrow)}$$

Figure 4.6: Well-formed Method Type

$$\frac{\langle q\uparrow\rangle \vdash \Theta_i\ meth \quad \forall i \in 1..n+m \quad q \prec: p}{\langle p\rangle \vdash [l_i : \Theta_i{}^{i\in 1..n+m}]^p_q <: [l_i : \Theta_i{}^{i\in 1..n}]^p_q}\ \text{(Sub Object)}\ (l_i \text{ distinct}) \qquad \frac{K \vdash A <: B \quad K \subseteq K'}{K' \vdash A <: B}\ \text{(Sub Allow)}$$

Figure 4.7: Subtyping

rule. Finally, (Sub Allow) states that any subtype relation which is valid given some permission is valid with any larger permission.

Well-typed Expressions Figure 4.8 defines well-typed expressions. Expressions are typed against a typing environment and a permission. The permission bounds the owners of objects and locations permitted in the given expression. The type rules depend on the following auxiliary function, $\lfloor \Gamma \rfloor_C$, which converts the method arguments Γ and the return type C into a method type Θ:

Definition 4.7 (Method Type Conversion $\lfloor \Gamma \rfloor_C$)

$$\begin{aligned} \lfloor \emptyset \rfloor_C &\mathrel{\widehat{=}} C \\ \lfloor x : A, \Gamma \rfloor_C &\mathrel{\widehat{=}} A \to \lfloor \Gamma \rfloor_C \end{aligned}$$

For example, if a method has the formal parameter list $x : A, y : B, \emptyset$ and return type C, the corresponding method type is $\lfloor x : A, y : B, \emptyset \rfloor_C \equiv A \to B \to C$.

Variable typing is by environmental assumption (Val x), though sufficient permission to construct its type is required. Locations are similarly typed by assumption, where the permission required is at least the point permission for the owner in the location's type (Val Location).

In (Val Object), the body b_i of each method $\varsigma(s_i : A, \Gamma_i)b_i$ is typed against an environment extended with the self parameter s_i having type A, the self type for the

$$\frac{x : A \in E \quad K \vdash A}{E;K \vdash x : A}\ (\text{Val } x) \qquad \frac{\iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q \in E}{E;\langle p\rangle \vdash \iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q}\ (\text{Val Location})$$

$$\frac{E, s_i : A, \Gamma_i; \langle q\uparrow\rangle \vdash b_i : C_i \quad \forall i \in 1..n}{E;\langle p\rangle \vdash [l_i = \varsigma(s_i : A, \Gamma_i) b_i{}^{i\in 1..n}]^p_q : A}\ (\text{Val Object}) \ (\text{where } A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_q \text{ and } \Theta_i \equiv \lfloor \Gamma_i \rfloor_{C_i})$$

$$\frac{E;K \vdash v : [l_i : \Theta_i{}^{i\in 1..n}]^p_q \quad E;K \vdash (\Theta_j)(\Delta) \Rightarrow C_j \quad j \in 1..n}{E;K \vdash v.l_j\langle\Delta\rangle : C_j}\ (\text{Val Select})$$

$$\frac{E;K \vdash v : A \quad E, s : A, \Gamma_j; K' \vdash b : C_j \quad K' \subseteq \langle q\uparrow\rangle \quad K' \subseteq K \quad j \in 1..n}{E;K \vdash v.l_j \Leftarrow \varsigma(s : A, \Gamma_j) b : A}\ (\text{Val Update}) \ (\text{where } A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_q \text{ and } \Theta_j \equiv \lfloor \Gamma_j \rfloor_{C_j})$$

$$\frac{E;K \vdash a : A \quad E, x : A; K \vdash b : B}{E;K \vdash \textbf{let } x : A = a \textbf{ in } b : B}\ (\text{Val Let}) \qquad \frac{E;K \vdash a : A \quad K' \vdash A <: B \quad K \subseteq K'}{E;K' \vdash a : B}\ (\text{Val Subsumption})$$

Figure 4.8: Well-typed Expressions

object, and with the formal parameter list Γ_i. The method type Θ_i is constructed from the return type of the method body C_i and the types of the formal parameters. The permission $\langle p\rangle$ is required to create an object with owner context p. The method is typed against the upset permission $\langle q\uparrow\rangle$, where q is the representation context. The precise role these play has already been discussed. Access to self s_i is permitted because $A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_q$ is present in environment E, and therefore well-formed (see discussion of the rule (Type Object)).

(Val Select) requires that the target have an object type with the appropriate method present. The clause $E;K \vdash (\Theta_j)(\Delta) \Rightarrow C_j$ checks that the arguments are well-typed and accessible given permission K. It also states that the return type is C_j. This form of judgement will be described below.

(Val Update) requires that the target have an object type with the appropriate method present. Firstly, the formal parameter list Γ_j and return type C_j of the new method must form the method's type Θ_j, that is, $\lfloor \Gamma_j \rfloor_{C_j} = \Theta_j$. The new method body is typed against a self s with the type of the object being updated and the formal parameters Γ_j. The new method body must have type C_j. The given permission K' is at most the intersection between the contexts accessible inside the object, $\langle q\uparrow\rangle$, and the accessible contexts in the surrounding expression, K. This 'intersection' of permissions prevents any otherwise inaccessible locations being added into the object, while maintaining the constraints on the expression performing the method update.

The type rule (Val Let) follows the standard pattern, except that it also carries the permission through to the subexpressions.

Finally, (Val Subsumption) allows an expression of one type to be given a supertype, as usual, and to be used with a larger permission.

We illustrate how the type system enforces the containment invariant by showing some type derivations based on Example 4.3. Recall that $\mathsf{Bob}, \mathsf{Alice} \prec: \mathsf{World}$.

Example 4.8 (A positive example) *Consider the expression*

$$x \mathrel{\widehat{=}} \textbf{let } pr = [\ldots]^{\mathsf{Bob}}_{\mathsf{Bob}} \textbf{ in } [priv = \varsigma(s:A)pr]^{\mathsf{World}}_{\mathsf{Bob}}$$

To simplify matters, assume that x has type $A \mathrel{\widehat{=}} [priv : [l_i : \Theta_i{}^{i\in 1..n}]^{Bob}_{Bob}]^{World}_{Bob}$ *and that* $E; \langle Bob\rangle \vdash [\ldots]^{Bob}_{Bob} : [l_i : \Theta_i{}^{i\in 1..n}]^{Bob}_{Bob}$.

The type derivation of the right hand side of the ***let*** *expression is:*

$$\dfrac{\dfrac{E, pr : [l_i : \Theta_i{}^{i\in 1..n}]^{Bob}_{Bob}, s : A; \langle Bob\rangle \vdash pr : [l_i : \Theta_i{}^{i\in 1..n}]^{Bob}_{Bob} \quad \langle Bob\rangle \subseteq \langle Bob\uparrow\rangle}{E, pr : [l_i : \Theta_i{}^{i\in 1..n}]^{Bob}_{Bob}, s : A; \langle Bob\uparrow\rangle \vdash pr : [l_i : \Theta_i{}^{i\in 1..n}]^{Bob}_{Bob}}\ \text{(Val Subsumption)}}{E, pr : [l_i : \Theta_i{}^{i\in 1..n}]^{Bob}_{Bob}; \langle World\rangle \vdash [priv = \varsigma(s:A)pr]^{World}_{Bob} : A}\ \text{(Val Object)}$$

This works because the owner of the field priv is a context which is included in the permission $\langle Bob\uparrow\rangle$. *Indeed, the owner of the field priv could be any context p for which* $Bob \prec: p$.

Example 4.9 (A negative example) *In Example 4.3 we stated that the following was illegal:*

$$\textbf{let } ins = [\ldots]^{\mathsf{Alice}}_{\mathsf{Alice}} \textbf{ in } x.priv \Leftarrow \varsigma(s:A)ins$$

The first reason is that the types of the fields are not compatible, since $\mathsf{Alice} \neq \mathsf{Bob}$*, and thus* $[l_i : \Theta_i{}^{i\in 1..n}]^{\mathsf{Alice}}_{\mathsf{Alice}} \neq [l_i : \Theta_i{}^{i\in 1..n}]^{\mathsf{Bob}}_{\mathsf{Bob}}$*. Furthermore, subtyping allows neither the object nor the representation context to vary, so neither of these two types is a subtype of the other.*

It would be impossible for the object x to have a field of type $[l_i : \Theta_i{}^{i\in 1..n}]^{\mathsf{Alice}}_{\mathsf{Alice}}$*, since the resulting type would not be well-formed. Thus no object can be constructed having type* $[l_i : \Theta_i{}^{i\in 1..n}]^{\mathsf{Alice}}_{\mathsf{Alice}}$*, as the following derivation shows:*

$$\dfrac{\dfrac{\langle \mathsf{Alice}\rangle \vdash [l_i : \Theta_i{}^{i\in 1..n}]^{\mathsf{Alice}}_{\mathsf{Alice}} \qquad \dfrac{\langle \mathsf{Alice}\rangle \subseteq \langle \mathsf{Alice}\uparrow\rangle \qquad \dfrac{\mathsf{Bob} \not\prec: \mathsf{Alice}}{\langle \mathsf{Alice}\uparrow\rangle \not\subseteq \langle \mathsf{Bob}\uparrow\rangle}\ (\text{SubPerm} \prec:)}{\langle \mathsf{Alice}\rangle \not\subseteq \langle \mathsf{Bob}\uparrow\rangle}\ (\text{SubPerm Trans})}{\langle \mathsf{Bob}\uparrow\rangle \not\vdash [l_i : \Theta_i{}^{i\in 1..n}]^{\mathsf{Alice}}_{\mathsf{Alice}}}\ (\text{Type Allow})}{\langle \mathsf{World}\rangle \not\vdash [non : [l_i : \Theta_i{}^{i\in 1..n}]^{\mathsf{Alice}}_{\mathsf{Alice}}]^{\mathsf{World}}_{\mathsf{Bob}}}\ (\text{Type Object})$$

These two examples demonstrate that typing reduces to the ordering on contexts, in particular between the representation context of the source and the owner context of the target of a reference, following the pattern specified by the containment invariant.

Well-typed Actual Parameter Lists The type rules for checking the conformance of actual parameter lists are given in Figure 4.9.[1] The judgement $E;K \vdash (\Theta_j)(\Delta) \Rightarrow C_j$ guarantees that the actual parameters Δ are correct in number and type. It also states that the return type is C_j. The argument and return types must all be typable given permission K, ensuring that sufficient permission is held both to pass arguments to the method and to access its return value.

Underlying this fragment are the usual rules for function application. In fact, $E;K \vdash (\Theta)(\Delta) \Rightarrow C$ resembles function application, where partial application is disallowed.

The following examples illustrate the behaviour of these type rules. When the parameter list conforms, a typing derivation can be found:

Example 4.10 (Conforming Parameter List) *Given method type* $\Theta \equiv A \to B \to C$ *and actual parameter list* $\Delta \equiv a, b, \emptyset$*, where* $a : A$ *and* $b : B$*, the following typing*

[1] This form of rule may seem a little odd and more work than required for something which could be done more simply using a tuple of arguments. When the next chapter introduces two more kinds of variables into the parameter list, this form becomes arguably more pleasant and more conducive to inductive arguments.

$$\frac{E \vdash \diamond \quad K \vdash C}{E;K \vdash (C)(\emptyset) \Rightarrow C}\ \text{(Arg Empty)} \qquad \frac{E;K \vdash v : A \quad E;K \vdash (\Theta)(\Delta) \Rightarrow C}{E;K \vdash (A \to \Theta)(v,\Delta) \Rightarrow C}\ \text{(Arg Val)}$$

Figure 4.9: Well-typed Actual Parameters

derivation shows that the actual parameter list conforms to the method type:

$$\frac{E;K \vdash a : A \qquad \dfrac{E;K \vdash b : B \quad E;K \vdash (C)(\emptyset) \Rightarrow C}{E;K \vdash (B \to C)(b,\emptyset) \Rightarrow C}}{E;K \vdash (A \to B \to C)(a,b,\emptyset) \Rightarrow C}$$

There are a number of ways a parameter list can fail to type check: too many parameters; too few parameters; type of an actual parameter does not match expected type; or that sufficient permission is not present to access argument values or the return type. The following derivation include two of these:

Example 4.11 (Non-Conforming Parameter List) *Given method type $\Theta \equiv A \to B \to C$ and actual parameter list $\Delta \equiv b,\emptyset$, where $b : B$ and B is not a subtype of A, the following demonstrates that the actual parameter list does not conform to the method type:*

$$\frac{E;K \not\vdash b : A \quad E;K \not\vdash (B \to C)(\emptyset) \Rightarrow ?}{E;K \not\vdash (A \to B \to C)(b,\emptyset) \Rightarrow ?}$$

A derivation cannot be found, firstly because the first actual parameter does not have a type which conforms to A, and secondly because $B \to C$ is not a well-formed type, indicating that insufficient actual parameters were supplied.

Stores and Configurations The type rules for stores and configurations are given in Figure 4.10. Store typing enforces that only objects are stored in a location and that the object's type is the same as that of the location. Configuration typing is performed in an environment consisting of the types for locations only, which means that the term a must be closed.

Now we can present the operational semantics, indicating not only where computation goes right, but also where it goes wrong, so that we can latter assert the soundness of the type system.

$$\frac{E;\langle p\rangle \vdash o : [l_i : \Theta_i{}^{i\in 1..n}]^p_q \quad \iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q \in E \quad \forall \iota \mapsto o \in \sigma}{E \vdash \sigma} \text{ (Val Store)}$$

$$\frac{E \vdash \sigma \quad E;K \vdash a : A \quad \mathsf{dom}(\sigma) = \mathsf{dom}(E)}{E;K \vdash (\sigma, a) : A} \text{ (Val Config)}$$

Figure 4.10: Well-typed Stores and Configurations

4.4 Dynamic Semantics

The operational semantics of the calculus are presenting in a big-step, substitution-based style in Figure 4.11. Fundamentally it differs little from the object calculus semantics of Gordon et. al. [90], although the named form of expression makes the presentation shorter and clearer than for similar object calculi.

The operational semantics specifies an *evaluation relation* between initial and final configurations, $(\sigma, a) \Downarrow (\sigma', v)$. This is interpreted as stating that the evaluation of expression a with store σ results in the value v and the new store σ'.

We use $\sigma + \iota \mapsto o$ to denote updating the store σ so that ι binds to the new object o, where $\iota \in \mathsf{dom}(\sigma)$. The notation $b\{\!\{\Delta/\Gamma\}\!\}$ used in (Subst Select) denotes the bindings from the formal to the actual parameters of a method. This is a sequence of substitutions defined as follows:

Definition 4.12 (Parameter Substitution $\{\!\{\Delta/\Gamma\}\!\}$)

$$\begin{aligned} \{\!\{\emptyset/\emptyset\}\!\} &\mathrel{\widehat{=}} \varepsilon \\ \{\!\{v, \Delta/x : A, \Gamma\}\!\} &\mathrel{\widehat{=}} \{\!\{^v/_x\}\!\}\{\!\{\Delta/\Gamma\}\!\} \end{aligned}$$

where ε *is the empty substitution. Otherwise* $\{\!\{\Delta/\Gamma\}\!\}$ *is undefined.*

Note that only closed terms are evaluated, since well-formed configurations are closed. Expressions either diverge, become stuck (signified by special configuration WRONG), or result in a value which must be a location. For example, the expression **let** $x = [l = \varsigma(s : A)s.l]^p_q$ **in** $x.l$ diverges, whereas the evaluation of an expression

(Subst Value)

$$\frac{}{(\sigma,\iota) \Downarrow (\sigma,\iota)}$$

(Subst Object)

$$\frac{\sigma_1 = \iota \mapsto o, \sigma_0 \quad \iota \notin \mathsf{dom}(\sigma_0)}{(\sigma_0, o) \Downarrow (\sigma_1, \iota)}$$

(Subst Select) where $j \in 1..n$

$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]^p_q \quad \{\Delta/\Gamma_j\} \text{ is defined} \quad (\sigma_0, b_j\{\iota/s_j\}\{\Delta/\Gamma_j\}) \Downarrow (\sigma_1, \iota')}{(\sigma_0, \iota.l_j\langle\Delta\rangle) \Downarrow (\sigma_1, \iota')}$$

(Subst Update) where $j \in 1..n$

$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]^p_q \quad \sigma_1 = \sigma_0 + \iota \mapsto [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..j-1, j+1..n}, l_j = \varsigma(s : A_j, \Gamma) b]^p_q}{(\sigma_0, \iota.l_j \Leftarrow \varsigma(s : A, \Gamma) b) \Downarrow (\sigma_1, \iota)}$$

(Subst Let)

$$\frac{(\sigma_0, a) \Downarrow (\sigma_1, \iota) \quad (\sigma_1, b\{\iota/x\}) \Downarrow (\sigma_2, \iota')}{(\sigma_0, \mathbf{let}\, x : A = a\, \mathbf{in}\, b) \Downarrow (\sigma_2, \iota')}$$

Figure 4.11: Big-step, substitution-based operational semantics

which selects a non-existent method $\mathbf{let}\, x = [l = \varsigma(s : A) s.l]^p_q\, \mathbf{in}\, x.k\langle\rangle$ becomes stuck and results in the configuration WRONG.

Values require no evaluation (Subst Value). Objects evaluate to a new location which maps to the original object in the new store (Subst Object). The resulting configuration includes the new store. The evaluation rule (Subst Select) uses Definition 4.12 to construct a substitution from the actual parameters Δ into the formal parameters Γ for the selected method. This is substituted into the method body b and the resulting expression evaluated. (Subst Update) replaces the method named l from the object at location ι with the one supplied, producing a new store. (Subst Let) evaluates the first expression a, substitutes the result for x in b, and evaluates the

(Error Select1)
$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]_q^p \quad l \notin \{l_1, \ldots, l_n\}}{(\sigma_0, \iota.l\langle\Delta\rangle) \Downarrow \textsc{Wrong}}$$

(Error Select2) where $j \in 1..n$
$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]_q^p \quad \{\!\{\Delta/\Gamma_j\}\!\} \text{ is not defined}}{(\sigma_0, \iota.l_j\langle\Delta\rangle) \Downarrow \textsc{Wrong}}$$

(Error Select3) where $j \in 1..n$
$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]_q^p \quad \{\!\{\Delta/\Gamma_j\}\!\} \text{ is defined} \quad (\sigma_0, b_j\{\!\{\iota/s_j\}\!\}\{\!\{\Delta/\Gamma_j\}\!\}) \Downarrow \textsc{Wrong}}{(\sigma_0, \iota.l_j\langle\Delta\rangle) \Downarrow \textsc{Wrong}}$$

(Error Update1)
$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]_q^p \quad l \notin \{l_1, \ldots, l_n\}}{(\sigma_0, \iota.l \Leftarrow \varsigma(s : A, \Gamma) b) \Downarrow \textsc{Wrong}}$$

(Error Update2) where $j \in 1..n$
$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]_q^p \quad |\Gamma_j| \neq |\Gamma|}{(\sigma_0, \iota.l_j \Leftarrow \varsigma(s : A, \Gamma) b) \Downarrow \textsc{Wrong}}$$

(Error Let1)
$$\frac{(\sigma_0, a) \Downarrow \textsc{Wrong}}{(\sigma_0, \textbf{let}\ x : A = a\ \textbf{in}\ b) \Downarrow \textsc{Wrong}}$$

(Error Let2)
$$\frac{(\sigma_0, a) \Downarrow (\sigma_1, v) \quad (\sigma_1, b\{\!\{v/x\}\!\}) \Downarrow \textsc{Wrong}}{(\sigma_0, \textbf{let}\ x : A = a\ \textbf{in}\ b) \Downarrow \textsc{Wrong}}$$

Figure 4.12: Errors and Error Propagation

expression obtained.

The evaluation rules in Figure 4.11 apply only when certain assumptions about the term being evaluated are satisfied, such as that a method of the appropriate name

is present in the object and that the correct number of arguments are supplied. In the event that one of these assumptions is not satisfied, evaluation becomes stuck. The rules in Figure 4.12 either detect stuck computations and convert them to the configuration WRONG, or propagates the WRONG configuration to the top level of the expression being evaluated. The rules in Figure 4.12 account for the following errors:

- the message-not-understood error, when the method is not present in the object (Error Select1) and (Error Update1);
- an incorrect number of arguments supplied to a method call (Error Select2), or a method update (Error Update2); and

The remainder, (Error Select3), (Error Let1), and (Error Let2), propagate errors which occur in subexpressions to become the resulting configuration of evaluating the top level expression.

The final piece of the puzzle is the initial configuration which is used to evaluate a program. If a is a program, then evaluation of a begins with the configuration $(\emptyset, a)$.

4.5 Key Properties

We have proven the soundness of the type system. The proofs have been omitted as the current calculus is subsumed by the one presented in the following chapter.

The key to proving soundness is the following lemma. It states that a type contains sufficient information to determine the permission required for values of that type, and that this information is preserved through subtyping.

Lemma 4.13 (Permissibility)

1. *If* $E; K \vdash v : A$ *and* $K' \vdash A$, *then* $E; K' \vdash v : A$, *where v is a value*.
2. *If* $K \vdash A <: B$ *and* $K' \vdash B$, *then* $K' \vdash A <: B$.

The first clause is essential for demonstrating type preservation for method selection and update, as both operations require values to pass between an object and an expression where in each case access is governed by different permissions. This clause is valid only for values, not expressions, because the type of an expression does not include all the information concerning which contexts it may access, only those of

the eventual result. A simple example is the expression **let** $x = [\ldots]^{\mathsf{Bob}}_{\mathsf{Bob}}$ **in** $[]^{\mathsf{World}}_{\mathsf{World}}$ which has type $[]^{\mathsf{World}}_{\mathsf{World}}$, but requires permission to access context Bob.

The second clause of the lemma is required to establish the validity of substitution and subsumption.

Definition 4.14 (Extension) *Environment E' is an extension of E, written $E' \gg E$, if and only E is a subsequence of E'.*

The soundness of the type system is the property that evaluation of well-typed configurations either diverges or produces a value of the expected type. It is given by the following two lemmas.

Lemma 4.15 (Type Preservation) *If $E;K \vdash (\sigma,a) : A$ and $(\sigma,a) \Downarrow (\sigma',v)$, then there exists an environment E' such that $E' \gg E$ and $E';K \vdash (\sigma',v) : A$.*

Lemma 4.16 (Faulty Programs are Untypable) *If $(\sigma,a) \Downarrow$ WRONG, then there are no E, K, or A, such that $E;K \vdash (\sigma,a) : A$.*

These are proved by induction over the evaluation relation $(\sigma,a) \Downarrow (\sigma',v)$, following the now-standard approach of Wright and Felleisen [198].

Theorem 4.17 (Soundness) *If $\emptyset;K \vdash (\emptyset,a) : A$, then either $(\emptyset,a)$ diverges, or $(\emptyset,a) \Downarrow (\sigma,v)$ and there exists an environment E such that $E;K \vdash (\sigma,v) : A$.*

The containment invariant does not immediately follow from soundness of the type system. We now fill that gap.

4.6 The Containment Invariant

The containment invariant is a statement about the well-formedness of stores. Recall that it says:

$$\iota \rightarrow \iota' \;\Rightarrow\; \mathsf{rep}(\iota) \prec\!: \mathsf{owner}(\iota'),$$

where $\mathsf{rep}(\iota)$ is the representation context of the object in location ι and $\mathsf{owner}(\iota')$ is the owner context for the object in location ι'. The *refers to* relation, $\rightarrow$, captures when one object refers to another and will be made precise below.

The containment invariant is a restriction on the reference structure between objects. We prove that it holds for well-formed stores, firstly by demonstrating an invariant relating permissions and the owners of locations in expressions, and then by applying this to the method bodies of objects in the store.

A location can appear in an expression only when its owner context is a part of the permission used to type that expression. Thus the collection of owner contexts appearing in a term should therefore be a subset of the contexts underlying a permission. To show this we define functions for projecting a permission onto the contexts it denotes and an expression onto the owners of locations appearing in the term. The signatures of these functions are:

- $\eta : \textsc{Location} \to C$.
- $[\![\_]\!] : \textsc{Permission} \to \mathbb{P}(C)$.
- $[\![\_]\!]_\eta : \textsc{Expression} \to \mathbb{P}(C)$.

The final function is parameterised by the function η which gives the owners of locations in an expressions. η is defined as follows:

Definition 4.18 *Define* $\eta \models E$ *to hold whenever* $\eta(\iota) = p$ *for all* $\iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q$ *in* E. *This serves to define* η *for the typing environment* E.

Permissions are modelled as sets of contexts as we have already anticipated:

Definition 4.19 (The Model of Permissions)

$$\begin{array}{rcl} [\![\langle p \rangle]\!] & \widehat{=} & \{p\} \\ [\![\langle p\uparrow \rangle]\!] & \widehat{=} & \{q \in C \mid p \prec\!: q\} \end{array}$$

The projection of expressions depends upon the locations present in the expression. The following function collects together the locations which are not wrapped in the body of another object's method. Because a different permission governs their presence, the locations wrapped inside another object are excluded.

Definition 4.20 (Locations in a Expression) *The locations in an expression,* $\mathsf{locs}(a)$, *is defined as follows:*

$$\mathsf{locs}(\iota) \;\widehat{=}\; \{\iota\}$$

$$
\begin{array}{rcl}
\mathit{locs}(x) & \widehat{=} & \emptyset \\
\mathit{locs}([l_i = \varsigma(s_i : A_i, \Gamma_i) b_i^{\,i \in 1..n}]_q^p) & \widehat{=} & \emptyset \\
\mathit{locs}(a.l\langle\Delta\rangle) & \widehat{=} & \mathit{locs}(a) \cup \mathit{locs}(\Delta) \\
\mathit{locs}(\emptyset) & \widehat{=} & \emptyset \\
\mathit{locs}(v, \Delta) & \widehat{=} & \mathit{locs}(v) \cup \mathit{locs}(\Delta) \\
\mathit{locs}(a.l \Leftarrow \varsigma(s : A, \Gamma) b) & \widehat{=} & \mathit{locs}(a) \cup \mathit{locs}(b) \\
\mathit{locs}(\textbf{\textit{let}}\ x : A = a\ \textbf{\textit{in}}\ b) & \widehat{=} & \mathit{locs}(a) \cup \mathit{locs}(b)
\end{array}
$$

Note that the locations in the new method body for a method update originate from outside of the object which is being updated.

The owner contexts of the locations appearing in an expression is defined as follows in terms of a function which gives the owner context of locations.

Definition 4.21 (The Projection of Expressions)

$$[\![a]\!]_\eta \ \widehat{=}\ \{\eta(\iota) \mid \iota \in \mathit{locs}(a)\}$$

The following theorem (clause 2 in particular) presents an invariant which states that the owners of locations appearing in a well-typed expression is bounded by the contexts underlying the permission used to type that expression.

Theorem 4.22 *Assume, where relevant, that* $\eta \models E$. *Then,*

1. *If* $K' \subseteq K$, *then* $[\![K']\!] \subseteq [\![K]\!]$;
2. *If* $E; K \vdash a : A$, *then* $[\![a]\!]_\eta \subseteq [\![K]\!]$; *and*
3. *If* $E; K \vdash (\Theta)(\Delta) \Rightarrow C$, *then* $[\![\Delta]\!]_\eta \subseteq [\![K]\!]$.

This is proven by mutual induction on the structure of typing derivations.

We now define the notion of a *well-contained store* which captures the containment invariant globally for all objects.

Definition 4.23 (Well-contained Store)

$$
\begin{array}{rcl}
\mathit{wf}_\eta(\sigma) & \widehat{=} & \forall \iota \mapsto o \in \sigma.\mathit{wf}_\eta(o) \\
\mathit{wf}_\eta([l_i = \varsigma(s_i : A_i, \Gamma_i) b_i^{\,i \in 1..n}]_q^p) & \widehat{=} & \forall i \in 1..n.[\![b_i]\!]_\eta \subseteq [\![\langle q\uparrow\rangle]\!]
\end{array}
$$

Now well-typed stores are well-contained:

Lemma 4.24 *If* $E \vdash \sigma$ *then* $\boldsymbol{wf}_\eta(\sigma)$, *where* $\eta \models E$.

We now convert this result to a local definition, that is, one defined between pair of locations, thus demonstrating that the containment invariant holds. Firstly define the *refers to* relation as follows.

Definition 4.25 (*refers to*) *The* refers to *relation,* $\rightarrow_\sigma$, *for store* σ *is defined as:*

$$\iota \rightarrow_\sigma \iota' \text{ iff } \iota \mapsto [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i^{\,i \in 1..n}]_q^p \in \sigma \wedge \iota' \in \mathsf{locs}(b_i), \text{ for some } i \in 1..n$$

The functions giving the owner and representation contexts are defined for each binding in the store. For $\iota \mapsto [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i^{\,i \in 1..n}]_q^p \in \sigma$, define $\mathsf{owner}_\sigma(\iota) \mathrel{\widehat{=}} p$ and $\mathsf{rep}_\sigma(\iota) \mathrel{\widehat{=}} q$. Note that when $E \vdash \sigma$ and $\eta \models E$ we have $\eta \equiv \mathsf{owner}_\sigma$.

The containment invariant can now be stated precisely.

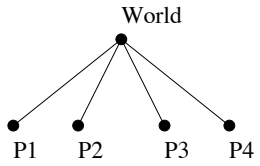
Theorem 4.26 (Containment Invariant) *If* $E \vdash \sigma$ *then* $\iota \rightarrow_\sigma \iota' \Rightarrow \mathsf{rep}_\sigma(\iota) \prec: \mathsf{owner}_\sigma(\iota')$.

PROOF: Straightforward. See the proof of Theorem 5.24 in Chapter 5 for an almost identical account. □

4.7 Examples

The type system explored in this chapter supports containment where the units of ownership are prespecified for a given system. This means that objects can be partitioned and contained within contexts which are defined per package or per class or come from some other prespecified collection. We demonstrate a few example uses of such collections, observing that different partial orders allow for different kinds of restriction, including a partition of objects without any containment.

Example 4.27 (Confined Types [24]) *In Bokowski and Vitek's Confined Types proposal the contexts correspond to the packages of Java including the package without a name, which we will call* World. *Each context is nested inside* World, *but there is no other ordering between contexts. In general, the ordering on contexts is depicted as:*



Classes can be annotated with the `confined` *keyword to indicate that instances of that class are contained within the package in which the class is defined. This means that instances of a confined class are not accessible by classes defined outside the package in which the class is defined.*

We model the type of an object from a confined class from package P *using a type such as* $[l_i : \Theta_i{}^{i\in 1..n}]^{\mathsf{P}}_{\mathsf{P}}$*. This means that it can access other objects contained in package* P*, but cannot be accessed by objects not in defined in* P*. An object from a non-confined class defined in package* P *has type* $[l_i : \Theta_i{}^{i\in 1..n}]^{\mathsf{World}}_{\mathsf{P}}$*, indicating that it is accessible to all objects, but can also access objects contained within package* P*.*

The following example can be coded using Confined Types [24].

Example 4.28 (Web Browsers and Applets) *Assume we have two packages,* Applet*, abbreviated as A, and* Browser*, abbreviated as B. Let* securityManager *be the object which controls a web browser's security policy. This must be confined to the* Browser *package to keep it away from untrusted applets. The* Applet *package contains the implementation of* applets*. These interact with a* browser *object. The* applets *have limited access to the outside world; this is governed by the* securityManager*. The* browser *object has access to the* securityManager *object to guide its interaction with the* applets*.*

Given the context ordering $A, B \prec: \mathsf{World}$*, we can represent this situation above with the following objects:*

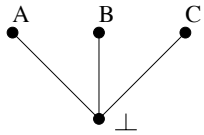
$$\begin{aligned} \text{securityManager} &\mathrel{\widehat{=}} [\ldots\ldots]^{B}_{B} \\ \text{browser} &\mathrel{\widehat{=}} [\text{securityManager} = \text{securityManager}, \text{applet} = \text{applet}, \ldots]^{\mathsf{World}}_{B} \\ \text{applet} &\mathrel{\widehat{=}} [\text{browser} = \text{browser}, \ldots]^{\mathsf{World}}_{A} \end{aligned}$$

This example can be extended so that each applet *object has its own protected* environment *object, using a term which dynamically creates contexts, to be described in Chapter 5.*

We now describe the essence of one part of the Universes [144] proposal.

Example 4.29 (Class Names as a Partition) *The Universes system can use class names as object owners [144]. These can be used to partition the collection of objects, using one partition per class, without providing any containment. All contexts are equally accessible, but objects owned by one class cannot be assigned to a field expecting an object with a different class as owner.*

The contexts required to model this consist of a single context for each class plus the additional context which we denote $\bot$. The partial order on contexts is $\bot \prec: C$, for all classes C. An example partial order is:

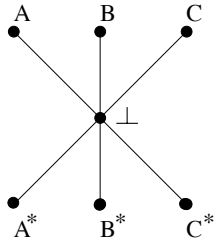


This tree, which is an upside down version of the context ordering used in our original ownership types [61], could have a top element to denote the owner of globally accessible objects.

An object in partition C is given owner C and representation context $\bot$. Such an object can access objects from any other partition and can be accessed by objects from any other partition, but can only be assigned to fields and variables which expect an object owned by partition C.

We can enhance the previous example to allow each class to have instances which are not accessible to other classes. This amounts to extending the previous example with some form of containment.

Example 4.30 (Adding Containment) *Extend the partial order from the previous example with $C^* \prec: \bot$ for each class C. The context ordering becomes:*

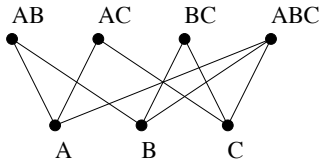


Give objects which are not accessible outside of class C both owner and representation context C^*. Objects from class C in partition B now have owner B and representation context C^*, not $\bot$ as above, so that it can access the elements contained within class C.

We can use ownership types to capture more general constraints on systems. For example, we can enforce that objects defined within one part of the system (a subsystem) are not accessible outside of that subsystem, or that such objects are accessible to a particular, prespecified collection of other subsystems.

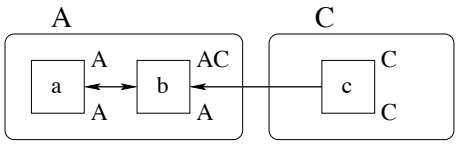
Example 4.31 (General Systems) *A system can be partitioned into a collection of subsystems (perhaps using packages or modules). Some objects are not accessible outside a given subsystem, whereas others may belong to a given subsystem, but be accessible by one or more other subsystems. We model this as follows.*

Let the set $P = \{A, B, ...\}$ denote the names of the subsystems. Let the collection of contexts be $C = \mathbb{P}(P) - \{\emptyset\}$, where $\mathbb{P}(P)$ is the powerset of P. We will write ABDF to represent the set $\{A, B, D, F\}$. The context ordering $\prec:$ is the defined as follows: for each $A \in P$, if $A \in P$, where $P \in \mathbb{P}(P)$, then $A \prec: P$, abusing notation slightly. For example, $A \prec: ABC$. Note that this means that the context ordering is a dag, as demonstrated in the following diagram for subsystems A, B, and C:



We set an object's representation context to be the subsystem in which it resides — that is, A, B, or C — and its owner context to the collection of subsystems which can access it — any context in the diagram above.

Consider the following collection of subsystems, A and C, and objects a, b, and c. The arrows represent the only references allowed between these objects.



Both a and b come from subsystem A, so their representation context is A. a is contained within subsystem A, so its owner is A. b is also accessible to subsystem C, so its owner is AC. Finally c has both owner and representation context C, and is accessible only within C. The containment invariant enforces that the references in the figure are the only ones allowed given these contexts and their ordering.

Adding more structure to the partial-order, for example, by letting the ordering on contexts be the powerset ordering on sets of subsystem names, allows us to specify in addition the sharing of contained objects among subsystems.

This granularity of the restrictions in the above example resembles Eiffel's export policies, except that in the example the constraints are defined between objects, rather than for individual methods [138].

Our final example is a model of the extended privacy constructs described in Section 3.6.

Example 4.32 *Recall the code from Chapter 3.*

```
class A is public {
    class B is private-up-to A { }
    class C is private-up-to A {
        class D is private-up-to C { }
        class E is private-up-to A { }
        class F is public { }
    }
}
```

Using the contexts $C = \{\varepsilon, A, B, C, D, E, F\}$*, with ordering* $B, C \prec: A \prec: \varepsilon$ *and* $D, E, F \prec: C$*, the following term models the above nest of classes:*

$$
\begin{array}{l}
[\texttt{newA} = \varsigma() \\
\quad [\textit{... fields and methods of class A ...}, \\
\quad \texttt{newB} = \varsigma()[\textit{... fields and methods of class B ...}]^{A}_{B}, \\
\quad \texttt{newC} = \varsigma() \\
\qquad [\textit{... fields and methods of class C ...}, \\
\qquad \texttt{newD} = \varsigma()[\textit{... fields and methods of class D ...}]^{C}_{D}, \\
\qquad \texttt{newE} = \varsigma()[\textit{... fields and methods of class E ...}]^{A}_{E}, \\
\qquad \texttt{newF} = \varsigma()[\textit{... fields and methods of class F ...}]^{\varepsilon}_{F}, \\
\qquad]^{A}_{C}, \\
\quad]^{\varepsilon}_{A} \\
\quad \textit{... remainder of program ...} \\
]^{\varepsilon}_{\varepsilon}
\end{array}
$$

The functions `newN` *are considered to be constructor functions. It is instructive to consider all objects which result from a particular constructor call (e.g,* `newE`*) as being an instance of that class (class* `E`*), even though they may be nested within different objects.*

The objects resulting from this code all respect the visibility restrictions denoted by the annotations on the code, or as depicted in Figure 3.5 from Chapter 3. For example, we have that `D` *objects are only visible to objects from classes* `C`*,* `E`*, and* `F`*; that* `F` *objects are visible to any object, and that every is visible to an* `F` *object; and the*

`B` objects are not visible at the top-level. The type system ensure that these visibility conditions hold even in the presence of subtyping.

Although the containment boundaries are specified at a class-level, the restrictions apply to individual objects. In Java this would be impossible since privacy annotations apply only to class names, but not necessarily to the names of its super classes [93]. However, the type system presented in this chapter cannot prevent an object restricted to class `C`, for example, from being passed to other objects from class `C`. The type system presented in Chapter 5 can do this.

4.8 Concluding Remarks

The calculus presented here is simple, but of interest in its own right, as the examples testify. It was developed as a prelude to the more sophisticated calculus which appears in the next chapter. Indeed it was not until we worked through the complete details of the present calculus did we gain enough understanding to develop the calculus which follows in the next chapter.

One limitation of the present calculus, and indeed the one which follows, is that the containment invariant applies only to the store. It says little about how a particular store was created. From a modelling perspective this is unsatisfactory, since we may wish to restrict which objects create other objects, depending on which context they reside in. We have more to say about this in Chapter 6 (Section 6.2.1).

In the next chapter we extend the present calculus with a mechanism that allows contexts to be created during evaluation. This allows, for example, every object to have its own unique representation context. Further extensions are made both to improve expressiveness of the calculus and to demonstrate how standard type-theoretic constructs, such as recursive types and universal quantification, can be transferred to the current setting.

Chapter 5

Infinitary Ownership Types

From an object-oriented modelling perspective, having only a fixed number of contexts as object owners is an obvious limitation, since each object should be able to own its own representation, rather than sharing it with all elements of the same class or package [174, 150]. Since the number of objects is potentially unbounded, the number of contexts also must be unbounded. To this end we extend the calculus with an operation which creates contexts during evaluation, so that each object may have its own representation context which is distinct from every other object's representation context. Consequently each object can have its own representation.

Context creation, however, raises a technical problem. Since contexts appear in object types, types now depend upon values and we risk the inherent undecidability problems associated with dependent types [107]. We avoid these problems using existential quantification over contexts, but unconstrained existential quantification over contexts introduces its own problems. In particular, when the owner of an object is hidden, the containment invariant can be violated, since the the owner determines which objects can access an object.

We avoid these problems. Indeed, one of the contributions of this chapter is to add dynamic context creation to the calculus in such a way that it avoids the checkability problems associated with dependent types and retains the containment invariant in the presence of existential quantification of contexts.

In addition to this important extension, we add other features to enhance the expressiveness of the underlying object calculus substrate. These include type variables, recursive types, top type(s), and universal quantification over types (in methods only). We add these features to obtain a calculus expressive enough to model the features of a class-based programming language with generics, such as GJ [34], with parametric

ownership, as in our original ownership types system [61]. To preserve the containment invariant these features require a more careful treatment in the type rules than usual, which we elegantly achieve using permissions.

In this chapter we roughly parallel the structure of the previous chapter, moving a little faster while presenting more detail. Section 5.1 presents an overview of the additional machinery required for this calculus. The extended calculus with the additions is presented in Section 5.2. The calculus is strictly an extension of that of Chapter 4 in the sense that all well-typed terms from the previous calculus are given the same type (and semantics) in this calculus. In Section 5.3 we present its type system and in Section 5.4 discuss the design choices made regarding Top type and existential quantification. Section 5.5 contains the calculus' dynamic semantics. Section 5.6 presents the key properties of the calculus, culminating in soundness. Showing that the containment invariant holds requires an extension of the techniques employed in the previous chapter. This is done in Section 5.7. Finally, Section 5.8 concludes. Important proofs are outlined along the way, with their complete details appearing in Appendix A.

Since this calculus is the most sophisticated one, we reserve major examples until the next chapter. We now begin by describing the additional machinery we add to the calculus.

5.1 New Contexts and More Types

We add a number of features to the calculus for the following reasons:

- context variables — to allow new context creation and context parameterisation
- new context creation — to allow each object to have its own representation context
- existential quantification over contexts — to avoid the dependent typing problems that new context creation might cause
- top types — to support the presentation of recursive types and universal quantification
- type variables — to allow recursive types and universal type quantification
- recursive types — to model self typing. We add these because a change is required to the usual typing rule.

- context and type parameterised methods — to model classes which are generic in both their owner and in the usual sense.

These features are now described in some detail.

Context Variables and Context Creation Dynamic context creation requires a term which creates new contexts as well as context variables so that the new contexts can be used in expressions. The new contexts will generally be used as the representation contexts of new objects, though the calculus does not enforce this. Since the partial order on contexts governs object access, new contexts must be incorporated into this ordering. Hence the ownership tree/partial order grows during evaluation.

Evaluation begins with a fixed set of constant contexts, denoted C, as before. The nesting between these contexts is now denoted $\prec{:}_C$. We can, but need not, assume that C has a distinguished maximal element, ε. For simplicity we assume that this partial order is a downwards-growing tree, when ε is present; or, in the absence of ε, a forest. In the simplest case $C = \{\varepsilon\}$, in which case ε corresponds to the keyword `norep` (no one's representation) from our original ownership types proposal [61].

The term for creating new contexts is **new** $\alpha \lhd p$ **in** a. This means create a new context directly inside context p and substitute the new context for α in a. The context p can be any context including those previously created by a **new**. The partial order on contexts is also extended to record that the new context is inside p. This operation resembles terms which create new names in other calculi [140, 152, 160, 83], except that we use them to create types, like [188, 201, 48, 49, 80], though we are the first to apply this notion in an object-oriented setting and with an extensive amount of subtyping. Since we attach properties to our names/contexts and use existential quantification to hide them, our overall technique most closely resembles Flanagan and Abadi's lock types [80], though our application is very different.

Existential Quantification of Contexts The bounded existentially quantified type $\exists(X{<:}A)B$ can be used to model a partially abstract data type which has *interface* B and *representation type* X known only to be a subtype of A [3]. This means that the actual type of the value is hidden, and that the value can only be accessed using the interface A. This construct, or more precisely, its underlying introduction and elimination terms, seems appropriate for protecting the representation of objects, but it can only protect the representation type of an object, but not the objects of that type, since subsumption allows those object to be exported using interface type A.

To use existential types for protecting representation, we must find a workaround. The key is to guarantee that subtyping does not forget when an object is representation, or, rather, that subtyping cannot be used to forget which objects are not allowed to access an object. We do this primarily by abstracting the representation context of an object, yielding a type such as $\exists(\alpha \prec: p)[f : [\ldots]^{\alpha}_{\_}]^{p}_{\alpha}$. Here the field f contains representation. The type system must guarantee that the contents of this field cannot be exported. It does so by ensuring that there is no supertype of $[\ldots]^{\alpha}_{\_}$ which forgets that permission $\langle\alpha\rangle$ is required to access it. Since α is unknown, it cannot be fabricated, thus objects of type $[\ldots]^{\alpha}_{\_}$ cannot be directly exported.

The type $\exists(\alpha \prec: p)A$ states that context α is hidden within interface type A. All that is known about the hidden context is that it is inside p. Associated with this type are introduction and elimination terms:

- **hide** p **as** $\alpha \prec: q$ **in** v:A – hides the context p of v as α, revealing only its bound q. The type A provides enough information to determine the type of this construct: $\exists(\alpha \prec: q)A$

- **expose** v **as** $\alpha \prec: p, x$:A **in** b:B – this term reveals the hidden context and value to expression b, binding them to α and x, respectively. The result type B cannot depend on α, ensuring that the hidden representation context is not exposed.

These terms are more commonly called *pack* and *open* [3], but we have chosen alternative names that are more indicative of their intended behaviour in our modelling. The **hide** term provides a protective wrapper to hide representation from external access, and **expose** removes the protective to allow access to the hidden object's methods, albeit within a limited scope.

Example 5.1 *The following is an example object where context hiding is combined with context creation to give an object a new context.*

new $\alpha \lhd p$ ***in***	create new context inside p
let $x = [\ldots]^{p}_{\alpha}$ ***in***	set as the representation context of new object
hide α ***as*** $\beta \prec: p$ ***in*** x:$[\ldots]^{p}_{\beta}$	hide context, producing a protected object
$: \exists(\beta \prec: p)[\ldots]^{p}_{\beta}$	resulting type

The syntax of the calculus can be restricted to guarantee that each object has its own unique representation context (see Section 6.1), but the present syntax makes no such guarantees.

We do not force that the context hidden be a representation context, so long as the hidden context is not the outer level owner, as in the type $\exists(\alpha \prec: p)[\ldots]^{\alpha}_{\_}$. As mentioned at the start of this chapter, the owner is used to determine which objects can access which other objects; hiding the results in a violation of the containment invariant. Our type system outlaws such types, but we leave the details until Section 5.4.

Finally, we could easily have added existential quantification over types, but doing so would unnecessarily complicate the calculus at hand.

Top Types Type systems with subtyping that also include recursive types and/or universal quantification over types often include a maximal type called Top for convenience and effect. Top type allows a simpler treatment of recursive types and subtyping [12, 85]. With Top unbounded universal quantification can be modelled using bounded universal quantification [3].

Since any value can have Top type, this type would unify collections of values which ownership types insist on keeping distinct. Adding Top type directly to our type system makes the containment invariant invalid, because the permission information governing who has access to the value is lost. One of our contributions is a remedy to this problem. We introduce a family of Top types indexed by permissions. The type Top^K denotes the collection of values accessible given permission K, and thus access to values of type Top^K requires at least permission K. This is sufficient to maintain the containment invariant in the presence of recursive and universally quantified types. More discussion is given in Section 5.4.

Type Variables. Recursive Types Type variables are required for recursive types, which are essential for providing the type of self, and for universally quantified types, which model genericity in object-oriented languages [3]. Indeed, much of Abadi and Cardelli's book is concerned with providing accurate type-theoretic models of self typing, but we need not travel so far here. We include recursive types to demonstrate the modifications to the usual type rules which are required to support them.

Type variables X are added to the language of types. They are declared in typing environments with a bound, $X<:A$. Recursive types, denoted $\mu(X)A$, are standard iso-recursive types [65]. They represent the solution to the equation $X = A$, where X can appear free in A. This establishes an isomorphism between $\mu(X)A$ and $A\{{}^{\mu(X)A}/_{X}\}$. Rather than consider these types equal, we use the terms **fold** and **unfold** to mediate between these two types [3].

For example, let $A \mathrel{\widehat{=}} \mu(X)[l_1 : \mathit{Int}, l_2 : X]$ be the type of an object whose first method returns an integer and whose second method returns something of type X, which is the same type as self. Now let v be a value with type A. Selecting the method l_2 first requires an unfolding of v and then the selection of method l_2. Now **unfold**(v) has type $[l_1 : \mathit{Int}, l_2 : X]\{{}^{A}/_{X}\} \equiv [l_1 : \mathit{Int}, l_2 : A] \equiv [l_1 : \mathit{Int}, l_2 : \mu(X)[l_1 : \mathit{Int}, l_2 : X]]$, and thus **unfold**$(v).l_2$ has type $\mu(X)[l_1 : \mathit{Int}, l_2 : X] \equiv A$.

Ensuring that the recursive type $\mu(X)A$ is well-formed requires that both isomorphic forms are well-formed. The trick is to ensure that the permission required to access the type variable X is sufficient to access A, thus providing enough permission when A is substituted for X. This is done by requiring that the bound Top^K on X is such that K is sufficient permission to access A.

Apart from these considerations, subtyping recursive types follows the standard pattern [47, 12, 85].

Context and Type Parameterised Methods To increase the flexibility of the calculus we allow methods to have both context and type parameters, with the appropriate bounds. The flexibility type parameters allow is well understood — e.g., for generic classes [3, 54, 115] — so we will focus only on context parameters here.

Context parameters allow owner polymorphism, an example of which is a method which can be applied to objects with different owners. A method can take a context parameter and use it as the owner of a freshly created object. This is fundamental for modelling classes, since a class should in general be able to create objects with different owners, otherwise classes would be very restricted. Context variables are constrained either to be inside ($\prec:$) or outside ($:\succ$) of another context. To illustrate, consider the following object, which has a method that takes two context parameters α and β as arguments:

$$[l = \varsigma(s : A, \alpha \prec: \varepsilon, \beta :\succ \alpha)b]$$

The context variable α is bounded above by ε, assuming that a maximal element exists. This means that *any* context can be given as the first argument to this method, and thus this is used to implement unbounded context parameterisation. The method l is a feature available to any object. If the upper bound had been something other than ε,such as $\alpha \prec: p$, then access to the feature is restricted to anything inside p, excluding those not.

The second context variable is bounded below by α. A consequence of this is that an object with owner β is accessible to an object with owner α. If α is the owner of

a container, then β could be the owner of the data held by the container, for example. This combination of context parameters forms a useful idiom, for example, to provide a generic print method which works irrespective of the owner of a container and of the data which it contains.

Contexts variables can be combined with type parameterisation to obtain even more flexibility.

More complex types could be added to the object calculus substrate to match the calculi in Abadi and Cardelli's book, but we believe that the system presented here captures the necessary modifications to the usual type rules so that adding more complex constructs would present no additional challenges with respects to the ownership veneer.

5.2 A Calculus with Ownership

The calculus in this section extends that of the previous chapter with the features just described. Objects remain essentially the same, though now methods take parameters of different kinds. The extended syntax is given in Figure 5.1. Here we focus mainly on the new features.

Contexts Contexts consist of constant contexts from the partial order $(C, \prec\!:_C)$ and variables from the denumerable collection CONTEXTVAR. Their nesting is given by the relation $\prec\!:$.

Permissions The two basic permissions, $\langle p \rangle$ and $\langle p\uparrow \rangle$, remain, but now p can be a context variable. Finite unions of permissions can also be formed using $\bigcup[K_1..K_n]$. We use the following abbreviations: $\mathsf{void} \mathrel{\widehat{=}} \bigcup[\,]$ and $K \cup K' \mathrel{\widehat{=}} \bigcup[K, K']$. Finite unions of permissions are used when a method takes context parameters as arguments. The permission governing the method body is extended to include the point permissions for the method's context parameters, for the duration of the method only.

Types Types include type variables for both recursive types and type parameterised methods, a top type for each permission, object types, recursive types, and a limited form of existential type. The types (except perhaps Top) differ little from the standard

α	$\in$	CONTEXTVAR			
p	$\in$	CONTEXT	$::=$	α	
			$\mid$	π	$\pi \in C$
K	$\in$	PERMISSION	$::=$	$\langle p \rangle$	
			$\mid$	$\langle p\uparrow \rangle$	
			$\mid$	$\bigcup[K_1..K_n]$	
X	$\in$	TYPEVAR			
A,B,C	$\in$	TYPE	$::=$	X	
			$\mid$	Top^K	
			$\mid$	$[l_i : \Theta_i{}^{i\in 1..n}]_q^p$	
			$\mid$	$\mu(X)A$	
			$\mid$	$\exists(\alpha \prec: p)A$	
Θ	$\in$	METHODTYPE	$::=$	$A \mid A \to \Theta \mid \forall(X<:A)\Theta$	
			$\mid$	$\forall(\alpha \prec: p)\Theta \mid \forall(\alpha :\succ p)\Theta$	
ι	$\in$	LOCATION			
x	$\in$	VAR			
u,v	$\in$	VALUE	$::=$	x	
			$\mid$	ι	
			$\mid$	$\mathbf{fold}(A,v)$	
			$\mid$	$\mathbf{hide}\ p\ \mathbf{as}\ \alpha \prec: q\ \mathbf{in}\ v{:}A$	
a,b	$\in$	EXPRESSION	$::=$	v	
			$\mid$	o	
			$\mid$	$v.l\langle\Delta\rangle$ *where* $\Delta ::= \emptyset \mid v,\Delta \mid A,\Delta \mid p,\Delta$	
			$\mid$	$v.l \Leftarrow \varsigma(s : A,\Gamma)b$	
			$\mid$	$\mathbf{let}\ x : A = a\ \mathbf{in}\ b$	
			$\mid$	$\mathbf{unfold}(v)$	
			$\mid$	$\mathbf{expose}\ v\ \mathbf{as}\ \alpha \prec: p, x{:}A\ \mathbf{in}\ b{:}B$	
			$\mid$	$\mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ a$	
o	$\in$	OBJECT	$::=$	$[l_i = \varsigma(s_i : A_i,\Gamma_i)b_i{}^{i\in 1..n}]_q^p$	*(l_i distinct)*
Γ	$\in$	PARAM	$::=$	$\emptyset \mid x : A,\Gamma \mid X<:A,\Gamma \mid \alpha \prec: p,\Gamma \mid \alpha :\succ p,\Gamma$	
σ	$\in$	STORE	$::=$	$\emptyset$	
			$\mid$	$\sigma, \iota \mapsto o$	
s,t	$\in$	CONFIG	$::=$	(Π,σ,a) *where* $\Pi ::= \emptyset \mid \alpha \prec: p,\Pi$	
			$\mid$	WRONG	

Figure 5.1: The Syntax

type theoretic constructs which they resemble. Method types are described in the next section.

The top type Top^K is the largest type of all types that can be constructed using permission K. Informally it represents the union of all values accessible given permission K.

Object types remain the same as those of the previous chapter, except now method types are more complex.

Recursive types $\mu(X)A$ are standard iso-recursive types, and $\exists(\alpha \prec: p)A$ represents a limited form of existentially quantified type which abstracts only context, as described in the introductory sections.

Method Types We extend methods to take both context and type parameters. Context parameters can be bounded above, giving method type $\forall(\alpha \prec: p)\Theta$, or below, giving $\forall(\alpha :\succ p)\Theta$. Type parameters are bounded as usual $\forall(X<:A)\Theta)$. The types and contexts of parameters appearing later in the formal parameter list can depend on those specified earlier.

Terms Terms are divided into values and expressions. Values are the results of computation.

For objects, $[l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i\in 1..n}]_q^p$, the formal parameters Γ_i are now a collection of term variables with their type, context variables with their bound, and type variables with their bound.

The actual parameters, Δ, passed to a method selection, $v.l\langle\Delta\rangle$, are a sequence of values, contexts, and types. All the method arguments must be supplied.

Method update, $v.l \Leftarrow \varsigma(s : A, \Gamma)b$, also uses the additional kinds of formal parameters.

Local declarations are defined using **let** $x : A = a$ **in** b.

The terms $\mathbf{fold}(A, v)$ and $\mathbf{unfold}(v)$ are coercions which mediate between the forms of iso-recursive type.

The hide expression, **hide** p **as** $\alpha \prec: q$ **in** v:A, abstracts context p from the term v (and type A), and represents it as bound variable α. The only information known about p is that it is inside q. The type of this expression is $\exists(\alpha \prec: q)A$. This corresponds to the usual term for packing existential types, except contexts are abstracted rather than types. When v is a location (or a fold of a location), then such a value can represent an object with protected representation.

The expose expression, **expose** v **as** $\alpha \prec: p, x{:}A$ **in** $b{:}B$, is used to unpack the contents of a hide expression, that is, the hidden context p and the value v. These are substituted for α and x respectively in the expression b. The type system ensures that the exposed context cannot escape the scope of the expression b.

The final term **new** $\alpha \lhd p$ **in** a is the most significant addition to the calculus. It creates a new context which is inside but not equal to p, and substitutes it for α in a. Generally this context will be used as the representation context of a new object, thus every object can have its own unique representation context, as discussed in the introductory section.

We are generally interested in closed expressions which can then be evaluated in an empty configuration. We refer to closed expressions as *programs*.

Configurations A configuration represents a snapshot of the evaluation. It consists of an expression, a, a store, σ, and a collection of constraints, Π, on the context variables free in the expression and store. The collection Π is a new addition required to track the constraints on the context variables created during evaluation. The constraints capture the nesting between contexts.

The WRONG configuration is the result of faulty programs. The type system aims at ensuring that faulty programs are ill-typed.

5.3 Type Rules

The type of an expression depends on a *typing environment*, E, which maps program variables and locations to types, and records subtyping assumptions for type variables and the bounds on context variables. The typing environment is organised as a sequence of bindings and constraints, where $\emptyset$ denotes the empty environment:

$$E \quad ::= \quad \emptyset \mid E, x : A \mid E, \iota : A \mid E, X <: A \mid E, \alpha \prec: p \mid E, \alpha :\succ p$$

Note that syntax of method formal parameters, Γ, and of constraint sets, Π, are included within the syntax for environments. This allows them to be used wherever environments are in the formal system.

We define the type system using the judgements described in Figure 5.2. All judgements concerned with types and expressions are formulated with respect to a permission K. This means that expressions are well-typed, that types are well-formed, and that a subtype relation holds only when sufficient permission is given.

$E \vdash \diamond$	*E is a well-formed typing environment*
$E \vdash p$	*p is a well-formed context in E*
$E \vdash p \prec: q$	*p is a context inside q in E*
$E \vdash K$	*K is a well-formed permission in E*
$E \vdash K \subseteq K'$	*K is a subpermission of K' in E*
$E;K \vdash A$	*A is a well-formed type in E given K*
$E;K \vdash A<:B$	*A is a subtype of B in E given K*
$E;K \vdash a : A$	*a is a well-typed expression of type A in E given K*
$E;K \vdash \Theta$ *meth*	*Θ is a well-formed method type in E given K.*
$E;K \vdash (\Theta)(\Delta) \Rightarrow C$	*The actual parameters Δ match method type Θ whose return type is C in E given K.*
$E \vdash \sigma$	*σ is a well-formed store in E*
$E;K \vdash (\Pi,\sigma,a) : A$	*(Π,σ,a) is a well-typed configuration with type A in E given K*

Figure 5.2: Judgements

Before proceeding, we update our definitions of the domain of a typing environment, $\mathsf{dom}(E)$, and of free variables, $FV(\,)$.

Definition 5.2 ($\mathsf{dom}(E)$)

$$\begin{aligned}
dom(\emptyset) &\mathrel{\widehat{=}} \emptyset \\
dom(E,\iota : A) &\mathrel{\widehat{=}} dom(E) \cup \{\iota\} \\
dom(E,x : A) &\mathrel{\widehat{=}} dom(E) \cup \{x\} \\
dom(E,X<:A) &\mathrel{\widehat{=}} dom(E) \cup \{X\} \\
dom(E,\alpha \prec: p) &\mathrel{\widehat{=}} dom(E) \cup \{\alpha\} \\
dom(E,\alpha :\succ p) &\mathrel{\widehat{=}} dom(E) \cup \{\alpha\}
\end{aligned}$$

This definition also applies to Γ *and* Π.

The different kinds of variables are assumed to come from syntactically distinct collections, so we expend no effort differentiating between them.

Definition 5.3 (Free Variables)

$$\begin{aligned}
\mathrm{FV}(\alpha) &\mathrel{\widehat{=}} \{\alpha\} \\
\mathrm{FV}(\pi) &\mathrel{\widehat{=}} \emptyset
\end{aligned}$$

$$
\begin{array}{rcl}
\mathrm{FV}(\langle p \rangle) & \widehat{=} & \mathrm{FV}(p) \\
\mathrm{FV}(\langle p\uparrow \rangle) & \widehat{=} & \mathrm{FV}(p) \\
\mathrm{FV}(\bigcup[K_1..K_n]) & \widehat{=} & \bigcup_{i\in 1..n} \mathrm{FV}(K_i) \\
\\
\mathrm{FV}(X) & \widehat{=} & \{X\} \\
\mathrm{FV}(\mathsf{Top}^K) & \widehat{=} & \mathrm{FV}(K) \\
\mathrm{FV}([l_i : \Theta_i{}^{i\in 1..n}]_q^p) & \widehat{=} & \bigcup_{i\in 1..n} \mathrm{FV}(\Theta_i) \cup \mathrm{FV}(p) \cup \mathrm{FV}(q) \\
\mathrm{FV}(\mu(X)A) & \widehat{=} & \mathrm{FV}(A)\backslash\{X\} \\
\mathrm{FV}(\exists(\alpha \prec: p)A) & \widehat{=} & \mathrm{FV}(p) \cup (\mathrm{FV}(A)\backslash\{\alpha\}) \\
\mathrm{FV}(A \to \Theta) & \widehat{=} & \mathrm{FV}(A) \cup \mathrm{FV}(\Theta) \\
\mathrm{FV}(\forall(X<:A)\Theta) & \widehat{=} & \mathrm{FV}(A) \cup (\mathrm{FV}(\Theta)\backslash\{X\}) \\
\mathrm{FV}(\forall(\alpha \prec: p)\Theta) & \widehat{=} & \mathrm{FV}(p) \cup (FV(\Theta)\backslash\{\alpha\}) \\
\mathrm{FV}(\forall(\alpha :\succ p)\Theta) & \widehat{=} & \mathrm{FV}(p) \cup (FV(\Theta)\backslash\{\alpha\}) \\
\\
\mathrm{FV}(\varsigma(s : A, \Gamma)b) & \widehat{=} & \mathrm{FV}(A) \cup \mathrm{FV}(\Gamma) \cup (\mathrm{FV}(b)\backslash(\{s\} \cup \mathit{dom}(\Gamma))) \\
\\
\mathrm{FV}(x) & \widehat{=} & \{x\} \\
\mathrm{FV}(\iota) & \widehat{=} & \emptyset \\
\mathrm{FV}(\textbf{\textit{fold}}(A, v)) & \widehat{=} & \mathrm{FV}(A) \cup \mathrm{FV}(v) \\
\mathrm{FV}(\textbf{\textit{hide}}\ p\ \textbf{\textit{as}}\ \alpha \prec: q\ \textbf{\textit{in}}\ v{:}A) & \widehat{=} & \mathrm{FV}(p) \cup \mathrm{FV}(q) \cup ((\mathrm{FV}(v) \cup \mathrm{FV}(B))\backslash\{\alpha\}) \\
\mathrm{FV}(v.l\langle\Delta\rangle) & \widehat{=} & \mathrm{FV}(v) \cup \mathrm{FV}(\Delta) \\
\mathrm{FV}(\emptyset) & \widehat{=} & \emptyset \\
\mathrm{FV}(v, \Delta) & \widehat{=} & \mathrm{FV}(v) \cup \mathrm{FV}(\Delta) \\
\mathrm{FV}(A, \Delta) & \widehat{=} & \mathrm{FV}(A) \cup \mathrm{FV}(\Delta) \\
\mathrm{FV}(p, \Delta) & \widehat{=} & \mathrm{FV}(p) \cup \mathrm{FV}(\Delta) \\
\mathrm{FV}(v.l \Leftarrow \varsigma(s : A, \Gamma)b) & \widehat{=} & \mathrm{FV}(v) \cup \mathrm{FV}(\varsigma(s : A, \Gamma)b) \\
\mathrm{FV}(\textbf{\textit{let}}\ x : A = a\ \textbf{\textit{in}}\ b) & \widehat{=} & \mathrm{FV}(A) \cup \mathrm{FV}(a) \cup (\mathrm{FV}(b)\backslash\{x\}) \\
\mathrm{FV}(\textbf{\textit{unfold}}(v)) & \widehat{=} & \mathrm{FV}(v) \\
\mathrm{FV}(\textbf{\textit{expose}}\ v\ \textbf{\textit{as}}\ \alpha \prec: p, x{:}A\ \textbf{\textit{in}}\ b{:}B) & \widehat{=} & \mathrm{FV}(v) \cup \mathrm{FV}(p) \cup (\mathrm{FV}(A)\backslash\{\alpha\}) \\
 & & \quad \cup (\mathrm{FV}(b)\backslash\{\alpha, x\}) \cup (\mathrm{FV}(B)\backslash\{\alpha\}) \\
\mathrm{FV}(\textbf{\textit{new}}\ \alpha \lhd p\ \textbf{\textit{in}}\ a) & \widehat{=} & \mathrm{FV}(p) \cup (\mathrm{FV}(a)\backslash\{\alpha\}) \\
\mathrm{FV}([l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i\in 1..n}]_q^p) & \widehat{=} & \bigcup_{i\in 1..n} \mathrm{FV}(\varsigma(s_i : A_i, \Gamma_i)b_i) \cup \mathrm{FV}(p) \cup \mathrm{FV}(q) \\
\mathrm{FV}(\emptyset) & \widehat{=} & \emptyset
\end{array}
$$

$$\frac{}{\emptyset \vdash \diamond}\ (\text{Env } \emptyset) \qquad \frac{E;K \vdash A \quad X \notin \mathsf{dom}(E)}{E, X{<:}A \vdash \diamond}\ (\text{Env } X) \qquad \frac{E;K \vdash A \quad x \notin \mathsf{dom}(E)}{E, x : A \vdash \diamond}\ (\text{Env } x)$$

$$\frac{E \vdash p \quad \alpha \notin \mathsf{dom}(E)}{E, \alpha \prec: p \vdash \diamond}\ (\text{Env } \alpha \prec:) \qquad \frac{E \vdash p \quad \alpha \notin \mathsf{dom}(E)}{E, \alpha :\succ p \vdash \diamond}\ (\text{Env } \alpha :\succ)$$

$$\frac{E;\langle p \rangle \vdash [l_i : \Theta_i{}^{i \in 1..n}]^p_q \quad \iota \notin \mathsf{dom}(E)}{E, \iota : [l_i : \Theta_i{}^{i \in 1..n}]^p_q \vdash \diamond}\ (\text{Env Location})$$

Figure 5.3: Well-formed Environments

$$\begin{aligned}
\mathrm{FV}(x : A, \Gamma) &\mathrel{\widehat{=}} \mathrm{FV}(A) \cup \mathrm{FV}(\Gamma) \\
\mathrm{FV}(X{<:}A, \Gamma) &\mathrel{\widehat{=}} \mathrm{FV}(A) \cup (\mathrm{FV}(\Gamma) \setminus \{X\}) \\
\mathrm{FV}(\alpha \prec: p, \Gamma) &\mathrel{\widehat{=}} \mathrm{FV}(p) \cup (\mathrm{FV}(\Gamma) \setminus \{\alpha\}) \\
\mathrm{FV}(\alpha :\succ p, \Gamma) &\mathrel{\widehat{=}} \mathrm{FV}(p) \cup (\mathrm{FV}(\Gamma) \setminus \{\alpha\})
\end{aligned}$$

Well-formed Environments The rules in Figure 5.3 define well-formed environments. Both (Env X) and (Env x) require that the type A be well-formed with respect to some permission which is well-formed in E. The bounds on context variables in both (Env $\alpha \prec:$) and (Env $\alpha :\succ$) can be any context valid in environment E. Finally, (Env Location) specifies that locations have object type. Note that each variable can appear only once in the domain of a well-formed environment.

Well-defined Contexts and Nesting Figure 5.4 defines well-formed contexts and their nesting relation. Contexts are either constants from C, by (Context π), or variables declared in E, by (Context α). The rule (In π) states that the nesting relation includes $\prec:_C$, the nesting of constants. The $\prec:$ relation is extended by assumption (In $\prec:$) and (In $:\succ$), where in the last case the assumption $\alpha :\succ p$ is reversed to become part of the inside relation. (In ε) states that ε is the maximal element — this case covers instances where this would not have otherwise been provable. The rule (In ε) can be omitted when there is no maximal context — a consequence is that contexts

$$\frac{E \vdash \diamond \quad \pi \in C}{E \vdash \pi}\ (\text{Context } \pi) \qquad \frac{\alpha \in \mathsf{dom}(E)}{E \vdash \alpha}\ (\text{Context } \alpha)$$

$$\frac{E \vdash \diamond \quad \pi \prec:_C \pi'}{E \vdash \pi \prec: \pi'}\ (\text{In } \pi) \qquad \frac{\alpha \prec: p \in E}{E \vdash \alpha \prec: p}\ (\text{In } \prec:) \qquad \frac{\alpha :\succ p \in E}{E \vdash p \prec: \alpha}\ (\text{In } :\succ)$$

$$\frac{E \vdash p}{E \vdash p \prec: \varepsilon}\ (\text{In } \varepsilon) \qquad \frac{E \vdash p}{E \vdash p \prec: p}\ (\text{In Refl}) \qquad \frac{E \vdash p \prec: q \quad E \vdash q \prec: r}{E \vdash p \prec: r}\ (\text{In Trans})$$

Figure 5.4: Well-formed contexts and their nesting

will form a forest. Context nesting is also reflexive (In Refl) and transitive (In Trans).

Well-formed Permission and Subpermissions Figure 5.5 defines well-formed permissions and the subpermission relation $E \vdash K \subseteq K'$. For each well-formed context, we can form the point permission (Perm p) and the upset permission (Perm $p\uparrow$). Unions of permissions are well-formed when their constituent permissions are (Perm Union).

The Subpermission relationship is reflexive and transitive, by (SubPerm Refl) and (SubPerm Trans). A point permission is a subpermission of the corresponding upset permission by rule (SubPerm p). The rule (SubPerm $\prec:$) lifts the nesting relation between contexts to upset permissions, based on the following intuition: starting with $q \prec: p$, (SubPerm $\prec:$) implies that $\langle p\uparrow\rangle \subseteq \langle q\uparrow\rangle$. If $p' \in \langle p\uparrow\rangle$, then from our understanding of the set underlying $\langle p\uparrow\rangle$, we can derive that $p \prec: p'$. From transitivity of $\prec:$ we obtain $q \prec: p'$, and hence $p' \in \langle q\uparrow\rangle$.

Rules (Perm Union), (SubPerm Union-LB), and (SubPerm Union-UB) extend permissions and the subpermission relation to unions in an obvious manner (following, for example, Pierce [157]).

Well-formed Types Figure 5.6 defines well-formed types.

The rule (Type X) helps ensure that type substitution does not violate permis-

$$\frac{E \vdash p}{E \vdash \langle p \rangle}\ (\text{Perm } p) \qquad \frac{E \vdash p}{E \vdash \langle p\uparrow \rangle}\ (\text{Perm } p\uparrow) \qquad \frac{E \vdash K_i \quad \forall i \in 1..n}{E \vdash \bigcup[K_1..K_n]}\ (\text{Perm Union})$$

$$\frac{E \vdash p}{E \vdash \langle p \rangle \subseteq \langle p\uparrow \rangle}\ (\text{SubPerm } p) \qquad \frac{E \vdash q \prec: p}{E \vdash \langle p\uparrow \rangle \subseteq \langle q\uparrow \rangle}\ (\text{SubPerm} \prec:) \qquad \frac{E \vdash K}{E \vdash K \subseteq K}\ (\text{SubPerm Refl})$$

$$\frac{E \vdash K \subseteq K' \quad E \vdash K' \subseteq K''}{E \vdash K \subseteq K''}\ (\text{SubPerm Trans}) \qquad \frac{E \vdash K_i \subseteq K \quad \forall i \in 1..n}{E \vdash \bigcup[K_1..K_n] \subseteq K}\ (\text{SubPerm Union-LB})$$

$$\frac{E \vdash \bigcup[K_1..K_n] \quad i \in 1..n}{E \vdash K_i \subseteq \bigcup[K_1..K_n]}\ (\text{SubPerm Union-UB})$$

Figure 5.5: Well-formed Permissions and Subpermissions

sions by requiring that the type variable is a valid type only when enough permission is given to construct its bound. The Permissibility Lemma (Lemma 5.8 here or Lemma 4.13 in the previous chapter), which is crucial for soundness, states that having permission to construct a type implies that there is enough permission to access all of its subtypes. Since any subtype of X's bound can be substituted for X, we require that accessing X requires at least the permission needed to access its bound. This confirms our choice in the rule (Type X).

The rule (Type Object) is similar to that of the previous chapter, although method types are more complicated and a typing environment has been added to account for free type and context variables.

The rule (Type Rec) for recursive types places the bound Top^K on the variable X, where K is sufficient permission to form type A. This is to ensure that both the type $\mu(X)A$ and its unfolding $A\{{}^{\mu(X)A}/_{X}\}$ are well-formed given permission K.

By rule (Type Exists), existential types can be formed with contexts and a given upper bound. From clause $E \vdash K$ it follows that $\alpha \notin K$, which implies that α does not appear free in the top level owner position of an object type anywhere within A,

$$\frac{X{<:}A \in E \quad E;K \vdash A}{E;K \vdash X}\ (\text{Type } X) \qquad \frac{E \vdash K}{E;K \vdash \mathsf{Top}^K}\ (\text{Type Top})$$

$$\frac{E;\langle q\uparrow\rangle \vdash \Theta_i\ meth \qquad \forall i \in 1..n \quad E \vdash q \prec: p}{E;\langle p\rangle \vdash [l_i : \Theta_i{}^{i\in 1..n}]_q^p}\ (\text{Type Object})\ (l_i \text{ distinct})$$

$$\frac{E, X{<:}\mathsf{Top}^K; K \vdash A}{E;K \vdash \mu(X)A}\ (\text{Type Rec}) \qquad \frac{E, \alpha \prec: p; K \vdash A \quad E \vdash K}{E;K \vdash \exists(\alpha \prec: p)A}\ (\text{Type Exists})$$

$$\frac{E;K \vdash A \quad E \vdash K \subseteq K'}{E;K' \vdash A}\ (\text{Type Allow})$$

Figure 5.6: Well-formed Types

because the first hypothesis cannot hold when $\alpha \in \mathsf{dom}(E)$. This guarantees that the existential cannot hide essential permission related information, which would break our desired containment invariant. We discuss this point in detail in Section 5.4.

The rule (Type Allow) behaves as before: a type which is well-formed from some permission is well-formed with a larger permission.

Well-formed Method Types Method types include function types, and types parameterised by type and context variables, both with appropriate bounds. The type rules are given in Figure 5.7. The two rules (Type All $\prec:$) and (Type All $:\succ$) allow the permission to be extended by the point permission corresponding to the context parameter. Thus the new context can be used in the top-level owner position of subsequent types in the method type. This facilitates methods which temporarily have access to contexts which otherwise it would not have permission to access and thus a kind of borrowing. The premise $E \vdash K$ for these rules ensures that α does not appear in K.

$$\frac{E;K \vdash A}{E;K \vdash A\ \mathit{meth}}\ \text{(Type Return)} \qquad \frac{E;K \vdash A \quad E;K \vdash \Theta\ \mathit{meth}}{E;K \vdash A \to \Theta\ \mathit{meth}}\ \text{(Type Arrow)} \qquad \frac{E,X{<:}A;K \vdash \Theta\ \mathit{meth}}{E;K \vdash \forall(X{<:}A)\Theta\ \mathit{meth}}\ \text{(Type All)}$$

$$\frac{E,\alpha \prec: p;K \cup \langle\alpha\rangle \vdash \Theta\ \mathit{meth} \quad E \vdash K}{E;K \vdash \forall(\alpha \prec: p)\Theta\ \mathit{meth}}\ \text{(Type All } \prec:\text{)} \qquad \frac{E,\alpha :\succ p;K \cup \langle\alpha\rangle \vdash \Theta\ \mathit{meth} \quad E \vdash K}{E;K \vdash \forall(\alpha :\succ p)\Theta\ \mathit{meth}}\ \text{(Type All } :\succ\text{)}$$

Figure 5.7: Well-formed Method Type

Well-formed Subtyping The subtyping rules in Figure 5.8 specify a transitive and reflexive subtyping relation which depends upon the given permission. A type variable is a subtype of its bound, given any K sufficient to construct that bound (Sub X). The rule (Sub Top) states that Top^K is the supertype of all types which can be constructed using permission K. The rule (Sub Object) allows neither the owner nor the representation context to vary, only which methods are present. Subtyping of recursive types (Sub Rec) follows the form in [3], but also includes the bound Top^K to ensure that the unfoldings of the types on either side of the $<:$ are valid, following (Type Rec). The rule (Sub Exists) follows the pattern of existential type subtyping, except that it applies only to contexts. Finally, (Sub Allow) states that a subtyping relationship valid for some permission is valid given a larger permission.

Well-typed Expressions Figure 5.9 defines well-typed expressions. Expressions are typed against a typing environment and a permission. The permission bounds the owners of objects and locations permitted in the top-level of the given expression. The type rules depend upon two auxiliary functions.

The function $\lfloor\Gamma\rfloor_C$ takes a method formal parameter list Γ and return type C and produces the corresponding method type Θ:

Definition 5.4 ($\lfloor\Gamma\rfloor_C$)

$$\begin{aligned}
\lfloor\emptyset\rfloor_C &\mathrel{\widehat{=}} C \\
\lfloor x:A,\Gamma\rfloor_C &\mathrel{\widehat{=}} A \to \lfloor\Gamma\rfloor_C \\
\lfloor X{<:}A,\Gamma\rfloor_C &\mathrel{\widehat{=}} \forall(X{<:}A)\lfloor\Gamma\rfloor_C \\
\lfloor \alpha \prec: p,\Gamma\rfloor_C &\mathrel{\widehat{=}} \forall(\alpha \prec: p)\lfloor\Gamma\rfloor_C
\end{aligned}$$

$$\frac{E;K \vdash A}{E;K \vdash A<:A}\ (\text{Sub Refl}) \qquad \frac{E;K \vdash A<:B \quad E;K \vdash B<:C}{E;K \vdash A<:C}\ (\text{Sub Trans})$$

$$\frac{X<:A \in E \quad E;K \vdash A}{E;K \vdash X<:A}\ (\text{Sub } X) \qquad \frac{E;K \vdash A}{E;K \vdash A<:\mathsf{Top}^K}\ (\text{Sub Top})$$

$$\frac{E;\langle q\uparrow\rangle \vdash \Theta_i\ meth \quad \forall i \in 1..n+m \quad E \vdash q \prec: p}{E;\langle p\rangle \vdash [l_i : \Theta_i{}^{i\in 1..n+m}]^p_q <: [l_i : \Theta_i{}^{i\in 1..n}]^p_q}\ (\text{Sub Object})\ (l_i \text{ distinct})$$

$$\frac{E;K \vdash \mu(X)A \quad E;K \vdash \mu(Y)B \quad E,Y<:\mathsf{Top}^K,X<:Y;K \vdash A<:B}{E;K \vdash \mu(X)A<:\mu(Y)B}\ (\text{Sub Rec})$$

$$\frac{E \vdash p \prec: p' \quad E,\alpha \prec: p;K \vdash A<:A' \quad E \vdash K}{E;K \vdash \exists(\alpha \prec: p)A<:\exists(\alpha \prec: p')A'}\ (\text{Sub Exists})$$

$$\frac{E;K \vdash A<:B \quad E \vdash K \subseteq K'}{E;K' \vdash A<:B}\ (\text{Sub Allow})$$

Figure 5.8: Subtyping

$$\lfloor \alpha :\succ p,\Gamma \rfloor_C \ \widehat{=}\ \forall(\alpha :\succ p)\lfloor \Gamma \rfloor_C$$

The function $\lceil \Gamma \rceil$ produces the additional permission given by the context parameters in Γ. These are added to the permissions available when typing a method body:

Definition 5.5 ($\lceil \Gamma \rceil$)

$$\begin{aligned} \lceil \emptyset \rceil &\ \widehat{=}\ \mathit{void} \\ \lceil x : A,\Gamma \rceil &\ \widehat{=}\ \lceil \Gamma \rceil \end{aligned}$$

$$\begin{aligned}\lceil X{<:}A, \Gamma\rceil &\mathrel{\widehat{=}} \lceil\Gamma\rceil \\ \lceil \alpha \prec: p, \Gamma\rceil &\mathrel{\widehat{=}} \langle\alpha\rangle \cup \lceil\Gamma\rceil \\ \lceil \alpha :\succ p, \Gamma\rceil &\mathrel{\widehat{=}} \langle\alpha\rangle \cup \lceil\Gamma\rceil\end{aligned}$$

Variable typing is by assumption (Val x) with permission sufficient to construct the variable's type. Locations are similarly typed by assumption, where the permission required is the point permission for the owner of the location's type (Val Location).

In (Val Object) permission $\langle p\rangle$ is required to create an object with owner context p. The body b_i of method $\varsigma(s_i : A, \Gamma_i)b_i$ is typed against an environment extended with the self parameter s_i having type A, the type for the object being typed, and the formal parameter list Γ_i. The permission the method is typed against is the upset permission $\langle q\uparrow\rangle$, where q is the representation context, extended with the point permissions for contexts declared in Γ_i. This means that the method bodies can have access to any location, object, or variable with permission p' such that $q \prec: p'$, that is anything outside of the representation context, as well as any object or variable with owner in Γ_i. Because the contexts in Γ_i are variables, no location with such an owner can appear in the method body b_i. The return type of the method body C_i and the formal parameters Γ_i must combine to give the method type Θ_i.

In (Val Select), given that the selected method's type is Θ_j, the clause $E; K \vdash (\Theta_j)(\Delta) \Rightarrow C_j$ guarantees that the arguments Δ are correct in number, kind and type, and that they and the return type C_j are all typable given permission K. This ensures that enough permission has been given to access the method's arguments and return value. The type rules for well-typed argument lists are given in Figure 5.10 and are explained in the next subsection.

In (Val Update) the new method body is typed against a permission K' which in effect is the intersection between the contexts accessible inside the object, $\langle q\uparrow\rangle$, and the contexts visible in the surrounding expression, K, along with the point permissions for the context parameters declared in Γ. This 'intersection' of permissions prevents any illegal locations from being added into an object, while maintaining the constraints on the expression performing the method update.

The type rule (Val Fold), (Val Unfold), and (Val Let) are standard, except that they also carry the same permission through to the subterms [47].

The first clause in (Val New) types the term a with the additional assumption about the new context α and with the permission extended with $\langle\alpha\rangle$. This allows objects to be created within the expression a with the new context α as owner. The second clause

(Val x)
$$\frac{x : A \in E \quad E;K \vdash A}{E;K \vdash x : A}$$

(Val Location)
$$\frac{\iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q \in E}{E;\langle p\rangle \vdash \iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q}$$

(Val Object) (where $A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_q$ and $\Theta_i \equiv \lfloor \Gamma_i \rfloor_{C_i}$)
$$\frac{E, s_i : A, \Gamma_i; \langle q\uparrow\rangle \cup \lceil \Gamma_i \rceil \vdash b_i : C_i \quad \forall i \in 1..n}{E;\langle p\rangle \vdash [l_i = \varsigma(s_i : A, \Gamma_i) b_i{}^{i\in 1..n}]^p_q : A}$$

(Val Select)
$$\frac{E;K \vdash v : [l_i : \Theta_i{}^{i\in 1..n}]^p_q \quad E;K \vdash (\Theta_j)(\Delta) \Rightarrow C_j \quad j \in 1..n}{E;K \vdash v.l_j\langle\Delta\rangle : C_j}$$

(Val Update) (where $A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_q$ and $\Theta_j \equiv \lfloor \Gamma \rfloor_{C_j}$)
$$\frac{\begin{array}{c} E;K \vdash v : A \quad E, s : A, \Gamma; K' \vdash b : C_j \\ E, \Gamma \vdash K' \subseteq \langle q\uparrow\rangle \cup \lceil \Gamma \rceil \quad E, \Gamma \vdash K' \subseteq K \cup \lceil \Gamma \rceil \quad j \in 1..n \end{array}}{E;K \vdash v.l_j \Leftarrow \varsigma(s : A, \Gamma) b : A}$$

(Val Fold) (where $A \equiv \mu(X)B$)
$$\frac{E;K \vdash v : B\{{}^{A}/_{X}\}}{E;K \vdash \mathbf{fold}(A, v) : A}$$

(Val Unfold) (where $A \equiv \mu(X)B$)
$$\frac{E;K \vdash v : A}{E;K \vdash \mathbf{unfold}(v) : B\{{}^{A}/_{X}\}}$$

(Val Let)
$$\frac{E;K \vdash a : A \quad E, x : A; K \vdash b : B}{E;K \vdash \mathbf{let}\ x : A = a\ \mathbf{in}\ b : B}$$

(Val New)
$$\frac{E, \alpha \prec: p; K \cup \langle\alpha\rangle \vdash a : A \quad E;K \vdash A}{E;K \vdash \mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ a : A}$$

(Val Hide)
$$\frac{E \vdash p \prec: q \quad E;K \vdash v\{{}^{p}/_{\alpha}\} : A\{{}^{p}/_{\alpha}\} \quad E;K \vdash \exists(\alpha \prec: q)A}{E;K \vdash \mathbf{hide}\ p\ \mathbf{as}\ \alpha \prec: q\ \mathbf{in}\ v{:}A : \exists(\alpha \prec: q)A}$$

(Val Expose)
$$\frac{E;K \vdash v : \exists(\alpha \prec: p)A \quad E;K \vdash B \quad E, \alpha \prec: p, x : A; K \cup \langle\alpha\rangle \vdash b : B}{E;K \vdash \mathbf{expose}\ v\ \mathbf{as}\ \alpha \prec: p, x{:}A\ \mathbf{in}\ b{:}B : B}$$

(Val Subsumption)
$$\frac{E;K \vdash a : A \quad E;K' \vdash A <: B \quad E \vdash K \subseteq K'}{E;K' \vdash a : B}$$

Figure 5.9: Well-typed Expressions

prevents the new context from appearing in the resulting type. Objects created with owner α can therefore be used to initialise the representation of an object, as in the following example:

new $\alpha \lhd p$ **in**	*create a new context*
let $y = [\ldots]^{\alpha}_{\_}$ **in**	*create object in that context*
let $x = [\ldots rep = \varsigma()y\ldots]^{p}_{\alpha}$ **in**	*set representation context to* α*; store y in new object bound to x*
hide α **as** $\alpha \prec: p$ **in** $x{:}[\ldots rep : [\ldots]^{\alpha}_{\_}\ldots]^{p}_{\alpha}$	*hide new context*

The objects with this context as owner must either be embedded in other objects, or discarded, because they cannot be directly returned. Without this restriction statically checking the type system would be difficult, if not impossible, though we do not know which.

The rule (Val Hide) parallels the standard rule for packing existential types. The additional clause, $E;K \vdash \exists(\alpha \prec: q)A$, guarantees that the owner of top level object types in A cannot be hidden (see Section 5.4 for a discussion). Similarly, (Val Expose) parallels the usual rule for opening existential types [3].

Finally, (Val Subsumption) allows an expression to be given any supertype of its type and to be used where a larger permission is given.

Well-typed Actual Parameters The type rules in Figure 5.10 check that the arguments supplied to a method are correct in number and type. Underlying the rules are the usual rules for function and type application. All types and values, including the return type, must be accessible given permission K.

Well-typed Stores and Configurations The type rules for stores and configurations are given in Figure 5.11. Store typing is as before, requiring that the type of each object is the same as the location where it is stored. Configuration typing is performed in an environment consisting of two parts: Π to account for the nesting of free context variables and E' for the types of all locations in the store. Note that all object owners and representation contexts which are not constant are included in the domain of Π, though Π may also include other contexts which were created but not attached to objects.

(Arg Empty)
$$\frac{E;K \vdash C}{E;K \vdash (C)(\emptyset) \Rightarrow C}$$

(Arg Val)
$$\frac{E;K \vdash v : A \quad E;K \vdash (\Theta)(\Delta) \Rightarrow C}{E;K \vdash (A \to \Theta)(v,\Delta) \Rightarrow C}$$

(Arg Type)
$$\frac{E;K \vdash B <: A \quad E;K \vdash (\Theta\{^{B}/_{X}\})(\Delta) \Rightarrow C}{E;K \vdash (\forall(X <: A)\Theta)(B,\Delta) \Rightarrow C}$$

(Arg Context $\prec:$)
$$\frac{E \vdash q \prec: p \quad E;K \vdash (\Theta\{^{q}/_{\alpha}\})(\Delta) \Rightarrow C}{E;K \vdash (\forall(\alpha \prec: p)\Theta)(q,\Delta) \Rightarrow C}$$

(Arg Context $:\succ$)
$$\frac{E \vdash p \prec: q \quad E;K \vdash (\Theta\{^{q}/_{\alpha}\})(\Delta) \Rightarrow C}{E;K \vdash (\forall(\alpha :\succ p)\Theta)(q,\Delta) \Rightarrow C}$$

Figure 5.10: Well-typed Actual Parameters

(Val Store)
$$\frac{E;\langle p \rangle \vdash o : [l_i : \Theta_i{}^{i\in 1..n}]^{p}_{q} \quad \iota : [l_i : \Theta_i{}^{i\in 1..n}]^{p}_{q} \in E \quad \forall \iota \mapsto o \in \sigma}{E \vdash \sigma}$$

(Val Config) (where $E \equiv \Pi, E'$)
$$\frac{E \vdash \sigma \quad E;K \vdash a : A \quad \mathsf{dom}(\sigma) = \mathsf{dom}(E')}{E;K \vdash (\Pi,\sigma,a) : A}$$

Figure 5.11: Well-typed Stores and Configurations

5.4 Top, Existential Quantification and Containment

Initially, adding Top type and existential quantification over contexts to our type system was problematic. In particular, without the special care that we now provide, these types could be used to construct terms which violated the containment invariant. Here we recall those problems and describe why our solution avoids them.

5.4.1 Top(s)

In type systems which have it, Top type is the largest type in that every type is a subtype of Top. It is often included in a type system to simplify the treatment of recursive types and their subtyping and so that unbounded universal quantification can be modelled in terms of bounded universal quantification [3].

Adding Top type directly to a type system which enforces structural constraints on object graphs, as ours does, is problematic because it allows subtyping to forget the owner of an object's type. For example, $E;K \vdash [l_i : \Theta_i{}^{i\in 1..n}]^p_q <: \mathsf{Top}$ would have been a valid judgement, which would have allowed any object to access a value of type $[l_i : \Theta_i{}^{i\in 1..n}]^p_q$. Any object could access values of type Top, but this unfortunately meant that it was easy to construct an object graph which did not satisfy the containment invariant — any object with a field of type Top could store a location owned by p, even though that object may not otherwise have permission to access such locations.

We have considered a number of approaches for dealing with this Top problem.

The first approach is to do nothing. This is based on the observation that nothing can be done with or to a value of type Top, except testing pointer equality, so having an otherwise inaccessible reference does not matter. The problem with this approach is that it becomes difficult to state the containment invariant simply, especially in the presence of universal type quantification, and at least as difficult to prove the desired properties.

The second approach revolves around forbidding the undesirable types, that is, the types of objects whose methods access values of type Top. However, it seems difficult to characterise exactly what those undesirable types may be. It may be the case that almost all types involving Top are bad. This choice suggests that Top serves no purpose at all, which leads to the next approach.

The third approach is to exclude Top altogether. But this means we lose unbounded universal quantification, and the treatment of recursive types becomes more difficult. While not an insurmountable problem, the resulting loss of elegance made us search

harder.

The final approach, which is adopted here, is based on observation that an object type implicitly contains the permission information required to access it, and that subtyping must preserve that information. The idea is to make the permission information increase monotonically with subtyping. Firstly, let $E;K \vdash_{\mathsf{min}} A$ mean that K is the minimum permission required for A to be well-formed. This is defined as a part of the proof of Lemma 5.8 in Appendix A. Assume that $E;K \vdash_{\mathsf{min}} A$ and $E;K' \vdash_{\mathsf{min}} B$. Then the monotonicity we seek is that if for some $K^\dagger$ we have $E;K^\dagger \vdash A{<:}B$, then $E \vdash K \subseteq K'$. Now the minimum permission required to access an object with type $[l_i : \Theta_i{}^{i\in 1..n}]^p_q$ is $\langle p \rangle$. Thus subtyping to Top requires a Top type which has at least that much permission. Thus we annotate Top types with the minimum permission required to access values of that type. Thus $E;\langle p \rangle \vdash [l_i : \Theta_i{}^{i\in 1..n}]^p_q{<:}\mathsf{Top}^{\langle p \rangle}$ is valid, as well as $E;K \vdash \mathsf{Top}^{K'}{<:}\mathsf{Top}^{K}$, for any K, K' such that that $E \vdash K' \subseteq K$. Using a permission-indexed family of Top types retains an elegant treatment of both recursive types and universal polymorphism in the presence of subtyping, while maintaining the containment invariant.

5.4.2 Existential Quantification

Once the **new** term was added to the calculus it became necessary both for reasons of expressiveness and to avoid the problems of dependent typing to add existential quantification over contexts. Unfortunately, directly adding existential quantifications introduced the possibility of abstracting away the owner context from an object type. Since permissions control access to objects based on the owner context, hiding this context leads to violations in the containment invariant. These violations are more serious than those related to Top type.

In order to illustrate this point and demonstrate our solution, we first fix some notation which we use only in this section: let ι^p represent a location whose owner is p, and the turnstile $\vdash_?$ indicate judgements which are questionable.

Consider the suspect value **hide** p **as** $\alpha \prec: p$ **in** $\iota^p{:}[l_i : \Theta_i{}^{i\in 1..n}]^\alpha_\alpha$ which hides the owner of a location. It has type $\exists(\alpha \prec: p)[l_i : \Theta_i{}^{i\in 1..n}]^\alpha_\alpha$, which reveals the full method suite of the object. Thus access to this value gives almost the same privileges as having access to ι^p. Since we can determine nothing about the owner from this type, one could conclude that permission void is all that's needed to access this value.

This means that the judgement $E;K \vdash_? \exists(\alpha \prec: p)[l_i : \Theta_i{}^{i\in 1..n}]^\alpha_\alpha$ holds, for every K such that $E \vdash K$, since $E \vdash \mathsf{void} \subseteq K$. It follows that the value **hide** p **as** $\alpha \prec: p$ **in** $\iota^p{:}$

$[l_i : \Theta_i{}^{i\in 1..n}]^{\alpha}_{\alpha}$ may be stored in objects which would not otherwise have access to ι^p.

Deriving a typing for this suspect term does actually require the permission information we would expect it to.

$$\frac{E;\langle p\rangle \vdash \iota^p : [l_i : \Theta_i{}^{i\in 1..n}]^p_p \qquad E;K \vdash_? \exists(\alpha \prec: p)[l_i : \Theta_i{}^{i\in 1..n}]^{\alpha}_{\alpha}}{E;\langle p\rangle \vdash_? \textbf{hide } p \textbf{ as } \alpha \prec: p \textbf{ in } \iota^p{:}[l_i : \Theta_i{}^{i\in 1..n}]^{\alpha}_{\alpha} : \exists(\alpha \prec: p)[l_i : \Theta_i{}^{i\in 1..n}]^{\alpha}_{\alpha}} \text{ (Val Hide)}$$

In terms of preserving the containment invariant, the type rule (Val Hide) seems to be on the right track. The permission required to access location ι^p is present even when the owner of this location is hidden in the type. This is what we want since the proof of containment relies fundamentally on the fact that the owners accessible in an expression are all present in the permission.

The problem must therefore be with the type itself. So the way to address this problem seems to be by ensuring that the offending type is not valid in the type system, thus excluding also all terms of that type as well. This is what the additional judgement in (Val Hide) does.

Now lets consider an instance of that judgement $E;K \vdash_? \exists(\alpha \prec: p)[l_i : \Theta_i{}^{i\in 1..n}]^{\alpha}_{\alpha}$:

$$\frac{\dfrac{E,\alpha \prec: p;\langle\alpha\rangle \vdash [l_i : \Theta_i{}^{i\in 1..n}]^{\alpha}_{\alpha} \quad E,\alpha \prec: p \vdash_? \langle\alpha\rangle \subseteq K}{E,\alpha \prec: p;K \vdash_? [l_i : \Theta_i{}^{i\in 1..n}]^{\alpha}_{\alpha}} \text{ (Type Allow)} \qquad E \vdash K}{E;K \vdash_? \exists(\alpha \prec: p)[l_i : \Theta_i{}^{i\in 1..n}]^{\alpha}_{\alpha}} \text{ (Type Exists)}$$

This cannot be derived because the judgement $E,\alpha \prec: p \vdash_? \langle\alpha\rangle \subseteq K$ fails due to the judgement $E \vdash K$, which implies that $\alpha \notin FV(K)$, since $\alpha \notin \mathsf{dom}(E)$. We can see by induction on the shape of K that $E,\alpha \prec: p \not\vdash \langle\alpha\rangle \subseteq K$. Clearly, if K were a union, we can proceed by induction on the components of the union. K cannot be of the form $\langle q\rangle$, because the only valid q would be α, which is disallowed. Consider then K of the form $\langle q\uparrow\rangle$ for some q. If $E,\alpha \prec: p \vdash_? \langle\alpha\rangle \subseteq \langle q\uparrow\rangle$, then $E,\alpha \prec: p \vdash_? \langle\alpha\uparrow\rangle \subseteq \langle q\uparrow\rangle$, from which follows $E,\alpha \prec: p \vdash_? q \prec: \alpha$. But this cannot be proven in the type system. Thus there is no K which makes the above derivation valid. We have show that the suspect value **hide** p **as** $\alpha \prec: p$ **in** $\iota^p{:}[l_i : \Theta_i{}^{i\in 1..n}]^{\alpha}_{\alpha}$ cannot be typed. Hence existential types cannot be used to hide information essential for preserving the containment invariant.

5.5 Dynamic Semantics

The operational semantics of the calculus are presented in Figure 5.12. They define an evaluation relation between initial and final configurations, $(\Pi,\sigma,a) \Downarrow (\Pi',\sigma',v)$.

(Subst Value)

$$\frac{}{(\Pi, \sigma, v) \Downarrow (\Pi, \sigma, v)}$$

(Subst Object)

$$\frac{\sigma_1 = \iota \mapsto o, \sigma_0 \quad \iota \notin \mathsf{dom}(\sigma_0)}{(\Pi, \sigma_0, o) \Downarrow (\Pi, \sigma_1, \iota)}$$

(Subst Select) where $j \in 1..n$

$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]_q^p \quad \{\!\{\Delta/\Gamma_j\}\!\}_{\Pi_0} \text{ is defined} \quad (\Pi_0, \sigma_0, b_j\{\!\{\iota/s_j\}\!\}\{\!\{\Delta/\Gamma_j\}\!\}) \Downarrow (\Pi_1, \sigma_1, v)}{(\Pi_0, \sigma_0, \iota.l_j\langle\Delta\rangle) \Downarrow (\Pi_1, \sigma_1, v)}$$

(Subst Update) where $j \in 1..n$

$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]_q^p \quad match(\Gamma_j; \Gamma) \quad \sigma_1 = \sigma_0 + (\iota \mapsto [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..j-1, j+1..n}, l_j = \varsigma(s : A, \Gamma) b]_q^p)}{(\Pi, \sigma_0, \iota.l_j \Leftarrow \varsigma(s : A, \Gamma) b) \Downarrow (\Pi, \sigma_1, \iota)}$$

(Subst Let)

$$\frac{(\Pi_0, \sigma_0, a) \Downarrow (\Pi_1, \sigma_1, v) \quad (\Pi_1, \sigma_1, b\{\!\{v/x\}\!\}) \Downarrow (\Pi_2, \sigma_2, u)}{(\Pi_0, \sigma_0, \mathbf{let}\ x : A = a\ \mathbf{in}\ b) \Downarrow (\Pi_2, \sigma_2, u)}$$

(Subst Unfold)

$$\frac{}{(\Pi, \sigma, \mathbf{unfold}(\mathbf{fold}(A, v))) \Downarrow (\Pi, \sigma, v)}$$

(Subst Expose)

$$\frac{(\Pi_0, \sigma_0, b\{\!\{v/x\}\!\}\{\!\{p/\alpha\}\!\}) \Downarrow (\Pi_1, \sigma_1, u)}{(\Pi_0, \sigma_0, \mathbf{expose}\ (\mathbf{hide}\ p\ \mathbf{as}\ \alpha \prec: p'\ \mathbf{in}\ v{:}A)\ \mathbf{as}\ \alpha \prec: p'', x{:}A'\ \mathbf{in}\ b{:}B) \Downarrow (\Pi_1, \sigma_1, u)}$$

(Subst New)

$$\frac{\alpha' \notin \mathsf{dom}(\Pi_0) \quad \Pi_1 = (\Pi_0, \alpha' \prec: p) \quad (\Pi_1, \sigma_0, a\{\!\{\alpha'/\alpha\}\!\}) \Downarrow (\Pi_2, \sigma_1, v)}{(\Pi_0, \sigma_0, \mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ a) \Downarrow (\Pi_2, \sigma_1, v)}$$

Figure 5.12: Big-step, substitution style semantics

Faulty computations are defined by the relation $(\Pi, \sigma, a) \Downarrow \textsc{Wrong}$ which is specified in Figures 5.13 and 5.14.

The evaluation rules (Subst Value) and (Subst Object) remain as in Section 4.4 with the extended context information carried through.

The rule (Subst Select) now takes a more sophisticated definition of parameter substitution to account for the different kinds of parameters.

Definition 5.6 (Parameter Substitution $\{\!|\Delta/\Gamma|\!\}_E$)

$$
\begin{array}{rcll}
\{\!|\emptyset/\emptyset|\!\}_E & \widehat{=} & \varepsilon & \\
\{\!|v, \Delta/x : A, \Gamma|\!\}_E & \widehat{=} & \{\!|{}^{v}/_{x}|\!\}\{\!|\Delta/\Gamma|\!\}_E & \\
\{\!|B, \Delta/X <: A, \Gamma|\!\}_E & \widehat{=} & \{\!|{}^{B}/_{X}|\!\}\{\!|\Delta/\Gamma\{\!|{}^{B}/_{X}|\!\}|\!\}_E & \\
\{\!|p, \Delta/\alpha \prec: q, \Gamma|\!\}_E & \widehat{=} & \{\!|{}^{p}/_{\alpha}|\!\}\{\!|\Delta/\Gamma\{\!|{}^{p}/_{\alpha}|\!\}|\!\}_E & \textit{where } E \vdash p \prec: q \\
\{\!|p, \Delta/\alpha :\succ q, \Gamma|\!\}_E & \widehat{=} & \{\!|{}^{p}/_{\alpha}|\!\}\{\!|\Delta/\Gamma\{\!|{}^{p}/_{\alpha}|\!\}|\!\}_E & \textit{where } E \vdash q \prec: p
\end{array}
$$

where ε *is the empty substitution. Otherwise* $\{\!|\Delta/\Gamma|\!\}_E$ *is undefined.*

This definition implicitly checks that the correct number of arguments are supplied, that they have the correct kinds, and that the contexts satisfy the bounds on the context parameters. To make this behave properly, the substitution is applied to subsequent bindings in Γ as they may depend on previous ones. This was not present in the system of the previous chapter because it did not matter there, as no checking was performed.

The rules for (Subst Value), (Subst Object), and (Subst Select) are as described before in Section 4.4. The rule (Subst Update) performs an additional check on the formal parameter lists of the existing method and the new method to make sure that the lengths of the parameter lists and kinds of their elements correspond, though it does not check that the bounds do. This is done with the following function.

Definition 5.7 (*match*$(\Gamma;\ \Gamma')$)

$$
\begin{array}{rcl}
\mathrm{match}(\emptyset;\ \emptyset) & \widehat{=} & \mathit{true} \\
\mathrm{match}(x : A, \Gamma;\ y : B, \Gamma') & \widehat{=} & \mathrm{match}(\Gamma;\ \Gamma') \\
\mathrm{match}(X <: A, \Gamma;\ Y <: B, \Gamma') & \widehat{=} & \mathrm{match}(\Gamma;\ \Gamma') \\
\mathrm{match}(\alpha \prec: p, \Gamma;\ \beta \prec: q, \Gamma') & \widehat{=} & \mathrm{match}(\Gamma;\ \Gamma') \\
\mathrm{match}(\alpha :\succ p, \Gamma;\ \beta :\succ q, \Gamma') & \widehat{=} & \mathrm{match}(\Gamma;\ \Gamma') \\
\mathrm{match}(\_;\ \_) & \widehat{=} & \mathit{false}
\end{array}
$$

Evaluation of **let** expressions is as before (Subst Let).

The evaluation rule (Subst Unfold) simply removes the **fold** wrapper from around a value — recall that **fold** and **unfold** are terms which mediate the isomorphism for recursive types.

The evaluation of an **expose** expression, rule (Subst Expose), is straightforward. It takes the hidden context p and value v and substitutes them for the variables α and x in the body b, and then evaluates the result. The presentation has been simplified from the usual rule [3] by using α-conversion to make the bound variables in the **hide** and **expose** expressions the same, following Flanagan and Abadi [80].

The rule (Subst New) is entirely new. It creates a new context inside p. This is bound to fresh variable α' which is both substituted into the term a, and added to the constraint collection Π. The resulting configuration is then evaluated.

As before, evaluation begins with a closed expression a in an empty environment, that is, with the initial configuration $(\emptyset, a, \emptyset)$, where $FV(a) = \emptyset$.

Errors and Error Propagation Evaluation can become stuck when the preconditions associated with an evaluation rule are not satisfied. Programs which result in struck expressions are dubbed *faulty*. Stuck expressions evaluate to the special configuration WRONG. The rules defining faulty expressions are giving in Figure 5.13; rules for propagating the result WRONG through expressions are given in Figure 5.14. These rules account for the following errors:

- (Error Select1) and (Error Update1) occur when a select or update is applied to a value which is not a location;
- (Error Select2) and (Error Update2) occur when the named method does not appear in the object. This is the message-not-understood error;
- (Error Select3) occurs when the parameter substitution is not valid. This means either that the number of actual parameters does not match the number of formal parameters, that the kinds of the actual parameters do not correspond to the expected kind of formal parameter, or that the contexts do not satisfy the bounds declared in the parameter list;
- (Error Update3) occurs when the number or kind of the formal parameters in the updated method does not correspond to those of the existing method;
- (Error Unfold) occurs when attempting to **unfold** a value which is not a **fold**; and

(Error Select1)
$$\frac{v \neq \iota}{(\Pi_0, \sigma_0, v.l_j\langle\Delta\rangle) \Downarrow \textsc{Wrong}}$$

(Error Select2)
$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i^{\,i\in 1..n}]_q^p \quad l \notin \{l_1..l_n\}}{(\Pi_0, \sigma_0, \iota.l\langle\Delta\rangle) \Downarrow \textsc{Wrong}}$$

(Error Select3) where $j \in 1..n$
$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i^{\,i\in 1..n}]_q^p \quad \{\Delta/\Gamma_j\}_{\Pi_0} \text{ is not defined}}{(\Pi_0, \sigma_0, \iota.l_j\langle\Delta\rangle) \Downarrow \textsc{Wrong}}$$

(Error Update1)
$$\frac{v \neq \iota}{(\Pi, \sigma_0, v.l_j \Leftarrow \varsigma(s : A, \Gamma)b) \Downarrow \textsc{Wrong}}$$

(Error Update2)
$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i^{\,i\in 1..n}]_q^p \quad l \notin \{l_1..l_n\}}{(\Pi, \sigma_0, \iota.l \Leftarrow \varsigma(s : A, \Gamma)b) \Downarrow \textsc{Wrong}}$$

(Error Update3) where $j \in 1..n$
$$\frac{\sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i^{\,i\in 1..n}]_q^p \quad \neg match(\Gamma_j; \Gamma)}{(\Pi, \sigma_0, \iota.l_j \Leftarrow \varsigma(s : A, \Gamma)b) \Downarrow \textsc{Wrong}}$$

(Error Unfold)
$$\frac{u \neq \mathbf{fold}(A, v)}{(\Pi, \sigma, \mathbf{unfold}(u)) \Downarrow \textsc{Wrong}}$$

(Error Expose1)
$$\frac{u \neq \mathbf{hide}\ p\ \mathbf{as}\ \beta \prec: p'\ \mathbf{in}\ v{:}A}{(\Pi_0, \sigma_0, \mathbf{expose}\ u\ \mathbf{as}\ \alpha \prec: p'', x{:}A'\ \mathbf{in}\ b{:}B) \Downarrow \textsc{Wrong}}$$

Figure 5.13: Errors

$$\frac{\text{(Error Select4) where } j \in 1..n \quad \sigma_0(\iota) = [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]_q^p \quad \{\!\{\Delta/\Gamma_j\}\!\}_{\Pi_0} \text{ is defined} \quad (\Pi_0, \sigma_0, b_j\{\!\{\iota/s_j\}\!\}\{\!\{\Delta/\Gamma_j\}\!\}) \Downarrow \text{WRONG}}{(\Pi_0, \sigma_0, \iota.l_j\langle\Delta\rangle) \Downarrow \text{WRONG}}$$

$$\frac{\text{(Error Let1)} \quad (\Pi_0, \sigma_0, a) \Downarrow \text{WRONG}}{(\Pi_0, \sigma_0, \textbf{let } x : A = a \textbf{ in } b) \Downarrow \text{WRONG}}$$

$$\frac{\text{(Error Let2)} \quad (\Pi_0, \sigma_0, a) \Downarrow (\Pi_1, \sigma_1, v) \quad (\Pi_1, \sigma_1, b\{\!\{v/x\}\!\}) \Downarrow \text{WRONG}}{(\Pi_0, \sigma_0, \textbf{let } x : A = a \textbf{ in } b) \Downarrow \text{WRONG}}$$

$$\frac{\text{(Error New)} \quad \alpha' \notin \mathsf{dom}(\Pi_0) \quad \Pi_1 = (\Pi_0, \alpha' \prec: p) \quad (\Pi_1, \sigma_0, a\{\!\{\alpha'/\alpha\}\!\}) \Downarrow \text{WRONG}}{(\Pi_0, \sigma_0, \textbf{new } \alpha \lhd p \textbf{ in } a) \Downarrow \text{WRONG}}$$

$$\frac{\text{(Error Expose2)} \quad (\Pi_0, \sigma_0, b\{\!\{v/x\}\!\}\{\!\{p/\alpha\}\!\}) \Downarrow \text{WRONG}}{(\Pi_0, \sigma_0, \textbf{expose } (\textbf{hide } p \textbf{ as } \alpha \prec: p' \textbf{ in } v{:}A) \textbf{ as } \alpha \prec: p'', x{:}A' \textbf{ in } b{:}B) \Downarrow \text{WRONG}}$$

Figure 5.14: Error Propagation

- (Error Expose1) occurs when attempting to **expose** a value which is not a **hide**.

5.6 Key Properties

In this section we present an account of the soundness of the type system, though the intricate details of most proofs are left until Appendix A.

Again soundness depends fundamentally on the following Permissibility Lemma, which states, firstly, that the type of a value contains sufficient information to determine which permissions are required to access values of that type, and secondly, that this information is preserved through subtyping. A consequence is that subtyping cannot be used to violate the containment invariant.

Lemma 5.8 (Permissibility)

1. *If* $E;K \vdash v : A$ *and* $E;K' \vdash A$, *then* $E;K' \vdash v : A$, *where* v *is a value*.

2. *If* $E;K \vdash A<:B$ *and* $E;K' \vdash B$, *then* $E;K' \vdash A<:B$.

PROOF SKETCH: 1. The proof is by induction over the derivation of $E;K \vdash v : A$, based on the structure of value v.
2. A more general result is proven. Firstly, the notion of minimum permission for a type is introduced: a permission $K^\dagger$ is the minimum permission for type B, written $E;K^\dagger \vdash_{\mathsf{min}} B$, if for all K such that $E;K \vdash B$, we have $E \vdash K^\dagger \subseteq K$. In Appendix A we prove that the minimal exists and is unique. The property we prove is that $E;K \vdash A<:B$ and $E;K^\dagger \vdash_{\mathsf{min}} B$ imply that $E;K^\dagger \vdash A<:B$. The desired result can be obtained by suitable application of (Sub Allow). Proof is by induction over the derivation of $E;K \vdash A<:B$. Full details are in Appendix A. □

This lemma may suggest that use of permissions in the typing and subtyping judgements may be redundant or that they can perhaps be inferred, since the permission is a part of the type. Unfortunately attempts to eliminate K from these judgements have failed. In the case of subtyping this seems inevitable because K governs the subtyping relation, rather than the elements which are related. In particular, a type may be a subtype of another for a certain permission, but not for another permission. This requirement seems essentially new; neither Flanagan and Abadi [80] nor Crary, Walker and Morrisett [66] need it. In our case, however, it is essential to prevent substitution and subtyping from interacting in such a way that the containment invariant fails.

Definition 5.9 (Extension) *Environment* E' *is an extension of* E, *written* $E' \gg E$, *if and only* E *is a subsequence of* E'.

Soundness is given in part by the following lemma which states that evaluation preserves typing. The lemma applies only when evaluation results in a value, but not when evaluation diverges or becomes stuck.

Lemma 5.10 (Preservation) *If* $E;K \vdash (\Pi,\sigma,a) : A$ *and* $(\Pi,\sigma,a) \Downarrow (\Pi',\sigma',v)$, *then there exists an environment* E' *such that* $E' \gg E$ *and* $E';K \vdash (\Pi',\sigma',v) : A$.

PROOF SKETCH: By induction over the derivation of $(\Pi,\sigma,a) \Downarrow (\Pi',\sigma',v)$. Full details in Appendix A. □

The following lemma states that faulty programs, those which evaluate to WRONG, are not typable.

Lemma 5.11 (Faulty Programs are Untypable) *If* $(\Pi, \sigma, a) \Downarrow \text{WRONG}$, *then there are no* E, K, A *such that* $E; K \vdash (\Pi, \sigma, a) : A$.

PROOF SKETCH: By induction over the derivation of $(\Pi, \sigma, a) \Downarrow \text{WRONG}$. Full details in Appendix A. □

Together these lemmas state that well-typed configurations either diverge or result in a well-typed final configuration of the expected type. This is the soundness of the type system:

Theorem 5.12 (Soundness) *If* $\emptyset; K \vdash (\emptyset, \emptyset, a) : A$, *then either* $(\emptyset, \emptyset, a)$ *diverges, or* $(\emptyset, \emptyset, a) \Downarrow (\Pi, \sigma, v)$ *and there exists an environment* E *such that* $E; K \vdash (\Pi, \sigma, v) : A$.

PROOF: Follows directly from Lemmas 5.10 and 5.11. □

This presentation style and the underlying proof techniques follow those developed by Wright and Felleisen [198].

We can now demonstrate that the containment invariant holds for well-typed programs.

5.7 The Containment Invariant

Following Section 4.6 of the previous chapter, we provide a model for permissions in terms of contexts and expressions in terms of the owners of locations it accesses, and demonstrate for well-typed expressions that all locations accessed are included within the permission against which the expression was typed. From this we demonstrate that the containment invariant follows. This time, however, we must provide a more complex model of permissions, because contexts are no longer constant and can be created during the evaluation of an expression.

To achieve this we provide a model for all contexts, both constant contexts and the free context variables occurring in expressions. Because the number of contexts and both the depth and breadth of their nesting are unbounded, we chose as the target for our model the set of finite sequences of natural numbers together with the prefix ordering on sequences, denoted $(\mathbb{N}^*, \sqsubseteq)$. We seed the functions with a mapping of the constant contexts $(C, :\succ_C)$ into $(\mathbb{N}^*, \sqsubseteq)$. This is done by the order preserving

embedding: $\mathsf{embed}(\_) : C \to \mathbb{N}^*$. This clearly exists when $(C, :\succ_C)$ is a finite forest. In the case where $(C, :\succ_C)$ has any nontrivial meets, a more complicated model would be required. Doing so does not present any significant additional challenge, so we persist with the simpler model.

The following projection functions map context constants and variables into an element of $\mathbb{N}^*$. These are then used to model locations as the owner of the location, permissions as the contexts which that permission deems accessible, and expressions in terms of the owners of locations accessed within that expression — without looking inside objects or following references. The projection functions are each derived from a function η which provides a model for the locations and context variables that occur free in expressions. The signatures of the projection functions are:

- $\eta : \textsc{ContextVar} \cup \textsc{Location} \to \mathbb{N}^*$
- $[\![\_]\!]_\eta : \textsc{Context} \to \mathbb{N}^*$
- $[\![\_]\!]_\eta : \textsc{Permission} \to \mathbb{P}(\mathbb{N}^*)$
- $[\![\_]\!]_\eta : \textsc{Expression} \to \mathbb{P}(\mathbb{N}^*)$

These functions will be defined shortly.

Before doing so, we need some additional machinery. The following definitions map an element of $\mathbb{N}^*$ to all the elements above or below it. These are called the up- and down-sets for that element. (Note that these are up- and down-sets for the context ordering, not for the prefix ordering in the model.)

Definition 5.13 (Upsets $(\kappa\uparrow)$ and Downsets $(\kappa\downarrow)$)

$$\begin{aligned} \kappa\uparrow &\mathrel{\widehat{=}} \{\kappa' \in \mathbb{N}^* \mid \kappa' \sqsubseteq \kappa\} \\ \kappa\downarrow &\mathrel{\widehat{=}} \{\kappa' \in \mathbb{N}^* \mid \kappa' \sqsupseteq \kappa\} \end{aligned}$$

These are used to construct a model of a typing environment. This is done by giving a value to all context variables in the environment in such a way that the constraints between them are satisfied. When this is the case we write $\eta \models E$, which is defined as:

Definition 5.14 ($\eta \models E$) $\eta \models E$ *whenever*

- $\alpha \prec:$ *p occurs in E, then* $\eta(\alpha) \in [\![p]\!]_\eta\downarrow$,

- $\alpha :\succ p$ *occurs in* E, *then* $\eta(\alpha) \in [\![p]\!]_\eta\uparrow$, *and*
- $\iota : [l_i : \Theta_i^{i\in 1..n}]_q^p$ *occurs in* E, *then* $\eta(\iota) = [\![p]\!]_\eta$.

Note that for a well-typed program, the model for locations corresponds to the function giving their owner. That is, $\eta(\iota)$ corresponds to the owner of object ι.

There is a certain amount of circularity in these definitions, but problems of well-definedness are avoided since environments are constructed inductively. The definition also depends upon the model of contexts, $[\![p]\!]_\eta$, which is defined as:

Definition 5.15 (Model of Contexts)

$$\begin{aligned} [\![\pi]\!]_\eta &\mathrel{\widehat{=}} \mathit{embed}(\pi) \\ [\![\alpha]\!]_\eta &\mathrel{\widehat{=}} \eta(\alpha) \end{aligned}$$

The projection of permissions gives the underlying set of contexts, based on some environment model η:

Definition 5.16 (Projection of Permissions)

$$\begin{aligned} [\![\langle p\rangle]\!]_\eta &\mathrel{\widehat{=}} \{[\![p]\!]_\eta\} \\ [\![\langle p\uparrow\rangle]\!]_\eta &\mathrel{\widehat{=}} [\![p]\!]_\eta\uparrow \\ [\![\bigcup[K_1..K_n]]\!]_\eta &\mathrel{\widehat{=}} \bigcup_{i\in 1..n} [\![K_i]\!]_\eta \end{aligned}$$

Expressions are projected onto the owners of objects appearing in the top-level of the expression, that is, those which are not a part of another object. The projection function is defined in two parts, firstly a function which collects the locations in the top-level of a term, and then the function which projects the locations onto their owners:

Definition 5.17 (Locations in a Term)

$$\begin{aligned} \mathit{locs}(x) &\mathrel{\widehat{=}} \emptyset \\ \mathit{locs}(\iota) &\mathrel{\widehat{=}} \{\iota\} \\ \mathit{locs}(\textbf{\textit{fold}}(A, v)) &\mathrel{\widehat{=}} \mathit{locs}(v) \\ \mathit{locs}(\textbf{\textit{hide}}\ p\ \textbf{\textit{as}}\ \alpha \prec: q\ \textbf{\textit{in}}\ v{:}A) &\mathrel{\widehat{=}} \mathit{locs}(v) \\ \mathit{locs}(o) &\mathrel{\widehat{=}} \emptyset \end{aligned}$$

$$
\begin{array}{rcl}
\mathit{locs}(v.l\langle\Delta\rangle) & \widehat{=} & \mathit{locs}(v) \cup \mathit{locs}(\Delta) \\
\quad \textit{where} & & \mathit{locs}(\emptyset) \;\widehat{=}\; \emptyset \\
& & \mathit{locs}(v,\Delta) \;\widehat{=}\; \mathit{locs}(v) \cup \mathit{locs}(\Delta) \\
& & \mathit{locs}(A,\Delta) \;\widehat{=}\; \mathit{locs}(\Delta) \\
& & \mathit{locs}(p,\Delta) \;\widehat{=}\; \mathit{locs}(\Delta) \\
\mathit{locs}(v.l \Leftarrow \varsigma(s:A,\Gamma)b) & \widehat{=} & \mathit{locs}(v) \cup \mathit{locs}(b) \\
\mathit{locs}(\textbf{\textit{let}}\ x:A = a\ \textbf{\textit{in}}\ b) & \widehat{=} & \mathit{locs}(a) \cup \mathit{locs}(b) \\
\mathit{locs}(\textbf{\textit{unfold}}(v)) & \widehat{=} & \mathit{locs}(v) \\
\mathit{locs}(\textbf{\textit{new}}\ \alpha \lhd p\ \textbf{\textit{in}}\ a) & \widehat{=} & \mathit{locs}(a) \\
\mathit{locs}(\textbf{\textit{expose}}\ v\ \textbf{\textit{as}}\ \alpha \prec: p, x{:}A\ \textbf{\textit{in}}\ b{:}B) & \widehat{=} & \mathit{locs}(v) \cup \mathit{locs}(b)
\end{array}
$$

The model of expressions is achieved simply by applying η to each of the locations collected from an expression. This is equivalent to the collection of owners of those locations:

Definition 5.18 (Projection of Expressions)

$$[\![a]\!]_\eta \;\widehat{=}\; \{\eta(\iota) \mid \iota \in \mathit{locs}(a)\}$$

These definitions culminate in the following lemma which states a few invariants that hold for certain well-formed judgements. It states firstly that the subpermission relation corresponds to subset in the projection of permissions in terms of contexts; secondly, and most importantly, that the owners of locations in a term are included in the projection of the permission, and thirdly, that this is also the case for the arguments passed to a method.

Lemma 5.19 *Assume that* $\eta \models E$. *Then:*

1. *If* $E \vdash K' \subseteq K$, *then* $[\![K']\!]_\eta \subseteq [\![K]\!]_\eta$;
2. *If* $E; K \vdash a : A$, *then* $[\![a]\!]_\eta \subseteq [\![K]\!]_\eta$;
3. *If* $E; K \vdash (\Theta)(\Delta) \Rightarrow C$, *then* $[\![\Delta]\!]_\eta \subseteq [\![K]\!]_\eta$.

PROOF SKETCH: By induction over judgements. Full details are in Appendix A. □

A key aspect of this result is that it holds for any η which provides a model of the appropriate typing environment, including the models which match our intention, namely those where each context is distinct from every other one (where possible).

Again we apply this invariant to a method body with the permission corresponding to the upset of the representation context, and from this deduce that the containment invariant holds. This time, however, we must be a little more careful, since a method body is also typed against a collection of point permissions for each context variable appearing in the formal parameters of that method.

The following definition of well-formed store captures the containment invariant for an entire store:

Definition 5.20 (Well-contained Store)

$$\begin{aligned} \boldsymbol{wf}_\eta(\sigma) &\mathrel{\widehat{=}} \bigwedge_{\iota \mapsto o \in \sigma} \boldsymbol{wf}_\eta(o) \\ \boldsymbol{wf}_\eta([l_i = \varsigma(s_i : A_i, \Gamma_i) b_i^{\,i\in 1..n}]^p_q) &\mathrel{\widehat{=}} \bigwedge_{i\in 1..n} [\![b_i]\!]_\eta \subseteq [\![\langle q\uparrow\rangle]\!]_\eta \end{aligned}$$

Remark 5.21 *We must be wary about the right-hand side of the second clause of this definition. To account for all the contexts visible in a method body it should read something like:*

$$\bigwedge_{i\in 1..n} [\![b_i]\!]_{\eta_i^+} \subseteq [\![\langle q\uparrow\rangle \cup \lceil \Gamma_i \rceil]\!]_{\eta_i^+} \quad \textit{where } \eta_i^+ \models E, s_i : A_i, \Gamma_i$$

where each η_i^+ extends η to model the extended environment used to type each method body. A consequence of the well-formedness of Γ_i is that no context variable in Γ_i can be the owner of a location appearing in the body b_i, since all locations and hence their owners appear further to the left in the typing environment. Removing Γ_i from the definition above, therefore, does not affect its validity. Lemma A.20 in Appendix A accounts formally for this.

Our major result is that well-typed stores are well-contained.

Theorem 5.22 *If $E \vdash \sigma$ then $\boldsymbol{wf}_\eta(\sigma)$ for any η such that $\eta \models E$.*

PROOF SKETCH: Straightforward from Lemma 5.19. □

Again we unravel this to show the validity of the containment invariant. We start with the *refers to* relation, which is identical in form to that of the previous chapter.

Definition 5.23 (*refers to*) *The* refers to *relation, $\rightarrow_\sigma$, for store σ is defined as:*

$$\iota \rightarrow_\sigma \iota' \text{ iff } \iota \mapsto [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i^{\,i\in 1..n}]^p_q \in \sigma \wedge \iota' \in \mathsf{locs}(b_i), \textit{ for some } i \in 1..n$$

The owner and representation contexts for a location are as before: for location-object $\iota \mapsto [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]^p_q \in \sigma$, let $\mathsf{owner}_\sigma(\iota) \mathrel{\widehat{=}} p$ and $\mathsf{rep}_\sigma(\iota) \mathrel{\widehat{=}} q$,

The containment invariant is now straightforward:

Theorem 5.24 (Containment Invariant) *If* $E \vdash \sigma$ *then* $\iota \rightarrow_\sigma \iota' \Rightarrow \mathit{rep}_\sigma(\iota) \prec: \mathit{owner}_\sigma(\iota')$.

PROOF:

1. ASSUME: $E \vdash \sigma, \eta \models E$, and $\iota \rightarrow_\sigma \iota'$.
2. By Lemma 5.22, $\mathbf{wf}_\eta(\sigma)$.
3. Thus $\iota \mapsto o \in \sigma$, where $o \equiv [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]^p_q$ and $\iota' \in \mathsf{locs}(b_i)$ for some $i \in 1..n$
4. Unravelling $E \vdash \sigma$ from step 1 using Definition 5.20, we get $[\![b_i]\!]_\eta \subseteq [\![\langle q\uparrow \rangle]\!]_\eta$, where $\eta \models E$.
5. By Definition, $[\![b_i]\!]_\eta = \{\eta(\iota) \mid \iota \in \mathsf{locs}(b_i)\} = \{\mathsf{owner}_\sigma(\iota) \mid \iota \in \mathsf{locs}(b_i)\}$.
6. Therefore, $\mathsf{owner}_\sigma(\iota') \in [\![\langle q\uparrow \rangle]\!]$, where $q = \mathsf{rep}_\sigma(\iota)$.
7. Thus, unravelling the definition of $[\![\langle q\uparrow \rangle]\!]$ and substituting $\mathsf{rep}_\sigma(\iota)$ for q, we obtain that $\mathsf{rep}_\sigma(\iota) \prec: \mathsf{owner}_\sigma(\iota')$, as required. □

5.8 Conclusion

We have shown the soundness of the type system for quite a sophisticated calculus. The type system enforces the containment invariant where the collection of contexts can be dynamically created during an expression's evaluation. This in turn can be used to model objects where each object has its own representation context. The additional features such as recursive types, context and type parameterised methods give the calculus enough power to model a class-based programming language where classes are generic in both types and the owners of the resulting objects, as we will show in the next chapter.

Obtaining the containment invariant required some nontrivial modifications to the standard type rules. One of our achievements is an elegant set of modifications which result in a calculus as close as possible to Abadi and Cardelli's object calculus. The main technical device for achieving this was permissions which implicitly propagated information essential for the containment invariant through the type rules. In addition, the family of Top types indexed by permissions helped retain elegant treatments of recursive types and unbounded universal quantification.

Now let's put the calculus to work.

Chapter 6

Examples, Encodings, and Extensions

The calculus presented in Chapter 5 is quite expressive. In this chapter we demonstrate some of its power. We describe a number of examples: objects with private representation, aggregates with multiple interfaces, friendly functions which access the representation of different objects, borrowing and so forth. We then describe a class encoding, before continuing with a discussion of how ownership types can be added to a full-featured, class-based programming language such as Java [93]. Rather than present every feature as a part of some detailed encoding, which would obscure more than it would reveal, we describe the class-based language informally, relying on the understanding developed thus far.

Before advancing that far, we note that most of the advantages of the calculus come when certain restrictions are imposed. Indeed, many of the examples presented here rely on these restrictions. The calculus presented in Chapter 5 avoided additional complexity so that we could prove its soundness and containment invariance in a general setting. To obtain additional properties, such as the dominance enjoyed by our original ownership types proposal [61], further restrictions must be imposed.

To this end, we present some additional versions of our calculus that rely on two purely syntactic restrictions which control an object's owner and representation context. These restrictions result in different properties on the object graphs underlying programs. Our original ownership types proposal [61] has both of these restrictions; the general calculus has neither.

Finally, we investigate a new clone operation which the containment invariant warrants. Java's shallow clone clearly breaks encapsulation, whereas Eiffel's deep clone copies too much. The clone operation we present follows from and preserves containment.

6.1 Restricted Calculi

We now present four versions of our calculus. These rely on the presence or absence of the following two properties. The more restricted calculi enforce stronger invariants on the object graphs underlying the programs that can be expressed.

The properties on which our restrictions are based are:

UR: each object has its own unique representation context

DI: an object's representation context is directly inside its owner

The first property requires a linearity of context use in the calculus, namely that the function rep mapping objects to their representation context is 1:1. We achieve this syntactically, rather than by modifying the type system, by requiring that the new context produced by a **new** term is used as the representation context of at most one object.

The second condition ensures that for any object $[l_i = \varsigma(s_i : A_i, \Gamma_i) b_i^{\,i\in 1..n}]^p_\alpha$ the representation context α is directly inside the owner p. This means that the term creating α must be **new** $\alpha \lhd p$ **in** a. This can be achieved syntactically when combined with the previous restriction.

We will name each calculus based on which of these properties the calculus has, using the short titles $\emptyset\varsigma$, **UR**ς, **DI**ς, **URDI**ς, as well as through more descriptive titles. Note that the calculus **DI**ς requires nontrivial modifications to the type system, so we merely mention it here and leave its complete development for another time.

We state without proof a number of propositions about some of our calculi. The proofs involve standard structural induction and are not difficult.

A Syntactic Convenience To simplify the examples, we include objects without a representation context, denoted $[l_i = \varsigma(s_i : A_i, \Gamma_i) b_i^{\,i\in 1..n}]^p$. This can be interpreted as an object which has a representation context that is not used.

6.1.1 The Unrestricted Calculus — $\emptyset\varsigma$

This is the calculus presented in the previous chapter. Neither restriction **UR** nor **DI** applies, thus it lacks the specific properties which the following calculi possess.

Values

$$
\begin{array}{rcl}
u,v & ::= & x \\
& | & \iota \\
& | & \mathbf{fold}(A,v) \\
& | & \mathbf{hide}\ p\ \mathbf{as}\ \alpha \prec: q\ \mathbf{in}\ v{:}A
\end{array}
$$

Terms

$$
\begin{array}{rcl}
a,b & ::= & v \\
& | & [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i\in 1..n}]^p \\
& | & v.l\langle\Delta\rangle \\
& | & v.l \Leftarrow \varsigma(s : A, \Gamma) b \\
& | & \mathbf{let}\ x = a\ \mathbf{in}\ b \\
& | & \mathbf{unfold}(v) \\
& | & \mathbf{expose}\ v\ \mathbf{as}\ \alpha \prec: p, x{:}A\ \mathbf{in}\ b{:}B \\
& | & \mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ \mathbf{let}\ \vec{x} = \vec{a}\ \mathbf{in}\ wfo_\alpha
\end{array}
$$

$$
\begin{array}{rcl}
wfo_\alpha & ::= & [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i\in 1..n}]^p_\alpha \\
& | & \mathbf{fold}(A, wfo_\alpha) \\
& | & \mathbf{hide}\ \alpha\ \mathbf{as}\ \alpha \prec: q\ \mathbf{in}\ wfo_\alpha{:}A
\end{array}
$$

Figure 6.1: **UR**ς — A calculus for objects with unique representation

6.1.2 The Unique Representation Calculus — URς

The **UR**ς calculus (Figure 6.1) uses a simple syntactic restriction to ensure that each object has a unique representation context. The main syntactic difference between **UR**ς and $\emptyset\varsigma$ is the addition of the compound term $\mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ \mathbf{let}\ \vec{x} = \vec{a}\ \mathbf{in}\ wfo_\alpha$ which creates a new context inside p, initialises a collection of local variables $\vec{x}$, potentially creating representation, that is, objects with owner α, before creating an object with that context as a representation. The syntax wfo_α denotes a single object with representation context α, potentially wrapped by a number of **fold**s and **hide**s.[1] The creation of contexts and objects are tied together in such a way that the only context an object can use as its representation context is the one created by the surrounding **new** term, and that is the only place where the context it creates can be used as the representation context. This simple syntactic construct is sufficient to achieve the invariant we desire. Note also that the **hide** part of this syntax relies on α-renaming to avoid problems with renaming context α.

[1] We have also loosened the named form of expressions to simplify the presentation.

Values

$$
\begin{array}{rcl}
u,v & ::= & x \\
 & | & \iota \\
 & | & \mathbf{fold}(A,v) \\
 & | & \mathbf{hide}\ p\ \mathbf{as}\ \alpha \prec: q\ \mathbf{in}\ v{:}A
\end{array}
$$

Terms

$$
\begin{array}{rcl}
a,b & ::= & v \\
 & | & [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i\in 1..n}]^p \\
 & | & v.l\langle\Delta\rangle \\
 & | & v.l \Leftarrow \varsigma(s : A, \Gamma) b \\
 & | & \mathbf{let}\ x = a\ \mathbf{in}\ b \\
 & | & \mathbf{unfold}(v) \\
 & | & \mathbf{expose}\ v\ \mathbf{as}\ \alpha \prec: p, x{:}A\ \mathbf{in}\ b{:}B \\
 & | & \mathbf{new}\ \alpha \lhd p\ \mathbf{in\ let}\ \vec{x} = \vec{a}\ \mathbf{in}\ wfo^p_\alpha \\
 & & \\
wfo^p_\alpha & ::= & [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i\in 1..n}]^p_\alpha \\
 & | & \mathbf{fold}(A, wfo^p_\alpha) \\
 & | & \mathbf{hide}\ \alpha\ \mathbf{as}\ \alpha \prec: q\ \mathbf{in}\ wfo^p_\alpha{:}A
\end{array}
$$

Figure 6.2: **URDI**ς — A calculus for objects with owners-as-dominators

The **UR**ς calculus satisfies the following proposition, recalling from Chapter 5 that rep_σ is the function that gives the representation context for each location in σ.

Proposition 6.1 *If a is a program in **UR**ς and $(\emptyset,\emptyset,a) \Downarrow (\Pi,\sigma,v)$, then rep_σ is a one-to-one function.*

6.1.3 The Owners-as-dominators Calculus — URDIς

This calculus adds the property **DI** to **UR**ς, again purely syntactically. The modified syntax is presented in Figure 6.2. The main difference is that the syntax wfo^p_α carries not only new context α for the representation context, but also the context p which immediately contains α. Again α is used as the representation context of a new object (and only one new object), but this time p must be used as the object's owner context. Consequently, the representation context for every object is directly inside the corresponding owner context.

The properties of this calculus are captured by the following propositions. The

first two state that the calculus does actually satisfy the **UR** and **DI** properties:

Proposition 6.2 *If a is a program in **URDI**ς and $(\emptyset,\emptyset,a) \Downarrow (\Pi,\sigma,v)$, then rep_σ is a one-to-one function.*

Proposition 6.3 *Let a be a program in **URDI**ς. Assume that $(\emptyset,\emptyset,a) \Downarrow (\Pi,\sigma,v)$ and $\iota \mapsto [l_i = \varsigma(s_i : A_i,\Gamma_i)b_i^{\,i\in 1..n}]_q^p \in \sigma$. We have that if $\Pi \vdash q \prec: \alpha$ and $\Pi \vdash \alpha \prec: p$ for some α, then either $\alpha = p$ or $\alpha = q$.*

The owners-as-dominators property requires a bit more effort to state. Recall the refers to relation $\rightarrow_\sigma$. We can define the relation covering access paths as the reflexive and transitive closure of this relation, $\rightarrow_\sigma^*$. That is, $\iota \rightarrow_\sigma^* \iota'$ means there is an access path from ι to ι'. We can capture the notion of "the root of the system" as objects whose owner is ε (or one of the constant contexts). Thus a path from the root of the system to object ι' exists whenever $\mathsf{owner}_\sigma(\iota) = \varepsilon$ and $\iota \rightarrow_\sigma^* \iota'$. Because of the **UR** property, we can define a partial function which gives the owner of an object as $\mathsf{ownerof}(\iota) = \mathsf{rep}^{-1}(\mathsf{owner}(\iota))$. This function is only partial because the owner of some objects may be a constant context.

The key property of the **URDI**ς calculus can now be stated.

Proposition 6.4 *If $\mathsf{owner}_\sigma(\iota) = \varepsilon$ and $\iota \rightarrow_\sigma^* \iota'$, then either $\mathsf{owner}(\iota') = \varepsilon$ or $\iota \rightarrow_\sigma^* \iota'$ can be factored as $\iota \rightarrow_\sigma^* \mathsf{ownerof}(\iota') \rightarrow_\sigma^* \iota'$.*

This is equivalent to stating that all paths from the root of the system to an object pass through that object's owner, or that owners are dominators. The proof adapts the one presented in Appendix B. This property implies that there is a single access point to an object, namely its owner, just as in our original ownership types proposal [61].

6.1.4 The Owners-as-Cutsets Calculus — DIς

There is an additional calculus we consider briefly, namely one exhibiting only property **DI**, that is, that the representation of each object is directly inside its owner. Since the representation context need not be unique per object in this calculus, there may be multiple owner objects, but no iterator like objects. A consequence is that the owner objects, given by the collection $\mathsf{rep}^{-1}(\mathsf{owner}(\iota))$, become a cutset [7] for paths from the root of a system to an object ι.

To specify this calculus, however, requires more than the simple syntactic restrictions used to produce the previous calculi. Indeed, the type system itself needs modification. While it does not seem to be overly difficult, we do not do this because it would further add further complication to an already large calculus. Nevertheless, we will on occasion refer to this hypothetical calculus as **DI**ς.

6.2 Examples

We present a number of examples using the calculi just described. Our examples cover a range of possibilities: objects with their own protected representation; simple objects sharing a protected representation; linked data structures which are entirely representation; aggregate objects with multiple interfaces, using sharing and not; the initialisation of objects with externally created representation; the direct access of representation, its use in friendly functions, and some encodings to prevent it; borrowing; garbage collection; and aggregate objects which can be allocated and deleted in a stack-based manner.

When an example relies on a specific property of one of our calculi, we use the appropriate calculus and discuss the example's properties. Most of the time the absence of a particular property is due to what we call *vampiric behaviour*. This occurs when an object obtains an indirect reference to another object's representation. Such action is allowed in $\emptyset\varsigma$, because that calculus imposes a constraint only over how objects fit together, not how they came to be together. The other calculi, especially **URDI**ς, avoid this to varying degrees. Thus before presenting any examples, we will describe the various forms of vampiric behaviour.

Notation: In the examples which follow we do not follow the syntax of expressions exactly. We will sometimes omit annotations and subexpressions which are unnecessary for the discussion. In addition, we introduce a special derived term which significantly simplifies the presentation.

A derived term: newhide A number of the examples employ a common pattern which is syntactically rather heavy. An example of this is (omitting the type annota-

tion on **hide**):

$$\begin{array}{l}\textbf{new}\ \alpha \lhd p\ \textbf{in}\\ \quad \textbf{let}\ \mathit{engine} = [\cdots]^{\alpha}\ \textbf{in}\\ \quad\quad \textbf{let}\ \mathit{car} = [\mathit{engine} = \mathit{engine}, \cdots]^{p}_{\alpha}\ \textbf{in}\\ \quad\quad\quad \textbf{hide}\ \alpha\ \textbf{as}\ \alpha \prec: p\ \textbf{in}\ \mathit{car}\end{array}$$

This expression creates a *car* object, which has representation context α, before hiding this context. Its representation, an *engine* is also initialised. This expression is equivalent to

$$\begin{array}{l}\textbf{new}\ \alpha \lhd p\ \textbf{in}\\ \quad \textbf{let}\ \mathit{car} =\\ \quad\quad \textbf{let}\ \mathit{engine} = [\cdots]^{\alpha}\ \textbf{in}\\ \quad\quad\quad [\mathit{engine} = \mathit{engine}, \cdots]^{p}_{\alpha}\ \textbf{in}\\ \quad\quad\quad\quad \textbf{hide}\ \alpha\ \textbf{as}\ \alpha \prec: p\ \textbf{in}\ \mathit{car}\end{array}$$

Now we introduce the derived term which couples the **new** and the **hide** together:

$$\textbf{newhide}\ \alpha \lhd p\ \textbf{in}\ a : A \quad \widehat{=} \quad \textbf{new}\ \alpha \lhd p\ \textbf{in let}\ x = a\ \textbf{in hide}\ \alpha\ \textbf{as}\ \alpha \prec: p\ \textbf{in}\ x{:}A$$

where $x \notin FV(a)$.

Using the derived term we can simplify the example above to

$$\begin{array}{l}\textbf{newhide}\ \alpha \lhd p\ \textbf{in}\\ \quad \textbf{let}\ \mathit{engine} = [\cdots]^{\alpha}\ \textbf{in}\\ \quad\quad [\mathit{engine} = \mathit{engine}, \cdots]^{p}_{\alpha}\end{array}$$

The typing rule for **newhide** is easily derived:

(Val Newhide)

$$\frac{E, \alpha \prec: p; K \cup \langle\alpha\rangle \vdash a : A \quad E; K \vdash \exists(\alpha \prec: p)A}{E; K \vdash \textbf{newhide}\ \alpha \lhd p\ \textbf{in}\ a : A : \exists(\alpha \prec: p)A}$$

Note that all the above calculi can be massaged to use this term, though the reverse is not true when the new context α generated by **newhide** is used more than once.

The **newhide** term seems to correspond closely to the term which would underly Gabbay and Pitts' fresh name quantifier [83]. However the presence of **hide** in our syntax means that **newhide** does not satisfy the semantics of the fresh name quantifier. Nevertheless, the typing rules for **newhide** are very close to those of the fresh name quantifier, modulo the conditions we have added to deal with containment.

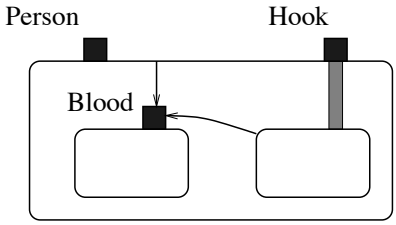


Figure 6.3: Additional object *Hook* accessing representation

6.2.1 Vampires

The type system of the unrestricted calculus $\emptyset\varsigma$ imposes a structural restriction on object graphs as they appear in the store. It says less about how an object graph was produced. More precisely, the containment invariant allows multiple objects to access the contents of a given representation context, but it says little about which objects created those objects. Now there are three ways to create an object which accesses another's representation: internally within the holder of the representation's method; externally using **expose**; and when the representation context is passed to a context-polymorphic method. We present an example of each of these which produces the collection of objects shown in Figure 6.3. This figure shows a *Person* object which has the object *Blood* as a part of its representation. Each of our examples creates the object labelled *Hook* which is a *proxy object* that can access the representation but is itself accessible outside of *Person*.

Internal creation In the first example, the principal object *Person* creates the *Hook* object in one of its methods, as follows:

$$
\begin{array}{rcl}
\mathit{person} & : & \exists(\alpha \prec: \varepsilon)A \quad \text{where } A \equiv [\mathit{blood} : [\cdots]^{\alpha}, \ldots] \\
\mathit{person} & \widehat{=} & \textbf{newhide } \alpha \prec: \varepsilon \textbf{ in} \\
 & & \quad [\mathit{blood} = \cdots, \\
 & & \quad \mathit{newhook} = \\
 & & \quad\quad \varsigma(x : A)\textbf{newhide } \beta \prec: \alpha \textbf{ in} \\
 & & \quad\quad\quad \textbf{let } b = x.\mathit{blood} \textbf{ in } [\mathit{drain} = \ldots \mathit{drink}(b), \ldots]^{\varepsilon}_{\beta}\ ]^{\varepsilon}_{\alpha} \\
\mathit{hook} & \widehat{=} & \mathit{person}_{\circ}\mathit{newhook}
\end{array}
$$

where $x_{\circ}l \mathrel{\widehat{=}} \textbf{expose } x \textbf{ as } \alpha \prec: \varepsilon, y : A \textbf{ in } x.l$ is an encoding of safe external method selection (see Section 6.2.9).

The *hook* object has access to the *person*'s representation context, because *hook*'s representation context β is inside *person*'s. Hence it has access to the object *blood*. This is how iterators are created.

We deem this example to be "safe" because the *hook* object is defined and created within the *person* object, which is the owner of the representation. However, it is possible to construct the same collection of objects by creating the *hook* object from outside the *person* object using **expose**. We show this next.

External creation via expose The example above can be modified slightly so that the *hook* object is created externally to the *person* object, by first **exposing** the representation context. While this difference may appear superficial, it is undesirable because it has moved the creation of the proxy object away from owner of the representation to a client object.

$$
\begin{array}{rcl}
\mathit{person} & \widehat{=} & \textbf{newhide } \alpha \prec: \varepsilon \textbf{ in } [\mathit{blood} = \cdots, \ldots]^{\varepsilon}_{\alpha} \\
\mathit{hook} & \widehat{=} & \textbf{expose } \mathit{person} \textbf{ as } \alpha \prec: \varepsilon, x : A \textbf{ in} \\
 & & \quad \textbf{newhide } \beta \prec: \alpha \textbf{ in let } b = x.\mathit{blood} \textbf{ in } [\mathit{drain} = \ldots \mathit{drink}(b), \ldots]^{\varepsilon}_{\beta}
\end{array}
$$

We call objects such as *hook* a *vampiric proxy object*, or *vampire*, and refer to behaviour such as opening the protection boundary around an object and attaching a proxy object to its representation as *vampiric behaviour*. Vampires create the kinds of aliases we are trying to avoid, indirectly through a proxy object, although they cannot expose the representation directly outside of the proxy.

This example can be excluded by limiting the kinds of expressions that appear in the right-hand side of an **expose** expression, for example, by restricting to just method select and update. (We do this in Section 6.2.9.) The problem, from a modelling perspective, is that we also use **expose** for friendly functions so a trade-off between expressiveness and the level of protection must be made. The behaviour can be alleviated to some degree by using subsumption to remove the fields and methods which have the representation context in their type from the object's interface, in effect, making them private [60].

Another form of vampiric behaviour can occur within a context polymorphic method.

External creation via a context polymorphic method In the first example the *hook* object was specified within the *person* object. It may be, however, that the *hook* object

is created externally, for example, using a class constructor. The constructor would need to know which context to set as the owner of the *hook* object, and thus it would need to be context polymorphic. Unfortunately, this idiom essential to classes is susceptible to vampiric behaviour, because it makes the representation context available.

The following example demonstrates how a vampiric proxy object may be created. This time the vampire is stored in field *hook* whenever the *sethook* method is called.

$$
\begin{array}{rcl}
\mathit{trap} & \widehat{=} & [\mathit{hook} = \varsigma(s)s.\mathit{hook}, \\
& & \quad \mathit{sethook} = \varsigma(s, \alpha \prec: \varepsilon, x : [\cdots]^{\alpha}) \\
& & \qquad \textbf{let } y = \textbf{newhide } \beta \prec: \alpha \textbf{ in} \\
& & \qquad\quad [\mathit{drain} = \ldots \mathit{drink}(x), \ldots]^{\varepsilon}_{\beta} \textbf{ in} \\
& & \qquad s.\mathit{trap} \Leftarrow \varsigma(\_)y \ ]^{\varepsilon} \\
\mathit{person} & \widehat{=} & \textbf{newhide } \alpha \prec: \varepsilon \textbf{ in} \\
& & \qquad [\mathit{blood} = \cdots, \\
& & \qquad \mathit{fall\_into\_trap} = \varsigma(\mathit{self} : A)\mathit{trap}.\mathit{sethook}\langle \alpha, \mathit{self}.\mathit{blood} \rangle, \ldots]^{\varepsilon}_{\alpha} \\
& & \quad \mathit{person}_{\circ}\mathit{fall\_into\_trap}\langle\rangle
\end{array}
$$

Of course, access to the representation objects is required as well as access to the representation context.

Because context parameters are essential for classes (and a few other idioms described in this chapter), they cannot simply be eliminated to control vampiric behaviour.

Variations A number of variations on these examples are possible. Firstly, rather than create a new representation context, the proxy object could share the existing representation context. (The calculi **UR**ς and **URDI**ς prevent this.) Alternatively, rather than accessing just the representation object *blood*, a reference to *person* (without its **hide** wrapper) could be retained. In this case, the current object *blood* can always be accessed, rather then the *blood* object which existed when the proxy object is created.

Comments It is first worth emphasising that none of these can be done in the **URDI**ς calculus, because each representation context must be directly inside the corresponding owner context and representation contexts cannot be shared between objects. This means that an object such as *hook* cannot exist. Thus vampiric proxy objects are not a problem in the **URDI**ς calculus, though we of course also lose the ability to create objects with external iterators.

From a modelling perspective, we would hope that a single abstraction creates the proxy objects so that their implementations are known, understood, and trusted enough to access the representation, and can thus be reasoned about. Unfortunately, the calculus $\emptyset\varsigma$ neither specifies nor constrains the creation of proxy objects such as the one described here. This was a deliberate choice to keep the type system simple in order to demonstrate its soundness in a general setting. Ultimately, the $\emptyset\varsigma$ calculus provides the mechanism for creating such objects; additional means are required to provide the policy. The calculus **URDI**ς provides one extreme, excluding proxy object entirely, since it excludes all forms of vampiric behaviour. The encodings we present in Section 6.2.9 eliminate the form based on **expose**. Class-based languages (discussed in Section 6.3) place better control over proxy objects, because they make it clearer when vampires are possible and when vampires can be excluded.

Resolving this issue more satisfactorily has been slated for future work. (We have done some work in this direction which is too preliminary to be included in this thesis.)

6.2.2 Protected Objects and Context Creation

Our first example shows how context creation interacts with hiding to encode protected objects. The expression **new** creates a new context which can be (or must be for calculi **UR**ς and **URDI**ς) used as the representation context of a new object. This context can then be hidden using **hide**. The new context is available only in the body of the **new** expression, and can therefore be used to initialise the new object's fields with representation. In fact, the restricted syntax of **UR**ς and **URDI**ς has been designed to support exactly this behaviour. Consider the example (presented both fully expanded and using **newhide**):

$$\begin{array}{l}\textbf{new}\ \alpha \lhd p\ \textbf{in}\\ \quad \textbf{let}\ \mathit{engine} = [\cdots]^{\alpha}\ \textbf{in}\\ \quad\quad \textbf{let}\ \mathit{car} = [\mathit{engine} = \mathit{engine}, \cdots]^{p}_{\alpha}\ \textbf{in}\\ \quad\quad\quad \textbf{hide}\ \alpha\ \textbf{as}\ \alpha \prec: p\ \textbf{in}\ \mathit{car}\end{array} \equiv \begin{array}{l}\textbf{newhide}\ \alpha \lhd p\ \textbf{in}\\ \quad \textbf{let}\ \mathit{engine} = [\cdots]^{\alpha}\ \textbf{in}\\ \quad\quad [\mathit{engine} = \mathit{engine}, \cdots]^{p}_{\alpha}\end{array}$$

The **new** construct creates a new context just inside p substitutes it for α in the remainder of the expression. An new *engine* object is then created with the new context α as its owner. Hence the *engine* will become part of the representation of the resulting *car* object. Finally, the actual representation context α is hidden, resulting in an expression of type $\exists(\alpha \prec: p)A$, where A is the type of *car*. The hidden context α does not appear free in the type $\exists(\alpha \prec: p)A$, following the usual rule for existential

type introduction (and a trick with α-conversion). The expression using **newhide** is more convenient to use, but its behaviour is exactly the same.

The resulting object can be called a *protected object* because its representation context is hidden. The engine is accessible only using an **expose** expression, and this access is limited because no other object can manufacture the necessary context. The exact extent of the protection depends on whether the calculus in question allows vampiric behaviour. In **URDI**ς the representation is very well-protected, since this calculus satisfies the owners-as-dominators property, and vampiric behaviour is impossible. Of course the **expose** term can be used to access the representation directly, as we shall see below, but dangerous use of this term can be eliminated using suitable encodings. In **UR**ς no other object will have the same representation context, but vampiric behaviour is still possible. In **DI**ς only sharing style vampiric behaviour is possible, because of the owners-as-cutsets property. And of course in the calculus $\emptyset\varsigma$ full vampiric behaviour is possible, so the actual protection a protected object enjoys is limited.

6.2.3 Linked Data Structures

One of the virtues of ownership types is that entire linked data structures can be a part of an object's representation, where every link in the data structure receives the same amount of protection. We illustrate this through a simple object-oriented, linked list data structure. For this example we assume that there is a value NULL in the calculus.

A linked list consists of a handle object which contains at least a reference to a link object, which is the first element of the list, or NULL if the list is empty. Each link object contains a reference to some data, that is, an element of the list, and another link, or NULL for the end of the list. We require that the links of the linked list are part of the representation of the list. This is achieved by giving them the same owner, which is the protected representation context of the handle object.

Assume that we have some type D which is the type of our data. Note that this is an externally accessible type because the data is not a part of the representation. Our links have type:

$$\text{LINK}^{\alpha} \;\widehat{=}\; \mu(Y)[next : Y, data : \text{D}]^{\alpha}$$

where the owner α will be the representation context of the list. It is a recursive type because the *next* field has the same type, LINK^{α}, including the same owner.

The list has the following type:

$$\exists(\alpha \prec: p)\text{LIST}_\alpha^p$$
$$\text{where } \text{LIST}_\beta^\alpha \mathrel{\widehat{=}} \mu(X)[\mathit{head} : \text{LINK}^\beta, \mathit{addfront} : \text{D} \to X]_\beta^\alpha$$

The following is an empty list object:

$$\begin{array}{l}\mathit{list} : \exists(\alpha \prec: p)\text{LIST}_\alpha^p \mathrel{\widehat{=}} \\ \quad \textbf{newhide } \alpha \lhd p \textbf{ in} \\ \quad\quad \textbf{fold}(\text{LIST}_\alpha^p, \\ \quad\quad\quad [\mathit{head} = \text{NULL}, \\ \quad\quad\quad\ \mathit{addfront} = \varsigma(l :, d : \text{D}) \\ \quad\quad\quad\quad l.\mathit{head} \Leftarrow \varsigma()\textbf{fold}(\text{LINK}^\alpha, [\mathit{next} = l.\mathit{head}, \mathit{data} = d]^\alpha) \\ \quad\quad\quad]_\alpha^p)\end{array}$$

The **fold**s are used to mediate between the different forms of recursive types for the list and links.

The list can be used as follows, assuming that D is *Integer* for this example:

$$\begin{array}{l}\textbf{expose } \mathit{list} \textbf{ as } \alpha \prec: p, x : \text{LIST}_\alpha^p \textbf{ in} \\ \quad \textbf{let } y = \textbf{unfold}(x) \textbf{ in} \\ \quad\quad \textbf{let } z = y.\mathit{addfront}\langle 10\rangle.\mathit{addfront}\langle 15\rangle.\mathit{addfront}\langle 25\rangle \textbf{ in} \\ \quad\quad\quad \textbf{hide } \alpha \textbf{ as } \alpha \prec: p \textbf{ in } \textbf{fold}(\text{LIST}_\alpha^p, z)\end{array}$$

The result corresponds to the list $25, 15, 10$.

6.2.4 Simple Objects Sharing Representation

We can model two objects sharing the same representation by creating a private representation context for the two objects to share. Here is a simple example of three objects which represent a husband, a wife and their shared car (packed together in a fourth object):

$$\begin{array}{l}\textbf{newhide } \mathit{ours} \lhd p \textbf{ in} \\ \quad \textbf{let } \mathit{car} = [\mathit{engine} = \cdots]^{\mathit{ours}} \textbf{ in} \\ \quad\quad [\mathit{husband} = [\mathit{car} = \mathit{car}, \mathit{books} = [\cdots]^{bc}]_{\mathit{ours}}^p, \\ \quad\quad\ \mathit{wife} = [\mathit{car} = \mathit{car}, \mathit{CDs} = [\cdots]^{cdc}]_{\mathit{ours}}^p\ ]^p\end{array}$$

The *husband* and *wife* have the same representation context *ours*. This is the owner of the *car*, indicating that the *husband* and the *wife* both own the *car*. References to the *car* can only be held by objects with access to the context *ours*, so the level of protection depends on the accessibility of that context.

In addition, the husband is in a book club which has books in context *bc*, and the wife is in a music club which has CDs in context *cdc*. The *husband* can share his *books* with anyone who has access to the book club context *bc*. Similarly, the *wife* can share her *CDs* with anyone who has access to the music club context *cdc*.

More complex examples are possible in the $\mathbf{DI}\varsigma$ and $\emptyset\varsigma$ calculi, particularly by using more dynamic means for creating objects with the same representation context, and by using **expose**.

6.2.5 Aggregate Objects with Multiple Interfaces

Aggregate objects often present what can be viewed as multiple interfaces, a principal interface and several alternatives, each of which has access to the aggregate's representation. Examples include external iterators to container data structures, where the handle object presents a principal interface for adding and deleting elements to and from the container, and the iterator presents an alternative interface for traversing the elements of the container.

The example we give models a *car* abstraction. Two interfaces are presented to its clients: the principal interface is the driver's; the other, less frequently used one, is for mechanics to tune the car's engine.

There are two ways to model this aggregate. Firstly, we can use sharing, adapting the example above. In the second approach, we allow the additional interface to have its own representation and thus its own private implementation.

Using Sharing Consider the following:

$$
\begin{array}{l}
\mathit{mycar} = \ \mathbf{newhide}\ \alpha \lhd p\ \mathbf{in} \\
\quad \mathbf{let}\ \mathit{engine} = [\cdots, \mathit{tamper} = \cdots]^{\alpha}\ \mathbf{in} \\
\qquad [\mathit{engine} = \varsigma()\mathit{engine}, \\
\qquad\ \mathit{tune} = \varsigma(s)\mathbf{let}\ t = [\ldots s.\mathit{engine}.\mathit{tamper}\ldots]^{p}_{\alpha}\ \mathbf{in\ hide}\ \alpha\ \mathbf{as}\ \alpha \prec: p\ \mathbf{in}\ t \\
\qquad]^{p}_{\alpha}
\end{array}
$$

The *mycar* object itself is the principal, and the *engine* is a part of its representation. Each time the *tune* method is invoked a new alternative interface is created.

This is a separate object that shares the representation context with the principal object, which means that this object has permission to access *mycar*'s representation and tamper with its *engine*. Furthermore, since both the principal object and the alternative interfaces have the same owner, they can be accessed by the same objects.

This example is possible in the **DI**ς and $\emptyset\varsigma$ calculi.

Without Sharing An alternative approach is to give the objects created by the *tune* method their own representation context, and thus the ability to have their own protected representation. This style of encoding is necessary to do this example in the **UR**ς calculus. The example now becomes:

$$
\begin{array}{l}
\mathit{mycar} = \ \textbf{newhide}\ \alpha \lhd p\ \textbf{in} \\
\quad \textbf{let}\ \mathit{engine} = [\cdots, \mathit{tamper} = \cdots]^{\alpha}\ \textbf{in} \\
\qquad [\mathit{engine} = \varsigma()\mathit{engine}, \\
\qquad\ \mathit{tune} = \varsigma(s)\textbf{newhide}\ \beta \lhd \alpha\ \textbf{in}\ [\ldots s.\mathit{engine}.\mathit{tamper}\ldots]^{p}_{\beta} \\
\qquad]^{p}_{\alpha}
\end{array}
$$

The idea is to create the representation context of the alternative interface objects inside that of the principal, and give both the principal object and the alternative interfaces the same owner. Again *mycar* is the principal object, and the method *tune* creates alternative interfaces. The representation context of these new objects is a new context β which is inside the representation context of the *mycar* object. Consequently the new object has permission to tamper with the *engine*, but is also allowed to have its own representation.

This example cannot be done in the **URDI**ς calculus.

6.2.6 Initialisation

Detlefs, Leino, and Nelson [67] discuss the problem of initialising the representation of newly created objects. The example they describe is a lexical analyser object (*lexer*) that takes its input from a stream reader object (*reader*), which is to be a part of the *lexer*'s representation. The twist is that the *reader* must be externally created/specified. Their problem was to guarantee that no external aliases to the *reader* remain after the *lexer* is initialised. We can guarantee this in the **URDI**ς calculus.

Various coding idioms allow the *reader* to be created independently of the *lexer*. A simple one follows; it uses a class *readerClass* to construct the *reader*. The *new* method of this class takes a context parameter for the owner of the new *reader*.

The representation context of the yet-to-be created *lexer* is passed to this constructor method, as follows:

$$\begin{array}{rcl} \mathit{readerClass} & \widehat{=} & [\mathit{new} = \varsigma(\_, \alpha :\succ \varepsilon)[\cdots \text{reader implementation} \cdots]^{\alpha}]^{\varepsilon} \\ \mathit{lexer} & \widehat{=} & \textbf{newhide } \alpha \lhd p \textbf{ in} \\ & & \quad \textbf{let } \mathit{newreader} = \mathit{readerClass}.\mathit{new}\langle\alpha\rangle \textbf{ in } - \text{here} \\ & & \quad\quad [\mathit{reader} = \varsigma()\mathit{newreader}, \cdots]^{p}_{\alpha} \end{array}$$

Another approach could use **expose** to access the representation context to create a *reader* with the correct owner.

In all cases the **URDI**ς calculus guarantees that no references to that can be retained outside of the new object because vampiric behaviour is disallowed.

6.2.7 Direct Access to Representation

The term **expose** is a part of the type-theoretic machinery which allows direct access to an object's representation context, and hence its representation, so that the object's methods can be accessed. The scope of the exposure is limited, but in calculi which allow vampiric behaviour, the scoping can be subverted by a vampiric proxy object. This is unsatisfactory in general, but the **expose** term allows, for example, friendly functions which access the representation of different objects without confusing their different representation contexts. We will describe friendly functions next, before giving a number of encodings which can be used to prevent direct access to the representation.

6.2.8 Friendly Functions

A friendly function is a function or method which can access the private parts of more than one object [72]. In the present context, a friendly function can access the private representation of more than one object. Representation containment requires that the representation contexts of different objects must not be confused, ensuring, for example, that the different representation objects are not swapped.

Dealing with friendly functions in the presence of representation containment is one of the limitations of most existing alias management schemes. They were impossible in our previous work [61], because there was no way to distinguish two different representation contexts. The solution Universes adopts restricts access to all but one party to being read-only [144], but this is potentially too restrictive. Neither Balloons

[10] nor Islands [109] mention friendly functions. Friendly functions also present a number of challenges for typing in general [158, for example].

Friendly functions can be encoded in our calculus by using **expose** statements to access the representation context of each member of the friendship, as illustrated by the following brief example. In this example the representation contexts of two engines are accessed in order to compare the rate of their exhaust emissions:

$$
\begin{array}{l}
[\mathit{analyse} = \varsigma(s : A, \mathit{car1} : \exists(\alpha \prec: p)\mathit{Car}, \mathit{car2} : \exists(\beta \prec: p)\mathit{Car'}) \\
\quad \textbf{expose}\ \mathit{car1}\ \textbf{as}\ \alpha \prec: p, c1 : \mathit{Car}\ \textbf{in} \\
\quad \textbf{expose}\ \mathit{car2}\ \textbf{as}\ \beta \prec: p, c2 : \mathit{Car'}\ \textbf{in} \\
\quad\quad c1.\mathit{engine}.\mathit{exhaust} > c2.\mathit{engine}.\mathit{exhaust}]^{\varepsilon}
\end{array}
$$

The representation contexts cannot be confused because they are bound to different variables, α and β, and thus the types of the representation will differ in their owner parameters when typing the body of the method *analyse*. Because the types are syntactically different, they are incompatible.

Other examples of friendly functions include matrix operations such as addition and multiplication which require direct access to the underlying arrays for efficiency reasons.

One further issue for friendly functions concerns the guarantees that can be made about the members of the friendship. Again in the more general calculi, friendly functions are susceptible to vampiric action, because the **expose** term reveals the representation contexts. The **URDI**ς calculus avoids this problem. Fortunately it is impossible in any of our calculi to create a vampiric proxy object which retains access to the representation of multiple different objects from the friendship.

6.2.9 Safe External Method Selection and Update

In most cases direct access to the representation is undesirable because it subverts the object's intended interface to the representation, opening the possibility for the violation of the object's invariants. The fact that the representation is accessible in the first place is very much an artifact of the type-theoretic machinery. Such direct access was successfully avoided in our Java-based ownership types system [61]. There we used a simple side condition in the appropriate type rules which prevented external access to types indicated as being representation.

To achieve the same effect here, we encode *safe external method selection* and *update* which are applicable only to methods which do not have the representation

context in their type. They work simply by exposing the object and selecting or updating the appropriate method, and then repacking the object if necessary. The type system guarantees that methods whose type includes the representation context are not accessible. The safety of the calculus can then be enhanced by removing the **expose** term from the programmer's regular syntax and using just these encodings. Then, if one wished to directly access representation, the use of the **expose** term would signal potentially unsafe behaviour. The encoding further supports the distinction between the external and internal view of an object, which should be different. Fewer features should be available externally.

Now let's look at the encodings. Assume that v is a value corresponding to a protected object of type $\exists(\alpha \prec: q)A$, where $A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_\alpha$ is the type of the unprotected object. Assume also that B is the return type of method l, and that α is a fresh variable which will act as the placeholder for the unknown representation context. Safe external method select and update are given by the following:

$$\begin{aligned} v_\circ l\langle\Delta\rangle &\;\widehat{=}\; \textbf{expose } v \textbf{ as } \alpha \prec: q, x{:}A \textbf{ in } x.l\langle\Delta\rangle{:}B \\ v_\circ l \Leftarrow \varsigma(s:A,\Gamma)b &\;\widehat{=}\; \textbf{expose } v \textbf{ as } \alpha \prec: q, x{:}A \textbf{ in} \\ &\qquad x.l \Leftarrow \varsigma(s:A,\Gamma)b \;;\; v : \exists(\alpha \prec: q)A \end{aligned}$$

In both cases the type annotation on the right-hand side is not necessary on the left-hand side, because they can both be determined from context — the return type of the method in the first instance, and the type of the value v in the latter. Observe also that the method update encoding returns the original object, using the sequence operator ";" (which can easily be encoded using **let**), rather than repacking the result of the method update using **hide**. The result is the same.

The derived type rule for external method selection is:

(Val External Method Selection) (where $A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_\alpha$)

$$\frac{E;K \vdash v : \exists(\alpha \prec: p)A \quad E;K \vdash (\Theta_j)(\Delta) \Rightarrow C_j \quad E;K \vdash \Theta_j}{E;K \vdash v_\circ l_j\langle\Delta\rangle : C_j}$$

This rule is as we would expect. We can deduce from $E;K \vdash \Theta_j$ that the representation context (the bound variable α) does not appear in the method type Θ_j, and thus this rule only applied to methods which do not have the representation context in their type.

The derived type rules for external method update is:

$$\frac{\begin{array}{c}\text{(Val External Method Udpate) (where } A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_\alpha \text{ and } \Theta_j \equiv \lfloor \Gamma \rfloor_{C_j}) \\ E;K \vdash v : \exists(\alpha \prec: q)A \quad E,\alpha \prec: q,s : A,\Gamma;K' \vdash b : C_j \quad E;K \vdash \Theta_j \\ E,\Gamma \vdash K' \subseteq \langle q\uparrow\rangle \cup \lceil \Gamma \rceil \quad E,\Gamma \vdash K' \subseteq K \cup \lceil \Gamma \rceil \quad j \in 1..n\end{array}}{E;K \vdash v_\circ l \Leftarrow \varsigma(s : A,\Gamma)b : \exists(\alpha \prec: q)A}$$

Again this type rule does not permit access to a method whose type contains the representation context. This rule is much more limiting that the usual version of method update, because the permission for typing the new method is limited by $\langle q\uparrow\rangle$ where q is outside of the representation context. Nevertheless, the type rule is admissible. The limitation causes no problem when the method being updated is really a field, because sufficient permission would be given in the surrounding expression.

Note that for these encodings we have made the assumption that protected objects are just the **hide** of an object. If the object is also wrapped by a **fold**, then an **unfold** would also be needed with the encoding. The required modifications are not difficult.

By adopting these encodings in the more expressive calculi, **UR**ς and $\emptyset\varsigma$, we can prevent the kind of vampiric behaviour which uses **expose**.

We now describe two potential benefits our calculus has for memory management.

6.2.10 Garbage Collection

The key property of the owners-as-dominators calculus **URDI**ς is that all access paths to the representation of an aggregate object from its outside must pass through the owner of that representation. Intuitively then, when there are no external references to an owner object, then there can be no external paths into its representation. Thus it is safe to delete the entire aggregate object — the owner and its interior — whenever the owner becomes detached from the root of the system.

Interestingly, this creates a surprising duality with the key behaviour of the regions calculus [188], where values are stored in regions of memory which are allocated and deallocated in a stack-based manner. In the regions calculus, the *lifetime of objects depends on the lifetime of regions*. In the **URDI**ς calculus, when we consider a context to be a region, we observe the dual, namely that *the lifetime of regions depends on the lifetime of objects*. Unfortunately, we have not investigated this in any depth, nor proven any of its safety properties, though the underlying invariants seem clear.

The practical benefits of these observations about objects remains to be determined. One clear benefit, though, is that if we have a construct which allocates stack-based regions, then we can allocate and deallocate entire aggregate objects in a stack based manner. We discuss this next.

6.2.11 Stack-based Object Allocation

We can provide a means for creating contexts whose life-times are restricted to a particular expression by reinstating the term **new** $\alpha \lhd p$ **in** a into the **URDI**ς calculus. Call the result **URDI**ς^+. Implicit in this is that a is not an object with representation context α, since this is enforced by the original syntax of **URDI**ς.

The term **new** $\alpha \lhd p$ **in** a in **URDI**ς^+ corresponds closely to the **letregion** term from the regions calculus [188]. The term **letregion** ρ **in** e creates a new region of memory named ρ and then evaluates e. Values created while evaluating e can be placed in region ρ. When the expression e has finished evaluating, the region ρ and its contents can be safely deallocated. The type system guarantees that the region can be deleted safely. The expression e may involve further **letregion** terms, which allows regions to be allocated and deallocated in a stack-based manner due to the lexical nesting of **letregion** constructs.

We will now compare how **new** behaves with the **letregion** construct.

The context α created by **new** $\alpha \lhd p$ **in** a cannot be used as the representation context of an object, since the restricted syntax of **URDI**ς^+ has sole control over the creation and use of representation contexts. Thus if α is used at all it must be as the owner of an object. Contexts can be created inside α and used as the representation context of objects, allowing arbitrary object graphs within context α. Now the type rule for **new** forbids α from appearing in the type of a. From this we can conclude that the context α and any contexts inside α and their contents become garbage when a finishes evaluating, and thus can all be deleted. The reasoning is as follows:

> Consider the term **new** $\alpha \lhd p$ **in** a, which has the syntactic restriction that α cannot be used as the representation context for any object. Assume that $\beta \prec: \alpha$ and that $[\ldots]^\beta$ is an object created while evaluating the expression a. To safely delete $[\ldots]^\beta$ there must be no references to it from an object which resides outside of context α. If such a reference existed it would be from an object $[\ldots]^\delta_\gamma$, where the containment invariant implies that $\gamma \prec: \beta$. For this object to be outside of α, we must have that $\delta \not\prec: \alpha$. Since the

context ordering is tree-shaped, and only contexts higher up in the tree are accessible, we must have that $\delta :\succ \alpha$ and $\delta \neq \alpha$. Pulling these constraints together, we obtain $\gamma \prec: \beta \prec: \alpha \prec: \delta$ and $\alpha \neq \delta$. Now the condition **DI** implies that $\gamma = \beta = \alpha$. But the syntactic restriction prevents α from being the representation context of an object, thus the external object $[\ldots]^{\delta}_{\gamma}$ with a reference to an object $[\ldots]^{\beta}$ cannot exist. We conclude that all objects inside α become garbage when *a* has finished evaluation.

There are some significant differences between this behaviour and that of **letregion**. Firstly, the nesting between our contexts is explicit in the program syntax and need not be linear like the stack of regions underlying the regions calculus [188]. An entire tree of contexts may exist beneath the context α when it is deleted. This is because each object placed in α may have its own representation context, which in turn may contain a nested collection of objects. This also means that not just the surface level structure of objects is deleted when the context goes out of scope; rather whole object graphs can be deleted. Secondly, we also have subtyping, whereas the region calculus does not. The main feature of the region calculus we lack concerns dangling pointers. The region calculus allows a region to be deleted even though there may be a reference into it from an active region. The regions type system ensures that such pointers are never dereferenced, so such behaviour is safe [188]. We cannot do this.

Note that we have not proven the required safety property, though the invariants seem clear based on the reasoning above.

6.2.12 Borrowing

A method is said to borrow one of its arguments if the object whose method it is retains no references to that argument. Boyland's Alias Burying proposal includes borrowing to allow unique references to objects to be passed to methods, temporarily violating the uniqueness property for the duration of the method to avoid unnecessary destructive reads [32]. Islands [109] and The Geneva Convention on the Treatment of Object Aliasing [110] also describe a notion of borrowing.

In the **URDI**ς calculus context polymorphic methods implement a kind of borrowing, since they can be passed objects which would otherwise be inaccessible.

The **URDI**ς calculus allows behaviour which we believe that Boyland's system does not. A method can create objects which store references to the borrowed objects with the proviso that such objects are not accessible outside of the method body. Such

objects are the stack-based objects described in the previous section. They become garbage when the method returns, and thus the references to borrowed objects vanish with them. This behaviour, and borrowing in general, is possible because the **URDI**ς calculus prevents all vampiric behaviour.

The following example illustrates borrowing (where Car^{myrep} denotes the type of my car, which has the context *myrep* as its owner, indicating that it is a part of my representation):

$$
\begin{array}{l}
mycar = [\ldots, fix = \cdots, \ldots]^{myrep} : Car^{myrep} \\
mechanic = [repair = \varsigma(\_, \alpha \prec: \varepsilon, c : Car^{\alpha}) \\
\quad \textbf{new}\ \gamma \lhd \alpha\ \textbf{in} \qquad\qquad -\gamma \textit{ and its contents exist for duration of method only} \\
\quad\quad \textbf{let}\ acidbath = \textbf{newhide}\ \beta \prec: \gamma\ in\ [bathe = \varsigma() \cdots c \cdots]^{\gamma}_{\beta}\ \textbf{in} \\
\quad\quad\quad acidbath.bathe\ ; \\
\quad\quad c.fix] \\
mechanic.repair\langle myrep, mycar\rangle
\end{array}
$$

Generally the *mechanic* does not have access to *mycar*, because it does not have access to *myrep*. But the method *repair* is polymorphic in the owner of the car. The owner of *mycar* can be passed to that method along with *mycar*, allowing the *mechanic* to hold a reference to *mycar* for the duration of the method. Within the *repair* method, an *acidbath* object is created which holds a reference to *mycar*. The owner of the *acidbath* object is a context γ, whose lifetime is limited to the scope of the **new** construct. Thus any reference to *mycar* held by the *acidbath* or *mechanic* objects vanishes when the method finishes evaluating. This can be shown following the reasoning given in the previous section.

Borrowing exhibits behaviour similar to the regions parameters in the region calculus [188]. Values from any region can be passed to a function which is region polymorphic, without violating the invariants that allow stack-based memory management. Similarly Zdancewic *et. al.*'s Principals [200] exhibit a similar degree of protection when values of abstract type are passed to functions. These proposals support neither objects nor subtyping. Boyland, Noble, and Retert [33] include borrowing as one of the features which can be encoded in their Capabilities for Sharing. Their setting is untyped and their notion of borrowing does not extend as far as the one described here.

6.2.13 Commentary and Extensions

We presented a number of calculi which are variations of the calculus presented in Chapter 5. The more expressive calculi had less strict properties, and thus some of the invariants assumed by some of our examples could not be satisfied for those calculi. The most important property was the absence of vampiric behaviour, which only the calculus **URDI**ς enforces. Unfortunately, this calculus is not expressive enough for all the examples we wish to write. Some work has been done towards obtaining an expressive calculus that is comparable in power to **UR**ς, but maintains some of the structural invariants of **URDI**ς. A preliminary version of this type system was presented at the informal SPACE 2001 workshop in London in January 2001. The key idea was to allow vampiric behaviour, but to place a bound on the position of the owner of the vampiric proxy object. Certain bounds could exclude vampires entirely. Subtyping allowed the bound to tighten, thus preventing external vampiric behaviour and facilitating borrowing, stack-based allocation of objects, safe initialisation, while still allowing aggregate objects with internally created iterator-like objects. While appealing, this extension is rather complicated and its properties have not been formally established. Thus we omit the details here.

We now describe how ownership types can be added to a class-based language.

6.3 Classes with Ownership Types

Most object-oriented programming languages are class-based. Although the calculus just presented is object-based, classes can be encoded using objects. The original ownership types proposal lacked inheritance, subtyping, and inner classes [61]. Now that we have developed quite an understanding of subtyping, the lexical nesting of objects, and their containment, we can apply this knowledge to add ownership types to a class-based language with inheritance, subtyping, and inner classes, such as Java [93], and thus provide representation containment in a more practical setting.

While a direct encoding of classes is possible [3], performing this tedious task would obscure the details of adding ownership types to a real programming language. Similarly, although the semantics of Java and the soundness of its type system are becoming well-understood [70, 146, 115], adopting one of these proposals as the basis for our work would be overkill, because of their complexity and their compliance to

the quirks of Java. The illustrative approach we take presents examples, both in a Java-like language and in our calculus, noting where variations are possible, and focusing predominantly on the constraints required for soundness and containment invariance.

6.3.1 The Original Ownership Types System (OOTS)

Our original ownership types proposal [61] is best explained using an example. Consider the following `Car` class:

```
class Car<owner|drowner>
    Engine<rep> engine = new Engine();
    Driver<drowner> driver;

    void drive() {
        driver.ready_and_alert();
        engine.start();
    }
}
```

This code declares a class where the owner of its instances is given by the context parameter `owner`. (In the original proposal `owner` was a keyword implicitly denoting the owner.) The context parameter `drowner` is the owner of objects stored in the `driver` field, as the type `Driver<drowner>` indicates. The separator `|` in the parameter list separates the owner from the other parameters, although there may not always be a parameter list. The keyword `rep` indicates that the owner of the `engine` field is the current object. For example, the type `Engine<rep>` indicates that the engine is a part of the current object's representation. Objects with owner `rep` are not accessible from outside the current instance.

An instance of the class `Car` could have type `Car<p|norep>`. In this type `p`, presumably a context parameter, is the owner of that instance, and `norep` is the owner of the `Car`'s `driver`. The context `norep` is a constant context corresponding to *nobody's representation*. Objects with this owner may be accessed by any object.

Type checking method and field selection and field update is simply a matter of determining whether the access is external or internal (via `this`), and then disallowing external accesses whenever `rep` appears in the field or method's type. Thus if `x` has type `Car<p|norep>`, the access `x.engine` is invalid, for example, though the internal

access to the engine field in the body of the drive method above is valid. Internal access to the driver field has type Driver<drowner>, whereas an external access such as x.driver has type Driver<norep>, to account for the binding of the parameters. Of course, type checking also must ensure that class names and parameters match when performing assignments, calling methods and so forth.

Further examples can exploit the owner parameter to enable complete linked data structures as part of the representation. Consider the following code fragment:

```
class Link<owner|d> {
   Data<d> data;
   Link<owner|d> next;
}

class List<owner|a> {
    Link<rep|a> head;
    ...
}
```

In class List the owner of the Link object is rep. The owner of the subsequent Link, head.next, is the context bound to owner, thus all Links are owned by the same object, which will be an instance of the List. Similarly, the owner of all the Data in the List is the context bound to parameter a of the List.

This list class can be used within another class to produce a deeper nesting of objects. For example, the Car class could have a field called passengers of type List<rep|drowner> to store the Car's Passenger objects. The List itself is a part of the representation of the Car, but the Passengers are owned by whichever context binds to the context parameter drowner.

The ownership types system does not confuse the rep context of different objects, even of two objects from the same class. This makes the protection object-based, as the following interpretation of contexts shows. Contexts can be interpreted as either an object's location or some global root for norep. Each rep context is interpreted as self or this for the class in which it appears. Stating that an object has owner rep indicates that it was owned by the current instance. Using this idea we were able to prove that well-typed programs satisfy the property that an object's owner was a dominator for access paths from the root of the system to that object [61].

OOTS has been implemented, modulo minor syntactic changes, as a Java extension by Boris Bokowski [23] who stressed the need for context parameters on methods, and by Alex Buckley [39] who included them plus numerous other extensions, and by us on top of Odersky and Wadler's Pizza [153].

Inheritance was not included in any of these systems, partially because we did not originally know how to add subtyping in a way which preserved the containment invariant. We now realise that we had not considered the nesting between contexts, which could be forgotten using subsumption, and thus programs could use subtyping to violate the dominators property.

Our first step along the path to inheritance is a discussion of the correspondence between the aspects of OOTS and the features of our calculus.

6.3.2 Classes and the correspondence with the calculus

One of the original reasons for developing the calculus studied in this thesis was to understand the annotations in our original ownership types system [61]. With this understanding we could then safely extend the original system while maintaining a well-specified and sound notion of containment. We now discuss the correspondence we sought between the features of OOTS and our calculus.

Firstly, the keyword `norep`, the owner of objects which can be shared system-wide, corresponds to the single maximal context ε.

To understand the meaning of the annotations `rep` and `owner` and the context parameters, we must rely on class encodings, but we can now examine objects for a hint. Consider the following general form of protected object and its type:

$$\textbf{hide } q \textbf{ as } \alpha \prec: p \textbf{ in } [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i\in 1..n}]^p_q \ : \ \exists(\alpha \prec: p)[l_i : \Theta_i{}^{i\in 1..n}]^p_\alpha.$$

The representation context q, hidden as α using existential quantification, corresponds approximately to the keyword `rep`, except that the former (q) is at the object level, while the latter (`rep`) is at the class level. The former can be seen as the implementation of instances of the latter. In OOTS, `rep` was interpreted as the object's identity. In calculi for which **UR** holds this is a sound perspective, since there is a one-to-one correspondence between objects and their representation contexts. In addition, the hiding behaviour of existential quantification captures the unknown nature of `rep` — the `rep` for objects other than `this` cannot be represented in code.

The owner context at the object level corresponds to the implementation of the `owner` keyword/parameter at the class level. In the interpretation underlying OOTS

$$
\begin{array}{l}
\mathit{classCar} : \mathit{ClassCar} \;\widehat{=}\; \\
\quad \lambda(\mathit{owner} \prec\!: \varepsilon, \mathit{drowner} :\!\succ \mathit{owner}) \\
\quad\quad [\mathit{new} = \varsigma(\mathit{cls} : \mathit{PreClassCar}\langle \mathit{owner}, \mathit{drowner}\rangle, d : \exists \mathit{Driver}\langle \mathit{drowner}\rangle) \\
\quad\quad\quad \textbf{newhide}\ \mathit{rep} \prec\!: \mathit{owner}\ \textbf{in} \\
\quad\quad\quad\quad \textbf{let}\ e = \mathit{engineClass}\langle \mathit{rep}\rangle.\mathit{new}\ \textbf{in} \\
\quad\quad\quad\quad\quad [\mathit{engine} = \varsigma()e, \\
\quad\quad\quad\quad\quad\ \mathit{driver} = \varsigma()d, \\
\quad\quad\quad\quad\quad\ \mathit{drive} = \varsigma(\mathit{self} : \mathit{Car}_{\mathit{rep}}\langle \mathit{owner} \mid \mathit{drowner}\rangle)\mathit{cls}.\mathit{drive}\langle \mathit{rep}, \mathit{self}\rangle]^{\mathit{owner}}_{\mathit{rep}} \\
\quad\quad\quad \mathit{drive} = \varsigma(\_, \mathit{rep} \prec\!: \mathit{owner}, \mathit{car} : \mathit{Car}_{\mathit{rep}}\langle \mathit{owner} \mid \mathit{drowner}\rangle) \\
\quad\quad\quad\quad \mathit{car}.\mathit{driver}_{\circ}\mathit{ready\_and\_alert}\langle\rangle; \\
\quad\quad\quad\quad \mathit{car}.\mathit{engine}_{\circ}\mathit{start}\langle\rangle \quad]^{\varepsilon}
\end{array}
$$

Figure 6.4: The Encoding of Class `Car`

the owner was denoted by the object corresponding to the actual owner, or the root of the system, when the owner context was `norep`. This interpretation is faithful whenever **UR** holds, because $\mathsf{rep}^{-1}(\mathsf{owner}(\iota))$ is either the object corresponding to the owner of ι, or undefined when $\mathsf{owner}(\iota) = \varepsilon$.

Context parameters are more easily understood at the class level, so we will postpone our discussion of them.

Field and method access in OOTS was governed by a syntactic condition which ensured that a field containing `rep` in its type could be accessed using only `this`, that is, only by the instance of the class in which the code appeared. The encodings of safe external method access and update presented in Section 6.2.9 correspond closely with this behaviour.

A Class Encoding Now let's consider an encoding of the `Car` class from above (with the addition of an argument to its constructor). We are using the **URDI**ς calculus, plus the safe external method access and update encodings from Section 6.2.9. Figure 6.4 presents the encoding. This follows the encoding of a second-order language in Abadi and Cardelli's object calculus [3], using contexts and context variables instead of type and type variables.

The encoding relies on the following simple function encoding:

$$\lambda(\mathit{owner} \prec\!: \varepsilon, \mathit{drowner} :\!\succ \mathit{owner})b \;\;\widehat{=}\;\; [\mathit{fun} = \varsigma(\_, \mathit{owner} \prec\!: \varepsilon, \mathit{drowner} :\!\succ \mathit{owner})b]^{\varepsilon}$$

which can be applied simply by selecting the method *fun* with the appropriate number of arguments; and on the following abbreviations:

- $\exists$*Driver*<*drowner*> corresponds to the type of a protected `Driver` object with owner *drowner*, that is, as it is viewed externally. Similarly $\exists$*Engine*<*rep*> corresponds to an `Engine` object with owner *rep*.

- $\exists$*Car*<*owner* | *drowner*> $\equiv \exists(rep \prec: owner)\,[engine : \exists Engine\langle rep\rangle,\ driver : \exists Driver\langle drowner\rangle,\ drive : Unit]_{rep}^{owner}$ is the type of protected objects from class `Car` with parameters *owner* and *drowner*. *Unit* corresponds to the unit type. Alternatively, but not equivalently, we could use $[\,]^{\varepsilon}$.

- Car_{rep}<*owner* | *drowner*> corresponds to the type of a `Car` object as viewed internally with owner *owner* and representation context *rep*.

- *PreClassCar*<*owner*, *drowner*> $\equiv [new : \exists Driver\langle drowner\rangle \to \exists Car\langle owner \mid drowner\rangle,\ drive : \forall(rep \prec: owner) Car_{rep}\langle owner \mid drowner\rangle \to Unit]^{\varepsilon}$ is the type of the `Car` after its parameters *owner* and *drowner* have been instantiated.

- *ClassCar* $\equiv \forall(owner :\succ \varepsilon)\forall(drowner :\succ owner)$*PreClassCar*<*owner*, *drowner*> is the type of the context polymorphic `Car` class, that is, before any parameters have been supplied.

The expression *classCar* encodes the class `Car`. It takes two parameters, the *owner* and *drowner* contexts, where *owner* is unconstrained, and *drowner* contains *owner*. The *owner* parameter corresponds to the `owner` keyword from OOTS. Context parameters such as *drowner* correspond to the context parameters from OOTS.

The object within the λ corresponds to the class after the parameters have been instantiated. Its *new* method is the constructor. *new* takes a parameter *d* to initialise the *driver* field.

The self parameter *cls* of *new* refers to the class object and is used to supply methods (and static fields) to objects of this class. This is an aspect of the usual class encoding from *A Theory of Objects* [3].

The *new* method first creates a new context inside the *owner* context, and hides it using existential quantification. This context is used as the representation context of

the object to be created, and corresponds to the `rep` keyword in OOTS. The remainder of the *new* method produces a new object with owner *owner* and representation context *rep*. The representation context is used as owner when creating a new engine for the *engine* field by a call to the *Engine* constructor (not given here). Next, the *driver* field is initialised with the parameter *d*. Finally, the *drive* method calls the *drive* method from the class, using *cls*, passing both its representation context and itself as parameters.

A new car is created using

$$classCar\langle p, q\rangle.new\langle d\rangle$$

where p and q are contexts and d is of type $\exists Driver\texttt{<}drowner\texttt{>}$. More precisely, by unravelling the function encoding of *classCar*, this expression corresponds to

$$classCar.fun\langle p, q\rangle.new\langle d\rangle.$$

The corresponding constructor call in OOTS is:

```
new Car<p,q>(d);
```

(Here we extended the constructor function with arguments; this was not present in the OOTS formalism.)

The *drive* method of the class takes a context parameter corresponding to the object's representation context and *Car* object from its internal view. The method then calls a method on the *car*'s *driver* and *engine*, just as the OOTS example did. In this case the representation context is used only to give a type to *car*. It could also be used to create other other objects, such as a new *engine*.

Summary The correspondence between elements of our original ownership types system and the calculus presented in this thesis can now be summarised:

- OOTS's `norep` keyword corresponds to the single maximal context ε.
- OOTS's `rep` keyword corresponds to a context in the class encoding which is newly created and hidden just inside the owner context and used as the representation context for a newly created object.
- OOTS's `owner` keyword corresponds to the parameter in the class encoding which will become the owner of the object which the constructor of the class produces.

- context parameters correspond to the non-owner parameters in the class encoding.

What was lacking from OOTS, as this encoding makes apparent, is the explicit nesting between contexts. The omission of the nesting constraints is the reason OOTS lacked subtyping, because without the constraints an expression could be constructed which violated the owners-as-dominators property.

We now outline the constraints required to extended OOTS with subtyping, inheritance, and so forth. All of the constraints and limitations discussed below follow from the constraints in the type system presented in Chapter 5, based on the encoding we have just outlined.

In the following we use `<:` and `:>` within the programming language syntax to correspond to $\prec:$ and $:\succ$ and $<:$ in the text for the subtype relation, as usual.

Constraints on classes Within each class body a number of contexts are visible. These include global contexts, context parameters to the class including its owner, and of course `rep`. We will introduce others a little later on. When type checking the body of a class, we need to know the constraints on the visible contexts to determine whether the types appearing in the class body are well-formed.

We now enumerate the kinds of constraint which may appear. More will be given later as we introduce more features into our language.

Consider the class header:

```
class Car<owner|drowner> { ... }
```

The following constraints are present when checking the body of this class:

1. `rep <: owner`

   The typing rules for objects (Type Object) and (Val Object) necessitate this constraint.

2. `owner <: drowner`

   Required to obtain that `rep <: drowner`, which is required so that the instance of `Car` can access objects with owner `drowner`, as the type rule for objects in the calculus dictates.

Without this constraint it is possible to construct the type `Car<p|rep>`, where `rep <: p`, and then use the subtype relation `Car<p|rep>` $<:$ `Object<p>` to export values of this type to objects which may have access to context `p`. Such objects would then have an indirect access path to the representation which does not pass through the owner of that representation. Thus this kind of subtype relation breaks the owner-as-dominators property. The constraint between `owner` and `drowner` excludes types such as `Car<p|rep>`.

3. Any constraints between the context parameters.

   Consider, for example,

   ```
   class ParkingSpace<owner|o,d :> o> {
       Car<o|d> car;
   }
   ```

   specifies a constraint that the third parameter `d` contains `o`, that is, `d :> o`. Without this constraint `Car<o|d>` is not a valid type, because the constraints on the parameters in the class `Car` are not satisfied.

4. `owner <: norep`

   This allows `owner` to be bound to any context. This constraint may be tightened to limit the use of the class, as we will demonstrate in Section 6.3.10.

5. Finally, we need to add to our collection of constraints any constraints on other contexts, such as constant contexts, which are visible in the scope of the class being checked.

A number of the constraints discussed above were implicit. These could be specified, possibly using different constraints than the defaults we give. The header for class `Car` with all the constraints made explicit is:

```
class Car<owner <: norep|drowner :> owner>
     where rep <: owner { ... }
```

Manifest Ownership It is possible to specify that all instances of a class have the same owner. Indeed this is essential when dealing with static methods and fields (Section 6.3.5) and sometimes in inner classes (Section 6.3.3). The context specified as the owner of all instances of a class is called the *manifest owner* and is indicated

using the ownedby keyword.[2] The manifest owner must be specified to determine the owner of this.

Consider the class header:

```
class A ownedby context { ... }
```

All instances of class A are owned by the context context, which can be any context visible in the scope where A is defined.

Manifest ownership can also be used to embed any Java class into our programming language. We can consider the Java class `class A { ... }` as shorthand for `class A ownedby norep { ... }`.

Note that manifest ownership is really just a convenient shorthand notation.

```
class A ownedby context { ... }
```

is shorthand for

```
class A extends Object<context> { ... }.
```

This relies on inheritance which will be discussed in Section 6.3.4.

A consequence of the type theory is that a class with manifest ownership cannot have any context parameters.

The type of `this` The type of this can now be determined. There are two cases:

- within the class with header `class Car<owner|drowner :> owner>`, the type of this is `Car<owner|drowner>`.
- within the class with header `class A ownedby context`, the type of this is A. The actual owner can be determined from the text of class A, if needed.

Types We now discuss well-formedness of types. Given the well-formed class header

```
class C<o <: norep | p1 :> c1, p2 <: c2, ..., pn :> cn>
```

the following method determines whether the type `C<a|b1,b2,...,bn>` is well-formed.

[2] We use the passive voice for this keyword to emphasise that the class has non-variable ownership.

1. From the header of class `C` derive the partial order

   $F = \{$`o<:norep, p1:> c1, p2<:c2, ..., pn:>cn, o<:p1,p2,...,pn` $\}$

   This is the partial order on the *F*ormal parameters.

2. Let A be the partial order of the constraints on the contexts visible in the scope where the type `C<a|b1,b2,...,bn>` appears. This is the partial order on the *A*ctual parameters.

3. The type `C<a|b1,b2,...,bn>` is invalid if any of `a`, `b1`, `b2`, ..., `bn` are not in the domain of partial order A.

4. The type `C<a|b1,b2,...,bn>` is valid whenever `a`, `b1`, `b2`, ..., `bn` satisfy the constraints on the formal parameters. This is true whenever the substitution of actual parameters for formal parameters is an order-preserving map from F to A. In the example, we require that $\theta = \{\texttt{o} \mapsto \texttt{a}, \texttt{p1} \mapsto \texttt{b1}, \texttt{p2} \mapsto \texttt{b2}, \ldots, \texttt{pn} \mapsto \texttt{bn}\}$ is an order-preserving map.

   An alternative way of characterising this check is to consider both A and F as sets of constraints on contexts (closed under reflexivity and transitivity). If $\theta(F)$ is the constraint set which results from substituting for the variables in F, then we say that a type is valid whenever

   $$A \supseteq \theta(F),$$

   that is, whenever the constraints in A subsume or imply those in $\theta(F)$.

Note that this procedure excludes types such as `Car<p|rep>` where `rep` appears on the right-hand side of the "|" but not on the left.

Field and Method Access and their types We now informally give rules governing the types and accessibility of fields and methods. There are two cases: internal access, whenever a field or method is accessed using `this` (perhaps implicitly), and external access, which is every other kind of access. In the presence of inner classes (discussed next) access to fields and methods in the surrounding class follows the rules for internal access.

The rules are:

- all fields and methods can be accessed internally; their type is as it appears in the program text;
- a field or method is externally accessible if and only if `rep` does not appear in the type which appears in the program text for that attribute. Thus such fields and methods can be considered private (or, more accurately, `protected`).
- Let `e` be an expression of type `C<...>` and θ be the substitution from the actual parameters of this type to the formal parameters of class `C`. If `T` is the type of field `f` in class `C`, then the type of `e.f` is $\theta(\texttt{T})$, where

  $$\theta(\texttt{D<o|p1,p2...>}) \mathrel{\widehat{=}} \texttt{D<}\theta(\texttt{o})\texttt{|}\theta(\texttt{p1}),\theta(\texttt{p2})\texttt{...>}.$$

  Method types are analogously determined by applying θ to all argument and return types of that method.

The last point is best illustrated with an example. Consider the following class:

```
class Car<owner|drowner> {
    Driver<drowner> driver;
}
```

From a variable `car` of type `Car<a|b>`, we obtain $\theta = \{\texttt{owner} \mapsto \texttt{a}, \texttt{drowner} \mapsto \texttt{b}\}$. `car.driver` has type $\theta(\texttt{Driver<drowner>}) = \texttt{Driver<}\theta(\texttt{drowner})\texttt{>} = \texttt{Driver<b>}$.

Determining the types becomes more complicated in the presence of context polymorphic methods (Section 6.3.7), but the principle remains the same.

Type Compatibility I A value can be stored in a field or passed as a method argument only if the type of the value conforms with the field or argument type. In the absence of subtyping the statement of type conformance is simple:

Two types in the same scope conform if they are syntactically identical.

This of course means that we must convert types which appear in different scopes, as we have described above.

After we discuss inheritance in Section 6.3.4, we will complete the definition of type compatibility in the presence of subtyping.

Remark 6.5 *Classes as described thus far can be encoded in the* ***URDI***ς *calculus. Inner classes of some form can be added, but not those which are externally accessible like iterators. To do this we must shift to the* ***UR***ς *calculus. We implicitly assume that this transition has been made from now on.*

We now account for a few additional features: inner classes, inheritance, inheritance of inner classes, static fields and methods, exceptions and errors, context polymorphic methods, friendly functions, generic classes, before describing some enhanced notions of privacy which a class-based programming language with ownership types allow.

6.3.3 Inner Classes

An inner class is a class which is defined within the body of another class. This inspired extension[3] to Java [93], present also in Simula [20] and Beta [134], can be used to implement aggregates objects which have multiple interfaces, where the additional interfaces are implemented using inner classes. Inner classes greatly improve the elegance of Java, in the very least, alleviating the programmer of much unnecessary parameter passing which is often required to build complex abstractions.

An inner class is best viewed as a class that is defined within an instance of the surrounding class. Thus each instance of the surrounding class can have within it many instances of the inner class. Each of these has access to all the fields and methods of the instance of the surrounding class [93]. In our context, a consequence is that all instances of an inner class should be able to access the surrounding instance's representation.

The syntax for an inner class is the same as that of an outer class, though now a few additional factors need to be considered.

Representation contexts Both the inner and the surrounding class have a representation context denoted (in the appropriate scope) by `rep`. Thus the `rep` in the inner class shadows `rep` in the surrounding class. For example, in

[3]Others disagree. "Yacc, yacc, yacc!" Personal Communication. Sophia Drossopoulou. Imperial College, London.

```
class C<owner> {
    F<rep> fc;                // instance of class C's rep
    class D<owner'> {
        F<rep> fd;            // instance of class D's rep
        F<C.rep> fc2 = fc;    // (*)
    }
}
```

variables `fc` and `fd` have a different type since `rep` refers to a different context in each case, as indicated. The representation context of the surrounding instance of class `C` can be referred to using the syntax `C.rep`. The line marked `(*)` is a valid assignment. This syntax parallels Java's `C.this` which is used for referring to the surrounding instance of class `C` [93].

The position of `rep` For a top-level class we have assumed that `rep <: owner`, but for an inner class a more sensible default is to assume that the representation context of the inner class is inside that of the surrounding instance. This will allow instances of the inner class access to the surrounding instance's representation.

To make this constraint more explicit, we could have written the classes `C` and `D` above as:

```
class C<owner> {
    ...
    class D<owner'> where rep <: C.rep {
        ... }
}
```

Since this is a natural default it can be omitted.

Being able to explicitly state the constraint allows some variation. For example:

```
class C<owner> {
    ...
    class D<owner'> where rep <: owner' {
        ... }
}
```

This constraint implies that that instances of class `D` cannot access the surrounding `C` object's representation, even though `D` is syntactically nested inside `C`. (We are unsure whether this has any application.)

The position of the owner and other parameters The owner parameter must satisfy the usual constraints on classes, namely, `rep <: owner` and `owner <: norep`. Ownership can be manifest, including as a parameter from the surrounding class. This is useful, for example, to limit access to a `List`'s `Iterator` to the objects which can access the `List`. The following code achieves just that by specifying the manifest owner of the `Iterator` to be the owner of the surrounding `List` object:

```
class List<owner|a> {
    Link<rep|a> head;
    class Iterator ownedby owner {
       Link<List.rep|a> current;
    }
}
```

Again, if ownership is manifest then there can be no context parameters.

Constraints on other context parameters can be present. Again we require that the constraint `owner <: param` holds for each parameter `param`, if not already present.

Externally creating instances of an inner class It is possible to externally create instances of an inner class following the Java [93] by using syntax such as:

```
x.new D<args>;
```

This may not be desirable. Again, if `D`'s ownership is manifest as `rep`, then `D` cannot be accessed externally.

Inner classes in methods In Java it is possible to have inner classes defined within method bodies. These can easily be added to the language described here, although we feel that they should be excluded from context polymorphic methods (Section 6.3.7) and friendly functions (Section 6.3.8) since this would allow vampires to be arbitrarily created.

The extent of inner class owners We can examine the owner (parameter or manifest) of a class's inner classes to determine the extent to which instances of that class have their representation indirectly accessed.

Contrast these two nests

```
class A1<owner> {              | class A2<owner> {
    class B1<owner'> { ... }   |     class B2 ownedby owner { ... }
}                              | }
```

In the nest `A1-B1` there is no constraint on `B1`'s owner. It may not be contained within the owner of `A1`, thus allowing a proxy object to an `A1` object's representation which may be available to objects which cannot access the `A1` object.

On the other hand, the nest `A2-B2` does not suffer from this property. A `B2` object is visible to exactly the same objects as the surrounding `A2` object, because they must have the same owner.

Thus the degree of access to the alternative interfaces can be determined by inspecting the bounds on the owners of a class's inner classes. Controlling the inner class's owner limits the indirect access to an object's representation through the additional interfaces which inner classes implement.

It may be desirable to specify bounds on the owners of inner classes and ensure that these are preserved through inheritance. (See Section 6.2.13.)

6.3.4 Inheritance

When adding inheritance to a class-based language with ownership types, we must ensure that subsumption cannot be used to forget the information governing the external accessibility of an object. Recall that values of a type are externally accessible if and only if the type does not contain the keyword `rep`. The constraints discussed until now ensure that a type contains `rep` if and only if it contains `rep` in the owner position. This is because the owner must be inside (`<:`) every other context parameter. Thus examining the owner of a type is sufficient to determine whether it is accessible. This is what we would expect; in the calculi presented in this thesis the owner determined whether values of a type were accessible.

From this discussion we can deduce two constraints to guide valid inheritance:

- subsumption must not forget an object's owner; and
- a derived class must preserve the constraints on contexts specified in its the base class.

There is also the constraint that the types of overridden methods are invariant, modulo the substitution into the base class's parameters, as one would expect.

Let's now flesh out some details.

The class `Object` At the top of the class hierarchy is the class:

```
class Object<owner> { }
```

Because this class is parameterised, it permits a derived class to either have their own owner parameter or have manifest ownership, as we would expect.

Unfortunately, this minor but fundamental difference from `java.lang.Object` in Java makes it difficult to directly interoperate a language with ownership types with Java.

Constraints on Inheritance Consider the classes:

```
class Base<owner|p1,p2:>p1> { ... }

class Derived<owner'|p1',p2'<:p1'>
    extends Base<owner'|p2',p1'> { ... }
```

A substitution from the contexts visible in the class `Derived` to the parameters of the class `Base` can be derived from the `extends` clause:

$$\theta = \{\texttt{owner} \mapsto \texttt{owner'}, \texttt{p1} \mapsto \texttt{p2'}, \texttt{p2} \mapsto \texttt{p1'}\}.$$

For the inheritance clause to be valid the substitution must satisfy two constraints:

- it must map the owner of the base class to the owner of the derived class; and
- it must be an order-preserving map (as described above). This is equivalent to stating that `Base<owner'|p2',p1'>` is a type.

The last point deserves closer attention. The parameters of class `Base` form a partial order,

$$F_{\texttt{Base}} = \{\texttt{owner<:norep}, \texttt{p1:>owner}, \texttt{p2:>p1}\}.$$

Similarly, the partial order of the contexts in scope in class `Derived` is

$$A_{\texttt{Derived}} = \{\texttt{owner'<:norep}, \texttt{p1':>owner'}, \texttt{p2':>owner'}, \texttt{p2'<:p1'}, \ldots\}$$

Now, the `Derived` class preserves the constraints in the `Base` class if and only if

$$A_{\texttt{Derived}} \supseteq \theta(F_{\texttt{Base}}).$$

In this case the `extends` clause is valid because

$$\theta(F_{\texttt{Base}}) \equiv \{\texttt{owner'<:norep}, \texttt{p2':>owner'}, \texttt{p1':>p2'}\} \subseteq A_{\texttt{Derived}}.$$

Now lets explore some consequences:

1. Extending a base class which has manifest ownership requires that the owner remains the same. Thus the derived class must have manifest ownership. Consider class `C`:

   ```
   class B ownedby context { ... }
   class C ownedby context extends B { ... }
   ```

   The "`ownedby context`" in `C` is redundant, so could be omitted.

   A consequence is that the language we present here admits Java's inheritance, by assuming that every Java class has manifest owner `norep`.

2. The bound on an owner can be tightened in the derived class.

   ```
   class B<owner <: b> { ... }
   class D<owner' <: d> extends B<owner'>  { ... }
   ```

   We have $\theta = \{\texttt{owner} \mapsto \texttt{owner'}\}$, $A_{\texttt{D}} = \{\texttt{owner'<:d}, \ldots\}$ and $F_{\texttt{B}} = \{\texttt{owner<:b}\}$. Now $A_{\texttt{D}} \supseteq \theta(F_{\texttt{B}}) \Leftrightarrow \{\texttt{owner'<:d}, \ldots\} \supseteq \{\texttt{owner'<:b}\}$, which will be true whenever $\texttt{d<:b} \in A_{\texttt{D}}$. This means that the access to the derived class can be more restricted.

3. If `rep` appears in the parameter list of the base class, then the derived class must have `rep` as its manifest owner. For example,

   ```
   class D ownedby rep extends B<rep|a,rep,b> { ... }
   ```

   This only makes sense when applied to inner classes, where the `rep` would belong to the enclosing class.[4]

Types If a field appearing in the base class has type `T`, and θ is the substitution from the base class's formal parameters to the derived class's actual parameters, then the type of that field in the derived class is $\theta(\texttt{T})$. For example, given:

```
class ParkingSpace<owner|o,d :> o> {
    Car<o|d> car;
}
```

[4] Be careful! The syntax denoting the enclosing class's representation context is `rep` at this point, and `C.rep` within the body of class `D`.

```
class ExclusiveParkingSpace<owner'>
    extends ParkingSpace<owner'|owner',owner'> {
    ... car ... }
```

Based on the substitution $\theta = \{\texttt{owner} \mapsto \texttt{owner'}, \texttt{o} \mapsto \texttt{owner'}, \texttt{d} \mapsto \texttt{owner'}\}$, the occurrence of `car` in class `ExclusiveParkingSpace` has type `Car<owner'|owner'>`.

Type Compatibility II We now need to consider type compatibility in the presence of inheritance. This is based on a subtype relation.

Assume that we have the following classes

```
class Base<...> { ... }
class Derived<...>
    extends Base<d1|d2,...> { ... }     (*)
```

Consider a type from the `Derived` class: $T \equiv$ `Derived<t1|t2,...>`. Associated with this is a substitution θ for the formal parameters of `Derived`. Now the *direct supertype* of T is the type $\theta($`Base<d1|d2,...>`$) =$ `Base<`θ`(d1)|`θ`(d2),...>`, where we obtained `Base<d1|d2,...>` from line (*). The *supertype* relation is defined as the reflexive, transitive closure of this direct supertype relation. Now we have that U is a subtype of T whenever T is a supertype of U.

The conformance rule can now be stated:

> *A field, variable or parameter of type T can be assigned a value of type U if and only if U is a subtype of T.*

Method Overriding Following Java [93] and GJ [34] we add the requirement that the type of a method overridden in the derived class must be the same as the type of the method in the base class after the appropriate parameter substitution has been applied. Methods with context parameters must retain the same context parameters, modulo any changes on their bounds which the substitution may entail.

Interfaces Interfaces are treated no differently than classes, except that they can have more context parameters and fewer constraints. The reason is that an interface has no body, and thus there is no code enforcing constraints on the contexts which appear in the interface. Having more parameters would generally make interfaces more flexible, though perhaps a little more painful to use.

Care must be taken not to be too general in the interface specification, because it is unsound to override a context polymorphic method with one which is not. Consider the following example (where the dangling `<a>` syntax denotes an unbound context parameter to the method `munge`):

```
interface Munger {
    <a> void munge(A<a> fst);
}
class BadMunger implements Munger {
    static A<norep> glob;
    void munge(A<norep> fst) { glob = fst } // unsound overriding
}

Munger mmm = new BadMunger();
A<rep> a;
mmm.munge(a);    // would typecheck
```

This code is unsound because `A`'s munge method assigns its argument `fst` to a global variable of `glob`. The call `mmm.munge(a)` exposes the representation stored in variable `a`, by placing it in `glob`. This is unsound, thus a context polymorphic method cannot be overridden by one which is less polymorphic. Having this issue to consider makes interface design tricky, because the constraints which must be enforced must be anticipated before all uses of the interface are known.

Inheritance of inner classes from the outside A derived class may have an inner class which derives from an inner class that appears in its base class (although this is not possible in Java). The constraints which must be satisfied follow the same pattern as those described thus far.

Consider the following:

```
class Base<...> {                          (1)
    class InnerBase<...> { ... }           (2)
}
class Derived<...> extends Base<...> {     (3)
    class InnerDerived<...>
        extends InnerBase<...> { ... }     (4)
}
```

As before, we can derive a number of partial order and substitutions from this code. From (1) and (2) we have the constraints on formal parameters $F_{\texttt{Base}}$ and $F_{\texttt{InnerBase}}$. We assume without loss of generality that their domains are distinct.

The contexts visible at (3) and (4) form the partial orders $A_{\texttt{Derived}}$ and $A_{\texttt{InnerDerived}}$, where $A_{\texttt{Derived}} \subseteq A_{\texttt{InnerDerived}}$, since the latter contains the formal parameters declared in class `Derived`.

From (3) and (4) we can determine the substitutions from formal to actual parameters $\theta_{\texttt{Derived}}$ and $\theta_{\texttt{InnerDerived}}$.

As before, the constraints on class `Derived` are that $A_{\texttt{Derived}} \supseteq \theta_{\texttt{Derived}}(F_{\texttt{Base}})$. Now the constraints on the formal parameters of class `InnerBase` may have bounds which are formal parameters of class `Base`. To determine the constraints on the formal parameters of `InnerDerived`, we must first apply the substitution $\theta_{\texttt{Derived}}$ to $F_{\texttt{InnerBase}}$ to convert any bounds in terms of parameters from `Base` into their counterpart in `Derived`, before applying $\theta_{\texttt{InnerDerived}}$ to obtain the constraints in terms of the contexts in the scope where `InnerDerived` is defined. Thus class `InnerDerived` must satisfy:

$$A_{\texttt{InnerDerived}} \supseteq \theta_{\texttt{InnerDerived}}(\theta_{\texttt{Derived}}(F_{\texttt{InnerBase}})).$$

These constraints can of course be generalised for deeper nesting of inner classes.

As a final note, we believe that it is reasonable to allow inner classes which have manifest owner `rep` to be inherited. Thus it is more fruitful to consider `rep` to be analogous with `protected` rather than `private`.

Some of the issues regarding inheritance discussed in this chapter were resolved with the assistance of Ryan Shelswell, James Noble and John Potter [62].

6.3.5 Static Fields and Methods

In Java and C++ a static field or method is a global variable or function defined within the text of a class [93, 72]. Since they are global, statics can only be owned by constant contexts. A consequence of this is that if somewhere in a class `this` is assigned to a static field, then the class cannot have any context parameters, since all instances must have the same type as the static field. Various design patterns employ a similar mechanism; the Flyweight pattern is one example [84].

Consider the following:

```
class A<owner> {
    static A<??> glob;
    void breakit() {
       glob = this;  // doesn't work
    }
}
```

The parameter `owner` of class `A` would normally be instantiated with any context. Because of the assignment in method `breakit()`, the owner (given by `??`) in the type of `glob` would have to match the instantiated context for all contexts. But this is impossible. Thus the example is not well-formed.

Instead class `A` must have no owner parameter, and all instances of class `A` would be owned by `norep`, by default, or some other visible context using manifest ownership. For example, the following is valid:

```
class A {
    static A glob;
    void breakit() {
       glob = this;  // okay
    }
}
```

As discussed above, such classes cannot have any other context parameters.

A similar problem also occurs when an instance of some inner class assigns itself to a field in an surrounding class. A consequence is that the inner class cannot be parameterised and must use manifest ownership.

Static methods are handled using similar reasoning.

6.3.6 Exceptions and Errors

Exceptions and errors are handled similarly to statics. The classes representing these cannot have context parameters. They must be owned by the context `norep` or some globally accessible constant context. Exceptions and errors can potentially be accessed anywhere in the source code as their control flow does not follow the usual paths. This approach follows that of Bokowski and Vitek's Confined Types [24].

6.3.7 Context Polymorphic Methods

A method which can be called with arguments having different owners is called context polymorphic. These can be specified in the class-based language by prefixing a method signature with the context parameters to the method (similar to GJ's type parameters [34]). For example, the method

```
<l,d :> l> int length(List<l|d>) { ... }
```

has the parameter list `<l,d :> l>`. This method can be applied to any list, because the owner parameter `l` is unbounded, and the parameter `d` satisfies the constraint on `List`'s first argument. We are forced to include the constraint `d:>l` on the parameter corresponding to the data's owner to match the constraints imposed by the class `List`, namely that the list's owner is inside the owner of the list's data (the first parameter),

When type checking the body of this method, the constraints `l<:norep` and `d:>l` are added to the typing environment. Checking the validity of a call to a context polymorphic method simply requires checking that the substitution of actual to formal parameters satisfies the constraints given in the parameter list. Note that it is possible to call the method without explicitly supplying the context parameters, by allowing the type checker to infer them from the types of the method's arguments.

The `length` method can be encoded in any of our calculi as:

$$[\ldots, length = \varsigma(\_, l \prec: \varepsilon, d :\succ l, list : List\langle l \,|\, d\rangle)\cdots, \ldots] :$$
$$[\ldots, length : \forall(l \prec: \varepsilon)\forall(d :\succ l) List\langle l \,|\, d\rangle \to Int, \ldots]$$

where $List\langle l \,|\, d\rangle$ is the type of list with owner l and data owned by d.

The syntax we suggest differs from Buckley's proposal [39]. He employs a Miranda-like syntax of `*`, `**`, `***`, etc [190] to represent the context parameters, and he omits the parameter list. His proposal lacked constraints between context parameters; with the constraints present our approach seems more expressive.

6.3.8 Friendly Functions or Subversive Methods

Friendly functions can be readily added. We do so by taking advantage of the so-called *dot notation* [52]. This provides a convenient and safe way of denoting a hidden type, or in our case, the hidden representation context, while preserving the scope of the hidden context.

Consider the following method, assuming that class `A` has a field of type `B<rep>`:

```
subvert bool equals(A<a> a1, A<b> a2) {
    return (a1.b == a2.b);
}
```

Normally this code would be invalid in a language with ownership types, since it involves external access to a `rep` field. But the intention of the `subvert` keyword is to permit access to the `rep` fields of each of the method's arguments (an alternative is to mark the arguments individually). For this to work we need to distinguish each argument's representation context. The dot notation supports this by denoting the representation of the object stored in variable `a1` as `a1.rep`. Thus `a1.b` has type `B<a1.rep>` and `a2.b` has type `B<a2.rep>`. These two types are incompatible, since their owners `a1.rep` and `a2.rep` are distinct. In addition, values of types `B<a1.rep>` and `B<a2.rep>` cannot escape the scope of the method, since part of each type depends upon the name of a variable that is not defined outside that scope.

The dot notation imposes one minor restriction: the parameter variables cannot be reassigned; or more generally, that any subverted variables cannot be reassigned. This is not a harsh restriction, since the contents of a parameter can be stored in another variable, but it does ensure that the actual representation context referred to by `a1.rep` cannot change. Reassigning a parameter could change the context which to `a1.rep` refers, and this could result in unsound behaviour.

To type check the body of this method, we would add the constraints `a1.rep<:a` and `a2.rep<:b` to the typing environment. This allows, for example, the subverted representation to be safely stored in local data structures and passed to context polymorphic utility functions.

There is also the possibility of adding the `subvert` keyword to local variables which one wishes to subvert. For example,

```
subvert A<a> a1 = ...;
```

`a1.b` is accessible and has type `B<a1.rep>`. Again, the variable cannot be reassigned to, so it must be initialised (or tracked so that it is assigned to only once).

The advantage of the `subvert` keyword is that it advertises whenever representation is accessed outside of its usual boundary. The dot notation offers an additional advantage. The type of any subverted representation includes the name of the variable which contains the subverted object. This makes it easier to keep track of which objects own which protected objects.

This approach has a disadvantage from a security point of view. An applet downloaded from the web may use subversion to access the internal representation of the web browser, for example.[5] Clearly, stronger interaction with the usual privacy mechanisms is needed to limit which variables can be viewed using subversion. Another alternative is to revert to something like C++'s friends [72].

6.3.9 Generic Classes

We have developed enough type theory to facilitate the addition of generic classes to our class-based language. We now briefly discuss the extensions, following the syntax of GJ [34]. The rules required for type checking this extension follow those of GJ, taking into account the factors discussed here.

The first thing we add are type parameters which may be bounded. These can appear in the context parameter lists of both classes and polymorphic methods.[6]

Before presenting our example, it is worth observing that unbounded type parameters admit a useful shorthand which somewhat reduces the programmer's syntactic burden. For example, we can consider the following code for a single slot container

```
class Cell<owner|X> {
    X item;
}
```

to be shorthand for

```
class Cell<owner|fresh :> owner, X extends Object<fresh>> {
    X item;
}
```

where `fresh` is a fresh context parameter. We simply expand the shorthand notation by generating a fresh context variable for each type variable, adding bounds as we have done here for X.

Care is of course required when checking types from class `Cell`. For example, the type `Cell<mine|Car<yours>>` is shorthand for `Cell<mine|yours,Car<yours>>`.

Consider the `HashTable` class in Figure 6.5. This class takes two type parameters, one for the `Key` type and one for the `Data` type. The `Key` variable includes a bound to

[5] Personal Communication. Mark Skipper, Imperial College, London.

[6] We begin the name of a context parameters with a lowercase letter and type parameters with an uppercase letter.

```
class HashTable<owner | ko, Key extends Hashable<ko>, Data>
{
   class Slot ownedby rep {
       Key  key;
       Data data;
       Slot next;
   }

   Array<rep|Slot> buckets = new Array(107);

   void add(Key key, Data data) {
       Slot newslot = new Slot();
       newslot.key  = key;
       newslot.data = data;
       int hash     = key.hashcode() % 107;
       newslot.next = buckets.get(hash);
       buckets.set(hash, newslot);
   }
   ...
}

class Array<owner|X> { ... }

interface Hashable<owner>
{
   int hashcode();
   // ignoring that this is already present in
   // java.lang.Object
}
```

Figure 6.5: A Hashtable

indicate that it must implement the `Hashable` interface,and thus have a `hashcode()` method. The `Hashable` interface requires an owner parameter, thus we include the context parameter `ko` for the owner of `Key`. The header of this class entails a number of constraints, which will become apparent when we see its encoding, so we leave them for the moment. So too do we postpone the discussion of arrays, which are used to implement `HashTable`.

The `Slot` class has a manifest owner which is the `rep` of the `HashTable` class. Thus all of its instances are a part of the `HashTable`'s representation. The `HashTable`

is implemented as an array of `Slots`, which requires that the array is also a part of the `HashTable`'s representation, as indicated by the type `Array<rep|Slot>`.

In is interesting to note that ownership types are pleasingly unobtrusive in this example. They are in fact completely transparent in the implementation of the `add` method, which looks like ordinary Java or GJ code.

The expanded header of the class `HashTable` is encoded in our calculus as follows:

$$\begin{aligned} &\mathit{hashTableClass} \mathrel{\widehat{=}} \\ &\quad \lambda(\mathit{owner} \prec\!: \varepsilon, \mathit{ko} :\!\succ \mathit{owner}, \mathit{Key}<\!:\exists \mathit{Hashable}\texttt{<}\mathit{ko}\texttt{>}, \\ &\qquad \mathit{fresh} :\!\succ \mathit{owner}, \mathit{Data}<\!:\exists \mathit{Object}\texttt{<}\mathit{fresh}\texttt{>})[\mathit{new} = \cdots] \end{aligned}$$

where $\exists \mathit{Hashable}\texttt{<}p\texttt{>} \equiv \exists(\alpha \prec\!: \varepsilon)[\mathit{hashcode} : \mathit{Int}]^p_\alpha$ and $\exists \mathit{Object}\texttt{<}p\texttt{>} \equiv \exists(\alpha \prec\!: \varepsilon)[]^p_\alpha$. From this we can see all the constraints on the types and context parameters which can appear in the class's argument list.

Including F-bounded polymorphism [43], as used in Pizza [153] and GJ [34], rather than the style of genericity we use here does not present any additional challenges, since the changes required apply to types not contexts.

Arrays In the above example, we treated arrays as a generic class rather than use their usual Java syntax. One reason for this is that the generic instance syntax better accommodates the kind of information we require to be stated, namely, the owner of the array and the type of its contexts. Buckley [39] in his implementation of OOTS modified the usual syntax of arrays to include the parameters to a class, as in `Class<params>[]`. Unfortunately his syntax omits the owner of the array. The syntax we have adopted includes all the required information. For example, we can represent a multidimensional array type

```
Array<a | Array<b | Array<c | Car<o | d>>>>
```

which is owned by `a`, holds arrays owned by `b`, which hold arrays owned by `c`, which hold `Cars` owned by `o` with parameter `d`. It is not clear how best to adapt Java's array syntax to include all this information.

Comments Flexible Alias Protection [150] was phrased in terms of a programming language with generic classes. Context parameters (called *role parameters*) were attached to type parameters but could not be separated from them. Apart from being a

limitation, since it prevents context parameters being used on their own, the example above shows that it was unnecessary, at least in the programmer's view. Only the owner parameter is really needed for generic classes which have unbounded type parameters (Flexible Alias Protection did not have a notion of owner either). In OOTS, the owner parameter was implicit in a class definition, though it was made available through the keyword `owner` within the body of a class [61]. If we had adopted this instead of making the owner parameter explicit, then the class `Cell` above could have been written concisely as:

```
class Cell<X> {
    X item;
}
```

This is a valid GJ program, which also corresponds to a program in a class based language with ownership types. From this we can conclude that generic polymorphism taken seriously can further ease the burden of ownership types. More work is required to determine the degree.

6.3.10 `private-up-to`, revisited

As a final example we present three interpretations of the `private-up-to` construct which we introduced in Chapter 3. Recall the following nest of classes:

```
class A is public {
    class B is private-up-to A { }
    class C is private-up-to A {
        class D is private-up-to C { }
        class E is private-up-to A { }
        class F is public { }
    }
}
```

The `private-up-to` limits the visibility of objects from a particular class. For example, objects from class `D` are not visible outside of class `C`. We can encode this nest of classes in three different ways in the programming language we have been developing. Each encoding provides a different degree of protection.

Firstly, assume that among the constant context are the class names `A` through `F` with the ordering derived from their nesting. The nest of classes can then be encoded as:

```
class A ownedby norep {
    class B ownedby A { }
    class C ownedby A {
        class D ownedby C { }
        class E ownedby A { }
        class F ownedby norep { }
    }
}
```

This provides class-level protection and corresponds to the encoding presented in Section 4.7, except that here each class can have its own private representation.

The second implementation restricts access to each inner classes to within the appropriate *instance* of the enclosing named class (or `norep`).

```
class A ownedby norep {
    class B ownedby A.rep { }
    class C ownedby A.rep {
        class D ownedby C.rep { }
        class E ownedby A.rep { }
        class F ownedby norep { }
    }
}
```

For example, instances of `D` are confined to the appropriate surrounding instance of class `C`. Class `D` objects from a different instance of `C` are inaccessible and of incompatible type. Thus this example provides object-level protection, but the amount of protection is the same for all instances of a class due to the manifest ownership employed.

A third, more flexible approach is possible using bounded context parameters:

```
class A<ownerA> {
    class B<ownerB <: A.rep> { }
    class C<ownerC <: A.rep> {
        class D<ownerD <: C.rep> { }
        class E<ownerE <: A.rep> { }
        class F<ownerF> { }
    }
}
```

In this case the objects are contained as in the previous example, but the actual level of containment is defined per instance, rather than for all instances of a class. Thus some instances of a class can be more restricted, so long as they are not less so.

6.3.11 Discussion

After first describing our original ownership types system [61], which was phrased in terms of a class-based programming language, we discussed how its constructs can be encoded in the calculus presented in this thesis. From this we determined a number of constraints which classes and types must satisfy in order to support ownership types, placing us in a position to extend the original proposal in a number of directions. And this we did, describing how to add inner classes, inheritance and subtyping, static fields and methods, context polymorphic methods, friendly functions, and generic classes. We finished by presenting three different ways of implementing the `private-up-to` construct, to demonstrate the flexibility of our approach.

As presented here, ownership types are syntactically heavy. More syntactic shorthands are needed to reduce the burden on a programmer using ownership types. Extending Buckley's idea [39], we could mark method parameters whose ownership we don't care about as `anonymous`. Boyland's `borrowed` annotation [32] could indicate context polymorphic methods, to save having to write down all the context parameters. Type synonyms could also be employed to reduce the burden where complex types appear more than once. Determining a useful set of shorthands should depend more upon experience than speculation, so we stop the latter at this point.

6.4 Clone

The preservation of representation containment warrants a new clone operation, especially when we require the uniqueness of each object's representation context. Java's shallow clone breaks containment [93], whereas Eiffel's deep clone copies too much [138]. Here we describe an alternative clone operation which is a consequence of the containment of objects. It clones not only the top-level object, but also its representation and everything inside the representation, preserving references to objects external to the object originally being cloned. We also present a simple representation of this clone operation in the diagrammatic notation we have been using.

Consider the `List` object depicted in Figure 6.6(a). As usual, the `Link` objects are

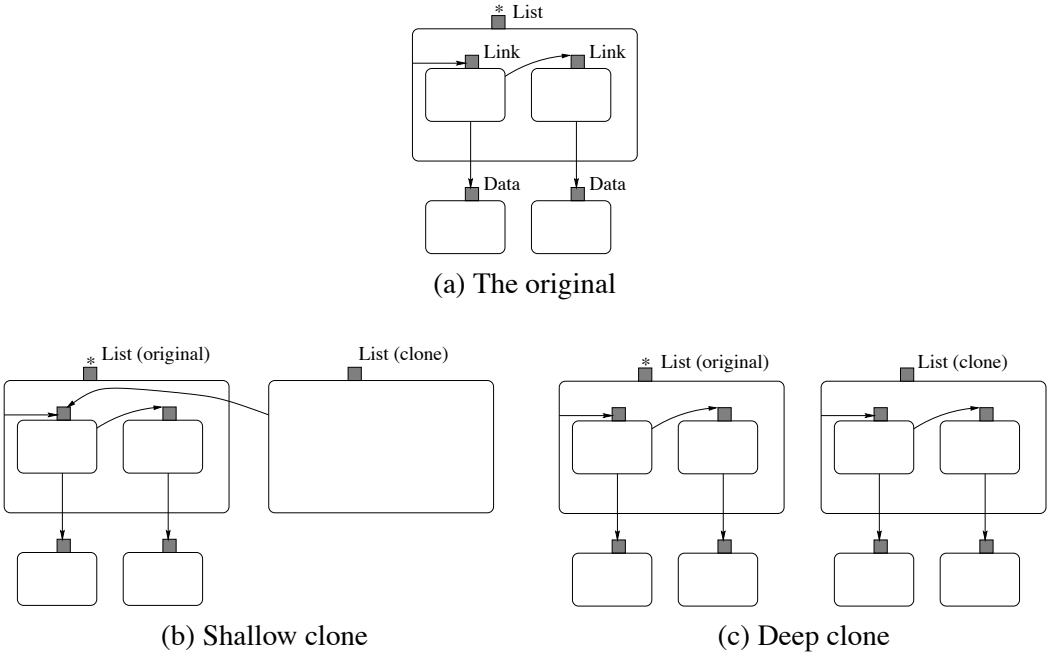


Figure 6.6: A slack shallow clone and an overzealous deep clone of a List.

a part of the representation, but the `Data` objects are not. We wish to clone the `List`.

At one extreme in the spectrum of possible clone operations is *shallow cloning*, which is the default clone provided by Java [93] and the object calculus [3]. This clone copies only the top-level of an object, preserving the values, including references, in the original object's fields. This introduces aliases from the new cloned object to objects referred to by the original object, including aliases to the original object's representation. The result of shallow cloning the `List` object is depicted in Figure 6.6(b). The clone of the `List` now has access to the original `List`'s `Link` objects, depicted by the arrow crossing the original `List`'s boundary from the outside to the inside. Hence, shallow copying is unsound from our perspective; it breaks encapsulation.

At the other extreme is *deep cloning*, such as the default copy provided by Eiffel [138]. This operation recursively duplicates the entire object graph reachable from the object being cloned, while preserving the underlying graph structure. The result of deep cloning the `List` is illustrated in Figure 6.6(c). This time not only is the `List` copied, but also all the `Data` stored in the `List`. This is probably too much copying, although this may depend upon the application. We shall see in a moment that when a `List`'s `Iterator` is deep cloned, the result is an `Iterator` to an entirely new `List`

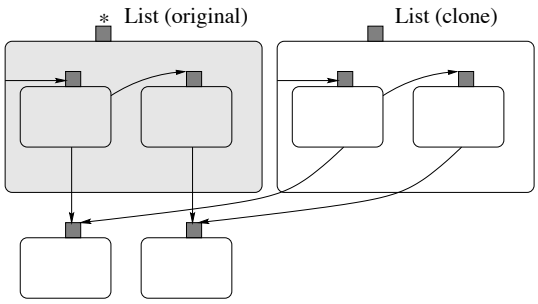


Figure 6.7: A sensible clone of a List.

with new `Data`. In this case, deep cloning is unsatisfactory as it overzealously copies too much.

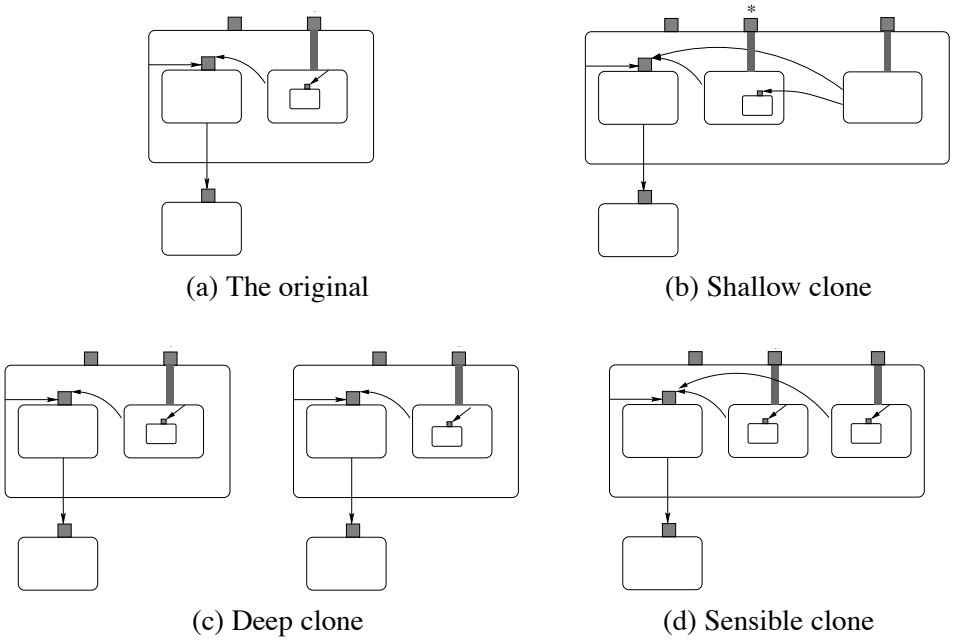
Neither clone produces a useful and safe copy for aggregate objects. The shallow clone exposes representation, and the deep clone simply copies too much to be useful. Writing special purpose clones for every class is not a satisfactory solution either, for two reasons: such operations are not always required since the default clone, whatever it may be, is often sufficient; and getting such a clone correct in the presence of representation containment may be nontrivial.

Fortunately, ownership and containment provide enough information to specify a new clone operation which copies the representation and all other internal objects, while preserving references to external objects. Pictorially, this clone operation has a simple representation, depicted in Figure 6.7. Everything within the rounded box denoting an object's representation is copied, as the shading indicates, whereas references to anything outside the box are preserved. Thus, references from inside to outside are kept intact, whereas internal objects are cloned, with their graph structure maintained. Here our clone copies the `List`'s representation (the shaded context), but nothing else. This clone operation copies the smallest collection of objects possible in order to preserve the containment invariant.

To determine which objects should be copied and which should not, we use the representation context of the object being cloned. To clone the object ι, we first define the two sets of objects:

$$\begin{aligned} \mathsf{copy}(\iota) &= \{\iota\} \cup \{\iota' \in \mathsf{dom}(\sigma) \mid \mathsf{owner}(\iota') \prec: \mathsf{rep}(\iota)\} \\ \mathsf{preserve}(\iota) &= \{\iota' \in \mathsf{dom}(\sigma) \mid \iota' \notin \mathsf{copy}(\iota)\}. \end{aligned}$$

Our clone copies the objects in the set $\mathsf{copy}(\iota)$, preserving references to the objects in set $\mathsf{preserve}(\iota)$, while maintaining the original object graph's structure. An im-



(a) The original (b) Shallow clone

(c) Deep clone (d) Sensible clone

Figure 6.8: Cloning a List Iterator.

plementation need not construct these sets, since only the reachable elements need be considered, but it must remember the representation context of the object being cloned, $\mathsf{rep}(\iota)$, and use the inside relation $\prec:$ to determine whether to copy or preserve each object encountered.

Let's return to the `List` example and add an `Iterator`, as indicated by the * in Figure 6.8(a). We wish to clone the `Iterator` to produce another `Iterator` for the same list. Figure 6.8(b) shows the effect of a shallow clone. This seems to do what we want, except that it violates the encapsulation of the `Iterator`'s representation. Figure 6.8(c) shows the effect of a deep clone. This clones too much; rather than producing a new `Iterator` for the original `List`, the result is a new `Iterator` for a copy of the original `List` with different `Data` objects. Figure 6.8(d) shows the result of our clone. It clones exactly the correct objects.

The clone[7] discussed here is a natural intermediate between shallow and deep cloning enabled by object ownership and containment. We have discussed a version of this clone elsewhere [148] in the context of prototype-based programming, where it was used to alleviate the *prototype corruption problem*.

[7]Because this lies somewhere between a shallow and a deep clone, we have been known to call this a *sheep* clone. We do not use this terminology here to avoid confusion with other sheep clones [42].

Encoding the clone in the calculus is impossible with current type-theoretic machinery. Because copying garbage collectors clone objects as a part of their operation, the technology underlying recent advances in type-preserving garbage collectors [194] may provide the appropriate machinery in which to specify a type-safe, containment-safe clone operation, which could then be demonstrated as sound.

6.5 Concluding Remarks

We have explored the expressiveness of our calculus through three subcalculi $\emptyset\varsigma$, **UR**ς, and **URDI**ς, where $\emptyset\varsigma$ is the calculus presented in Chapter 5. The calculus **UR**ς has the restriction that each object has a unique representation context, and **URDI**ς is restricted further so that each object's representation context is directly inside its owner context. The calculus **URDI**ς has the property that owners are dominators, as in our original ownership types system [61]. In addition, we hypothesised the existence of a fourth calculus **DI**ς which has the property that an object's representation context is directly inside its owner. This calculus would have the property that owners are cutsets, and may be worthy of further investigation.

We described a modelling problem our calculus exhibits, namely, *vampiric behaviour*. This means that our calculus can control only the form of the store, but not how stores are formed. We presented two ways of preventing vampiric behaviour: the most extreme is the calculus **URDI**ς which excludes all forms of vampiric behaviour, but also many desirable examples; the second approach was to use encodings for external method selection and update and to remove the **expose** term from the programmer's syntax.

We then gave a series of examples — protected objects, aggregate objects with multiple interfaces, linked data structures, friendly functions — and then discussed the properties of the **URDI**ς calculus in relation to garbage collection and the stack-based allocation of aggregate objects.

The type theory developed in Chapter 5 made it possible to add ownership types to a class-based language. Using this theory as a guide, we described the additions and constraints required for a compete, class-based programming language resembling GJ [34], a Java extension with generic types. Unfortunately, at this stage, no complete definition or implementation of this class-based language exists, although there are three incomplete implementations (ours, Bokowski's [23] and Buckley's [39]).

Finally, we described a new clone operation which ownership types warrant if

the containment invariant is to be preserved through cloning. We argued that this operation was a natural one. None of the implementations support this operation.

Chapter 7

Conclusion

7.1 Summary

Object (or alias) encapsulation is an essential ingredient in the object-oriented recipe. It tempers the bitter taste of aliasing, boosting our confidence when reasoning about object-oriented programs, facilitating the construction of more robust software. Unfortunately object encapsulation has received little attention, especially as a characteristic which can be specified within a programming language and enforced by a compiler.

We have presented here a type-theoretic account of ownership types which provides this ingredient. Firstly, we introduced the notion of *contexts* as the unit of ownership. These are associated with objects in two ways. Each object resides in a context (its owner context), and has a context which holds its protected objects (its representation context). Contexts are nested, generally, but not always, in a tree-shaped partial-order. The owner and representation contexts together with the partial-order are used to form a containment invariant which is central to our approach to enforcing object encapsulation.

Our first calculus and type system presumed a prespecified collection of contexts which are used to model objects encapsulated within packages or classes, for example. Our second system extended this with an operation to dynamically create contexts, allowing, for example, every object to have its own collection of private objects. In this system each object can be properly considered as an aggregate object.

After exhibiting the salient features of our calculus through many examples, we showed how it models our earlier work on an ownership types system for a Java-like programming language without inheritance. We then applied our greater understand-

ing to extend this earlier work to include inheritance and other features which were not previously possible. Thus we finish the thesis with an offering to programmers of object-oriented languages: enforceable object encapsulation in their favourite programming language.

7.2 Critique

This dissertation formalised and proved sound a type system for object encapsulation using mostly standard type theory. It was the first system to do so. Our type system was refined over many, many iterations to the point where the surface differences between our type system and type systems for Abadi and Cardelli's object calculus became minimal. This makes our type system more comprehensible to those who understand such type systems, and hopefully more readily adaptable to other settings. By using standard type theory, we were able to use established techniques such as the dot notation and (context) polymorphic methods, as we did in Chapter 6.

An unfortunate consequence of the amount of time spent producing a sound and concise type system is that this thesis did not advance as far as we would have liked. In particular, no complete implementation of the class-based language discussed in Chapter 6 has been made, and thus we have little experience with using ownership types. We comfort ourselves a little by adopting the view that by developing the type system and demonstrating its soundness before implementing a programming language which uses that type system, we are, in a sense, putting the horse before the cart.

The type system provides a framework for exploring various containment models. Three were investigated and another hypothesised. Unfortunately, since the containment model is embedded in the type rules, each required a specialised modification to the type system or the syntax of the calculus. This was not ideal. In hindsight, a type system parameterised by the aspects of the containment model would have been better and more flexible than the approach taken. This could have allowed more containment models to be explored. The containment models we presented in part stemmed from a study of object graphs [163]. Some are too weak and others too strong. Since our containment models are restrictions on object graphs, a type system parameterised by containment model would have facilitated the independent development of containment models, and consequently we may have found some useful middle ground.

This dissertation fell short of proving all the properties we believe that certain

ownership types systems exhibit, namely those relating to garbage collection (Section 6.2.10), the stack-based allocation of objects (Section 6.2.11), and borrowing (Section 6.2.12). In a sense, the first two are obvious from the owners-as-dominators property, but borrowing, which has an intuitively clear meaning, is difficult to formalise.

It is worth considering the appropriateness of the formal framework, namely Abadi and Cardelli's object calculus [3], chosen as the basis for our work. On the downside, it is more likely that our research would make an immediate impact if it had been phrased in terms of a class-based programming language. But, when formalising concepts in a class-based language, especially initially, one must deal with unsavory aspects such as inheritance and overriding.[1] But with our work, classes, inheritance, and overloading are orthogonal to the issue at hand. Object encapsulation is an object-level concept which is best studied in an object calculus. An advantage of Abadi and Cardelli's object calculus is that, in its simplest form, the only construct is the object. Thus we have as small as possible number of features to concern ourselves with. Other formal calculi are not so trim. The object calculus is expressive enough to encode the features of class-based programming languages; we have done so here incorporating our extensions. Finally, because the object calculus includes method update as one of its basic operations, we can provide object encapsulation for prototype-based programming languages as a by-product of our work [148]. For these reasons we feel that we chose the appropriate formal framework.

The thesis could have gone further. We now discuss some of the possible directions.

7.3 Future Directions

The work presented in this thesis can be taken in a number of directions. Firstly we discuss two recent revelations which ought to contribute to a better system of ownership types, before discussing more long term possibilities.

[1] "The history of all hitherto existing society is the history of class struggles." [135].

Eliminating Existentials

It is possible to remove existential types from the system presented in Chapter 5 by adopting the following rule for object subtyping:

$$\frac{\begin{array}{c}\text{(Sub Object)}\ \ (l_i \text{ distinct}) \\ E;\langle q\uparrow\rangle \vdash \Theta_i\ meth \quad \forall i \in 1..n \quad E;\langle q'\uparrow\rangle \vdash \Theta_i\ meth \quad \forall i \in n{+}1..n{+}m \\ E \vdash q' \prec: q \quad E \vdash q \prec: p\end{array}}{E;\langle p\rangle \vdash [l_i : \Theta_i{}^{i\in 1..n+m}]^p_{q'} <: [l_i : \Theta_i{}^{i\in 1..n}]^p_q}$$

The type $[l_i : \Theta_i{}^{i\in 1..n}]^p_p$, which can result using this rule, must contain no mention of any method which has the original representation context as a part of its type. Under this interpretation, the fields containing representation and the methods which take representation as an argument or return representation are private, and subsumption removes such fields and methods from the external view of the object (as in [171]). Such a type can be considered to be the external interface type of an object, since it contains only methods which are accessible to all objects which have permission to access the owner context p.

The new rule comes into play when using the type rule for **new**, which we replicate here:

$$\frac{\begin{array}{c}\text{(Val New)} \\ E, \alpha \prec: p; K \cup \langle\alpha\rangle \vdash a : A \quad E; K \vdash A\end{array}}{E; K \vdash \textbf{new}\ \alpha \lhd p\ \textbf{in}\ a : A}$$

This rule is generally used to create a new representation context for an object. Previously a would be an object wrapped with an existential type so that the new context α does not appear in the type. Using the type rule above, we can use subsumption to remove α from the object's type and achieve the desired constraint.

Unfortunately, this approach has two drawbacks. Firstly, eliminating existentials makes it impossible to encode friendly functions directly, as outlined in Chapter 6, since these relied on existential types to abstractly access the name of the hidden representation context. Other encodings are possible using context parameters to carry the representation contexts and double dispatch to bring both representation contexts into the same method body [36]. Secondly, this subtype rule may interact badly with the subtyping of recursive types.

A paper describing this extension in a version of the calculus from Chapter 4 will soon appear [60].

Monotonic Ownership and Wooden Stakes

A recent discussion with Sophia Drossopoulou revealed a constraint which can be added to the containment model to eliminate the worst kind of vampires (Section 6.2.1), while still allowing multiple objects to access the same representation. The additional requirement is simply stated:

$$\mathsf{rep}(\iota) \prec: \mathsf{rep}(\iota') \Rightarrow \mathsf{owner}(\iota) \prec: \mathsf{owner}(\iota')$$

When rep is 1:1, this states that owner is monotonic.[2] This constraint, combined with the containment invariant, can be interpreted as stating:

- any object that ι' can access, ι can also access;
- no object that can access ι' can access ι; and
- all objects ι with representation context $\mathsf{rep}(\iota)$ such that $\mathsf{rep}(\iota) \prec: \mathsf{rep}(\iota')$ can properly be considered as a part of the aggregate ι'. Those for which $\mathsf{owner}(\iota)$ is strictly inside $\mathsf{owner}(\iota')$ can be considered to be a part of the aggregate's implementation, whereas those for which $\mathsf{owner}(\iota) = \mathsf{owner}(\iota')$ can be considered as a part of the aggregate's interface. All objects in the latter collection are accessible to the same collection of objects, namely, those which can access the aggregate.

Incorporating this requirement into the type system will require a little effort, but it will not be insurmountable. The soundness proofs will require similarly minor modifications.

Implementation and Experience

We need to develop a complete implementation of the class-based language described in Chapter 6. We will then need to develop some non-trivial applications to determine the benefits and detriments of programming with ownership types. As a part of this process, we will discover idioms or design patterns which programmers may find useful. The experience gained could then be used to reduce the complexity of the ownership types system to its most useful elements. If we have no implementation, then we cannot evaluate ownership types in practice. If we have no practice, we cannot judge its true worth.

[2] That is, $\iota \prec: \iota' \Rightarrow \mathsf{owner}(\iota) \prec: \mathsf{owner}(\iota')$.

Cloning

A part of the above exercise would be implementing the clone operation described at the end of Chapter 6. This operation clones only the interior of the object being cloned, while preserving references to external objects. This raises a few questions which need answers:

- Can the clone operation be implemented without carrying around ownership information at run-time? There may be enough information present at compile time for class-based languages to avoid this expense.
- It the resulting operation a useful default? How does it compare to the clone operation Grogono and Sakkinen describe [96]?
- Can it be integrated with user-defined clone operations? How?

Lastly, it would be instructive to specify and prove the correctness of the clone operation using the advanced type-theoretic techniques such as those used to specify type-preserving garbage collectors [194].

Garbage Collection and the Stack-based Allocation of Objects

The original ownership types system and the **URDI**ς calculus both enjoy the owners-as-dominators property. From this it follows that an object and its entire interior can be deleted when the object becomes garbage, since no references into its interior can exist (Section 6.2.10). In addition, a minor extension to the calculus or programming language allows contexts whose lifetimes are limited to lexical scopes, such as the method bodies. This extension allows behaviour similar to that of the `letregion` term from the regions calculus [188], and hence the stack-based allocation of entire aggregate objects, as discussed in Section 6.2.11. Work is required to show that garbage collection and the stack-based allocation of objects are in fact safe and to incorporate them into the implementation to determine whether they offer any benefit in practice.

It is worth mentioning that the stack-based allocation of objects resembles the intended semantics of scoped memories in Real-time Java [25]. The main difference is that we see types as controlling the allocation of objects to different regions, whereas in Real-time Java a scoped memory is just another object. Real-time Java suffers from the problem that the scoped memory in which an object resides is not a part of the object's type, and thus the entire system can easily be violated. Either run-time

checks or an *ad hoc* program analysis would be required to ensure safety in Real-time Java. The type systems presented in this thesis provide a grounding for doing this properly, safely, and statically, without requiring run-time checks.

Reasoning in the Presence of Ownership

Since ownership types can encapsulate an object's implementation and thus reduce the amount of aliasing, they should make it easier to reason about objects. Informally, it means that one does not always have to search possibly the entire program to find the aliases to certain objects. It would be nice to extend a formal system for reasoning about object with the concepts underlying object ownership, and to see whether reasoning is really simplified, as our intuition suggests it would be.

Semi-automatic Inference of Ownership

The most important information an ownership type specifies is which objects are a part of an object's representation. Most of the other annotations allows the representation to be used more flexibly, while preserving its integrity. Unfortunately these additional annotations make ownership types quite syntactically heavy. In Chapter 6 we suggested a number of shortcuts which reduce this burden, but these only go part of the way. Ideally, we would like to specify only which objects are representation — that is have only the keyword `rep` — and let the compiler use static analysis to determine whether the representation really is protected. This is probably very difficult to do in a modular fashion and limited by the use of abstract classes and interfaces where the implementation is deferred.

More Sophisticated Type Systems

The type systems presented in this thesis could be extended in a number of directions. Types could be made to further control the properties of objects or to make stronger statements about an object's behaviour. This actually raises the question of whether our types should constrain the behaviour of an object or to report the behaviour an object exhibits or both. The choice can greatly affect the definition of subtyping. When subsumption preserves the properties of an expression's behaviour stated in the types, then the type system is said to exhibit behavioural subtyping [131]. Effects systems fall into this category, because subtyping can introduce more information about the potential behaviour, but never forget any.

For example, normal subtyping would take an object which has multiple interfaces and reduce its type to one which no additional interfaces can be created — externally. Behavioural subtyping would state that an object has only a single interface, but allow that to be used where one which may have multiple interfaces is expected.

Behavioural subtyping assists with the reasoning about programs because the properties of objects are included in the type and cannot be forgotten by subsumption.

Possible dimensions for extension include:

Full, Flexible, or Fractal? It would be nice to allow full, flexible, and fractal alias encapsulation into the one type system (Section 2.5). This would be similar to merging the calculi **UR**ς and **URDI**ς, while imposing some additional restrictions. The result would allow tighter control over the creation of objects, as well as facilitating better memory management properties. From a behavioural perspective, knowing whether an object has been implemented using full, flexible or fractal alias protection we could determine, respectively, that there is a single entry point to its representation and no outgoing references, that there is a single entry point but outgoing references may exist, or that the object may have multiple interfaces. This information helps determine the extent to which the remainder of a program can affect the value of an object. Subtyping could then have to be designed to ensure that these properties are preserved (ordinary subtyping), but possibly also ensure that when a type states that an object is fully alias encapsulated, for example, that it really is (behavioural subtyping).

Effects Combining an ownership types system with the effects system of Greenhouse and Boyland [94] or JAC [122], for example, would enhance the expressive power of ownership types and enable more control over behaviour in object-oriented programs. Flexible Alias Protection's `arg` mode for reducing the dependence on mutable state [150] could also be added, giving the benefit of immutability while reducing the dependence between an aggregate and the objects which are external to it.

Borrowing In certain circumstances, context parameters on methods can be used to implement the notion of borrowing. Context polymorphic methods can guarantee that references to objects owned by that parameter are not captured within the method body. This was only possible in the strictest type system, for cal-

culus **URDI**ς, but it would be useful to be able to incorporate borrowing into other versions of the type system.

Borrowing gels well with the stack-based allocation of objects. Borrowed objects could be stored in objects whose lifetime is the duration of the method body. A key difficulty then is defining the type system in such a way that the desired properties can be readily demonstrated.

Disjointness Two different context parameters of a method can be bound to the same context. Thus the context parameters do not say anything about the disjointness of the contexts. This means that the aliasing properties which can be inferred within a method body are weaker than we would like; certain non-interference properties cannot be assumed. The type system could be extended to state that different context parameters refer to distinct contexts. These parameters could then never be bound to the same context. This extension would strengthen the aliasing properties one could assert. The Calculus of Capabilities [66] has this feature.

Leavens and Antropova [124] take a completely different approach to dealing with disjointness. They suggest that different implementations of a method be given, each depending on the amount of aliasing between the method's arguments. These methods and their properties can be specified independently. The result is that the specification of each method is simpler because certain aliasing properties are guaranteed. Dynamic dispatch is then used to call the appropriate method based on the aliasing actually present between the method's arguments.

When some form of disjointness specification is added to our ownership types system, we then are able to assert not only that a method's arguments are not aliased, but also that their internal representation are completely disjoint.

Partitioned and Scoped Memory: Names and Nomenclature

The notions of partitioned memory and scoped memory (and similar concepts) appear elsewhere in the literature, in each case with slightly different names: arenas [99], collections [74], contexts and owners [61], contours [56], data groups [125], demneses [196], first-class stores [142], groups [114], local stores [192], object spaces [38], regions [188], roles [150], sandwiches [88], scoped memory [25], universes [144] and zones [173]. The idea is of paramount importance and deserves a uniform treatment.

Underlying most, but not all, of these concepts is the idea that fresh store partitions can be created as a program evaluates. It seems that a suitable starting point could be *names* and *name restriction*, as used in what Gordon calls nominal calculi [91] — as in the π [140], spi [1], Ambient [50], **conc**ς [92], etc. calculi, for example. Relevant also is some of Cardelli, Ghelli, Gordon, and Dal Zilio's recent work using name restriction to create fresh types, called *groups* [49, 48, 201], as is the work on the so-called *fresh name* quantifier [83] which can be used to reason about name restriction [51]. How exactly to apply this work on names to the notions of partitioned and scoped memory stores is not yet clear. We believe that it does, however, deserve future investigation.

Appendix A

Proofs from Chapter 5

This chapter presents the proofs of the main results from Chapter 5. The general flow of results may be of interest to the reader, although in general the path taken and the techniques employed are standard, modulo some interesting lemmas. Preliminary results and definitions, often standard, are also presented and proved as required.

A.1 Preliminary Lemmas

Substitution of values for term variables, contexts for context variables, and types for type variables can be defined in a straightforward and standard manner based on the definition of free variables given in Chapter 5 [105, 3, for example]. Thus the definition will not be given here. We do, however, define substitution into an environment (which is also standard):

Definition A.1 (Environment Substitution)

$$
\begin{array}{rcl}
\emptyset\{{}^{B}/_{Y}\} & \widehat{=} & \emptyset \\
(x : A, \Gamma)\{{}^{B}/_{Y}\} & \widehat{=} & x : A\{{}^{B}/_{Y}\}, \Gamma\{{}^{B}/_{Y}\} \\
(\iota : A, \Gamma)\{{}^{B}/_{Y}\} & \widehat{=} & \iota : A\{{}^{B}/_{Y}\}, \Gamma\{{}^{B}/_{Y}\} \\
(X <: A, \Gamma)\{{}^{B}/_{Y}\} & \widehat{=} & X <: A\{{}^{B}/_{Y}\}, \Gamma\{{}^{B}/_{Y}\} \\
(\alpha \prec: p, \Gamma)\{{}^{B}/_{Y}\} & \widehat{=} & \alpha \prec: p, \Gamma\{{}^{B}/_{Y}\} \\
(\alpha :\succ p, \Gamma)\{{}^{B}/_{Y}\} & \widehat{=} & \alpha :\succ p, \Gamma\{{}^{B}/_{Y}\} \\
\emptyset\{{}^{q}/_{\beta}\} & \widehat{=} & \emptyset \\
(x : A, \Gamma)\{{}^{q}/_{\beta}\} & \widehat{=} & x : A\{{}^{q}/_{\beta}\}, \Gamma\{{}^{q}/_{\beta}\}
\end{array}
$$

$$
\begin{array}{rcl}
(\iota : A, \Gamma)\{{}^{q}/_{\beta}\} & \widehat{=} & \iota : A\{{}^{q}/_{\beta}\}, \Gamma\{{}^{q}/_{\beta}\} \\
(X <: A, \Gamma)\{{}^{q}/_{\beta}\} & \widehat{=} & X <: A\{{}^{q}/_{\beta}\}, \Gamma\{{}^{q}/_{\beta}\} \\
(\alpha \prec: p, \Gamma)\{{}^{q}/_{\beta}\} & \widehat{=} & \alpha \prec: p\{{}^{q}/_{\beta}\}, \Gamma\{{}^{q}/_{\beta}\} \\
(\alpha :\succ p, \Gamma)\{{}^{q}/_{\beta}\} & \widehat{=} & \alpha :\succ p\{{}^{q}/_{\beta}\}, \Gamma\{{}^{q}/_{\beta}\}
\end{array}
$$

The following lemma states that the components of each valid judgement are also well-formed in the appropriate sense:

Lemma A.2

1. *If* $E; K \vdash a : A$, *then* $E; K \vdash A$.
2. *If* $E; K \vdash (\Theta)(\Delta) \Rightarrow C$, *then* $E; K \vdash C$.
3. *If* $E; K \vdash A <: B$, *then* $E; K \vdash A$ *and* $E; K \vdash B$.
4. *If* $E; K \vdash A$, *then* $E \vdash K$.
5. *If* $E \vdash K \subseteq K'$, *then* $E \vdash K$ *and* $E \vdash K'$.
6. *If* $E \vdash K$, *then* $E \vdash \diamond$.

PROOF: Straightforward induction on the appropriate derivations. □

In the following lemmas the meta-variable $\Im$ corresponds to right-hand side of the turnstile for some judgement. $\Im\{{}^{v}/_{x}\}$ denotes substituting value v for x in whatever expressions appear in $\Im$; substitutions for other kinds of variable are denoted similarly. The meta-variable K refers to a permission K' whenever permissions form part of the judgement, otherwise it denotes nothing. That is, in the judgement $E; K' \vdash A$, K can be used to refer to K'; in judgement $E \vdash \diamond$, K refers to nothing. $K\{{}^{p}/_{\alpha}\}$ is substitution, as expected. Note that these lemmas do not apply to the well-formed configuration and store judgements, since substitution never occurs over configurations and entire stores.

Lemma A.3 *If* $E; K \vdash \Im$, $E' \gg E$ *and* $E' \vdash \diamond$, *then* $E'; K \vdash \Im$.

Lemma A.4 (Bound Weakening)

1. *If* $E, x : B, E'; K \vdash \Im$ *and* $E; K' \vdash B' <: B$, *then* $E, x : B', E'; K \vdash \Im$.
2. *If* $E, \alpha \prec: p, E'; K \vdash \Im$ *and* $E \vdash p' \prec: p$, *then* $E, \alpha \prec: p', E'; K \vdash \Im$.

3. *If* $E, X<:B, E'; K \vdash \Im$ *and* $E; K \vdash A<:B$, *then* $E, X<:A, E'; K \vdash \Im$.

PROOF: This is proven by straightforward induction over the appropriate derivations. The proof depends upon the Permissibility Lemma, which in turn depends upon this lemma. This is not problematic because clause 3 of this Lemma depends on clause 2 of the Permissibility Lemma which depends on clause 2 of this Lemma, and there the dependency stops. □

Lemma A.5 (Substitution)

1. *If* $E, \alpha \prec: p, E'; K \vdash \Im$ *and* $E \vdash q \prec: p$, *then* $E, E'\{{}^{q}/_{\alpha}\}; K\{{}^{q}/_{\alpha}\} \vdash \Im\{{}^{q}/_{\alpha}\}$.

2. *If* $E, \alpha :\succ p, E'; K \vdash \Im$ *and* $E \vdash p \prec: q$, *then* $E, E'\{{}^{q}/_{\alpha}\}; K\{{}^{q}/_{\alpha}\} \vdash \Im\{{}^{q}/_{\alpha}\}$.

3. *If* $E, X<:A, E'; K \vdash \Im$ *and* $E; K \vdash A'<:A$, *then* $E, E'\{{}^{A'}/_{X}\}; K \vdash \Im\{{}^{A'}/_{X}\}$.

4. *If* $E, x : A, E'; K \vdash \Im$ *and* $E; K \vdash v : A$, *for value* v, *then* $E, E'; K \vdash \Im\{{}^{v}/_{x}\}$.

PROOF: Straightforward induction on typing derivations. □

The Permissibility Lemma, proven in the next section, accounts for the apparent lack of relationship between K and K in clauses 3 and 4. Access to the value being substituted is guaranteed, because access to the variable of the appropriate type requires the same permission. Access is guaranteed because access to the type variable is governed by its bound, which requires at least as much permission to access as the type substituted.

Remark A.6 *It is interesting to point out that although the proof of this lemma is extremely straightforward, proving its counterpart for a preliminary version of the calculi presented in this thesis revealed the unsoundness of that calculus, even though the type preservation proof went through. The failed proof revealed the interaction between between substitution and subtyping which necessitated the use of permissions in typing and subtyping judgements.*

A.2 Proof of Lemma 5.8

The Permissibility Lemma is the key to soundness. It states the requirements for passing values between expressions and objects.

$$\frac{X{<:}A \in E \quad E;K \vdash_{\mathsf{min}} A}{E;K \vdash_{\mathsf{min}} X}\ (\text{Type-min } X) \qquad \frac{E \vdash K}{E;K \vdash_{\mathsf{min}} \mathsf{Top}^K}\ (\text{Type-min Top})$$

$$\frac{E;\langle q\uparrow\rangle \vdash \Theta_i\ meth \qquad \forall i \in 1..n \quad E \vdash q \prec: p}{E;\langle p\rangle \vdash_{\mathsf{min}} [l_i : \Theta_i{}^{i\in 1..n}]_q^p}\ (\text{Type-min Object})\ (l_i \text{ distinct})$$

$$\frac{E, X{<:}\mathsf{Top}^K; K \vdash_{\mathsf{min}} A}{E;K \vdash_{\mathsf{min}} \mu(X)A}\ (\text{Type-min Rec}) \qquad \frac{E, \alpha \prec: p; K \vdash_{\mathsf{min}} B \quad E \vdash K}{E;K \vdash_{\mathsf{min}} \exists(\alpha \prec: p)B}\ (\text{Type-min Exists})$$

Figure A.1: Minimal Permission Typing

Our proof of this lemma requires the additional notion of the minimum permission required for a type's validity. Figure A.1 defines this notion via the judgement $E;K \vdash_{\mathsf{min}} A$. The type rules simply disallow the use of (Type Allow), except in obtaining method typings.

The following Lemma states that there will always be a permission which satisfies the above judgement and that it is indeed the minimum.

Lemma A.7 (Minimum Permission) *If $E;K' \vdash A$, then there exists a K such that $E;K \vdash_{min} A$, where $E \vdash K \subseteq K'$. Furthermore, if $E;K^\dagger \vdash_{min} A$, then $K \equiv K^\dagger$.*

PROOF: Straightforward induction over the derivation of $E;K' \vdash A$. □

The Permissibility Lemma can now be proven:

Lemma A.8 (Permissibility (Restated))

1. *If $E;K \vdash v : A$ and $E;K' \vdash A$, then $E;K' \vdash v : A$, where v is a value.*
2. *If $E;K \vdash A{<:}B$ and $E;K' \vdash B$, then $E;K' \vdash A{<:}B$.*

PROOF: 1. Proof is by induction over the derivation of $E;K \vdash v : A$.

CASE: (Val x) Assume that $E;K \vdash x : A$. By (Val x), $x : A \in E$ and $E;K \vdash A$. Assume further that $E;K' \vdash A$. By (Val x), $E;K' \vdash x : A$.

CASE: (Val Location) Assume that $E;\langle p\rangle \vdash \iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q$. By (Val Location), $E',\iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q, E'' \vdash \diamond$. By (Env Location), $E';\langle p\rangle \vdash [l_i : \Theta_i{}^{i\in 1..n}]^p_q$. By Lemma A.3, $E;\langle p\rangle \vdash [l_i : \Theta_i{}^{i\in 1..n}]^p_q$. Assume that $E;K \vdash [l_i : \Theta_i{}^{i\in 1..n}]^p_q$. By (Type Allow), $E \vdash \langle p\rangle \subseteq K$. Hence, by (Val Subsumption), $E;K \vdash \iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q$.

CASE: (Val Fold) Assume that $E;K \vdash \mathbf{fold}(A,v) : A$, where $A \equiv \mu(X)B$. By (Val Fold), $E;K \vdash v : B\{{}^{A}/_{X}\}$. Assume that $E;K' \vdash \mu(X)B$. By (Type Rec), $E, X{<:}\mathsf{Top}^{K'}; K' \vdash B$ and by (Sub Top) $E;K' \vdash \mu(X)B{<:}\mathsf{Top}^{K'}$, hence by Lemma A.5, $E;K' \vdash B\{{}^{A}/_{X}\}$. By the induction hypothesis, $E;K' \vdash v : B\{{}^{A}/_{X}\}$. Hence, by (Val Fold), $E;K' \vdash \mathbf{fold}(A,v) : A$.

CASE: (Val Hide) Assume that $E;K \vdash \mathbf{hide}\ p\ \mathbf{as}\ \alpha \prec: q\ \mathbf{in}\ v{:}A : \exists(\alpha \prec: q)A$. By (Val Hide), $E \vdash p \prec: q$, $E;K \vdash v\{{}^{p}/_{\alpha}\} : A\{{}^{p}/_{\alpha}\}$, and $E;K \vdash \exists(\alpha \prec: q)A$. Assume that $E;K' \vdash \exists(\alpha \prec: q)A$. By (Type Exists), $E,\alpha \prec: q; K' \vdash A$ and $E \vdash K'$. By Lemma A.5, $E;K' \vdash A\{{}^{p}/_{\alpha}\}$, observing that $\alpha \notin FV(K')$. Therefore, by the induction hypothesis, $E;K' \vdash v\{{}^{p}/_{\alpha}\} : A\{{}^{p}/_{\alpha}\}$. Hence, by (Val Hide), $E;K' \vdash \mathbf{hide}\ p\ \mathbf{as}\ \alpha \prec: q\ \mathbf{in}\ v{:}A : \exists(\alpha \prec: q)A$.

CASE: (Val Subsumption) Assume $E;K \vdash v : A$. By (Val Subsumption), $E;K^\dagger \vdash v : A^\dagger$, $E;K \vdash A^\dagger{<:}A$, and $E \vdash K^\dagger \subseteq K$. Assume that $E;K' \vdash A$. By the induction hypothesis(2), $E;K' \vdash A^\dagger{<:}A$. By Lemma A.2, $E;K^\dagger \vdash A^\dagger$. By Lemma A.7, there exists a $K^\ddagger$ such that $E;K^\ddagger \vdash_{\mathsf{min}} A^\dagger$, where $E \vdash K^\ddagger \subseteq K^\dagger$. By the induction hypothesis, $E;K^\ddagger \vdash v : A^\dagger$. Again by Lemma A.2, $E;K' \vdash A^\dagger$, and by Lemma A.7, $E \vdash K^\ddagger \subseteq K'$. Hence by (Val Subsumption), $E;K' \vdash v : A$.

PROOF: 2. We prove a more general result. Rather than take any K' such that $E;K' \vdash B$, we instead choose the minimum permission, that is $K^\dagger$ which satisfies $E;K^\dagger \vdash_{\mathsf{min}} B$. It follows then that $E \vdash K^\dagger \subseteq K'$, and by suitable application of (Sub Allow), we can obtain the desired result. Proof is by induction over the derivation of $E;K \vdash A{<:}B$.

CASE: (Sub Refl), (Sub Trans), (Sub X), (Sub Top), and (Sub Object). Straightforward.

CASE: (Sub Rec) Assume that $E;K \vdash \mu(X)A{<:}\mu(Y)B$. By (Sub Rec), $E;K \vdash \mu(X)A$, $E;K \vdash \mu(Y)B$, and $E, Y{<:}\mathsf{Top}^K, X{<:}Y; K \vdash A{<:}B$.
Assume that $E;K^\dagger \vdash_{\mathsf{min}} \mu(Y)B$. By (Type-min Rec), $E, Y{<:}\mathsf{Top}^{K^\dagger}; K^\dagger \vdash_{\mathsf{min}} B$. By Lemma A.3, $E, Y{<:}\mathsf{Top}^{K^\dagger}, X{<:}Y; K^\dagger \vdash B$. By the induction hypothesis, $E, Y{<:}\mathsf{Top}^{K^\dagger}, X{<:}Y; K^\dagger \vdash A{<:}B$. By (Sub Rec), $E;K^\dagger \vdash \mu(X)A{<:}\mu(Y)B$.

CASE: (Sub Exists) Assume that $E;K \vdash \exists(\alpha \prec: p)A{<:}\exists(\alpha \prec: p')A'$.
By (Sub Exists), $E \vdash p \prec: p'$, $E,\alpha \prec: p; K \vdash A{<:}A'$, and $E \vdash K$. Assume

that $E;K^\dagger \vdash_{\mathsf{min}} \exists(\alpha \prec: p')A'$. By (Type-min Exists), $E,\alpha \prec: p';K^\dagger \vdash_{\mathsf{min}} A'$, and $E \vdash K^\dagger$. By Lemma A.4, $E,\alpha \prec: p;K^\dagger \vdash A'$. By the induction hypothesis, $E,\alpha \prec: p;K^\dagger \vdash A<:A'$. Hence, by (Sub Exists), $E;K^\dagger \vdash \exists(\alpha \prec: p)A<:\exists(\alpha \prec: p')A'$.

CASE: (Sub Allow) Assume that $E;K \vdash A<:B$. By (Sub Allow), $E;K' \vdash A<:B$ and $E \vdash K' \subseteq K$. Assume that $E;K^\dagger \vdash_{\mathsf{min}} B$. Hence, by the induction hypothesis, $E;K^\dagger \vdash A<:B$. □

A.3 Proof of Lemma 5.10

The proof that type preservation holds depends upon some additional lemmas and definitions.

Lemma A.9 shows that $\lfloor\Gamma\rfloor_C$ commutes with substitution, in other words, that the definition is stable in a very important sense.

Lemma A.9 *If* $\lfloor\Gamma\rfloor_C \equiv \Theta$ *then,*

1. $\lfloor\Gamma\{{}^{B}/_{X}\}\rfloor_{C\{{}^{B}/_{X}\}} \equiv \Theta\{{}^{B}/_{X}\}$ *and*
2. $\lfloor\Gamma\{{}^{q}/_{\alpha}\}\rfloor_{C\{{}^{q}/_{\alpha}\}} \equiv \Theta\{{}^{q}/_{\alpha}\}$.

PROOF: Induction on the structure of Γ.

To prove type preservation for method selection, we need some additional machinery. Firstly we extend the definition of $\lceil\_\rceil$, the function which converts a collection of formal parameters, to apply to the collection of actual parameters Δ.

Definition A.10 ($\lceil\Delta\rceil$)

$$\begin{aligned}
\lceil\emptyset\rceil &\mathrel{\widehat{=}} \mathit{void}\\
\lceil\vec{v},\Delta\rceil &\mathrel{\widehat{=}} \lceil\Delta\rceil\\
\lceil A,\Delta\rceil &\mathrel{\widehat{=}} \lceil\Delta\rceil\\
\lceil p,\Delta\rceil &\mathrel{\widehat{=}} \langle p\rangle \cup \lceil\Delta\rceil
\end{aligned}$$

The following lemma states that substituting appropriate actual parameters for the formal parameters of a method body gives an expression of the expected type. ($\{\!\{\Delta/\Gamma\}\!\}_E$ is defined in Definition 5.6.)

Lemma A.11 *If* $E,\Gamma;K\cup\lceil\Gamma\rceil\vdash b:B$ *and* $E;K'\vdash(\Theta)(\Delta)\Rightarrow C$, *where* $\Theta\equiv\lfloor\Gamma\rfloor_B$, *then* $C\equiv B\{\!|\Delta/\Gamma|\!\}_E$ *and* $E;K\cup\lceil\Delta\rceil\vdash b\{\!|\Delta/\Gamma|\!\}_E:C$.

PROOF: Proof is by induction over the structure of Γ.

CASE: ($\Gamma\equiv\emptyset$) Straightforward.

CASE: ($\Gamma\equiv x:A,\Gamma'$)

1. ASSUME: $E,x:A,\Gamma';K\cup\lceil x:A,\Gamma'\rceil\vdash b:B$ and $E,K'\vdash(\Theta)(\Delta)\Rightarrow C$, where $\Theta\equiv\lfloor x:A,\Gamma'\rfloor_B$.
2. By the definition of $\lfloor\ \rfloor$, $\Theta\equiv A\to\Theta'$, where $\Theta'\equiv\lfloor\Gamma'\rfloor_B$.
3. By (Arg Val), $\Delta\equiv v,\Delta'$, $E;K'\vdash v:A$, and $E;K'\vdash(\Theta')(\Delta')\Rightarrow C$.
4. Note also that $\lceil x:A,\Gamma'\rceil\equiv\lceil\Gamma'\rceil$, and similarly that $\lceil v,\Delta'\rceil\equiv\lceil\Delta'\rceil$.
5. By 1, 3, and Lemma A.5, $E,\Gamma';K\cup\lceil\Gamma'\rceil\vdash b\{\!|{}^{v}/_{x}|\!\}:B$.
6. By 2, 3, 4, 5, and the induction hypothesis, $C\equiv B\{\!|\Delta'/\Gamma'|\!\}_E$ and $E;K\cup\lceil\Delta'\rceil\vdash b\{\!|{}^{v}/_{x}|\!\}\{\!|\Delta'/\Gamma'|\!\}_E:C$, from which we obtain the desired result, using the definition of $\{\!|\Delta/\Gamma|\!\}_E$.

CASE: ($\Gamma\equiv X{<:}A,\Gamma'$)

1. ASSUME: $E,X{<:}A,\Gamma';K\cup\lceil X{<:}A,\Gamma'\rceil\vdash b:B$ and $E,K'\vdash(\Theta)(\Delta)\Rightarrow C$, where $\Theta\equiv\lfloor X{<:}A,\Gamma'\rfloor_B$.
2. By the definition of $\lfloor\ \rfloor$, $\Theta\equiv\forall(X{<:}A)\Theta'$, where $\Theta'\equiv\lfloor\Gamma'\rfloor_B$.
3. By (Arg Type), $\Delta\equiv B',\Delta'$, $E;K'\vdash B'{<:}A$, and $E;K'\vdash(\Theta'\{\!|{}^{B'}/_{X}|\!\})(\Delta')\Rightarrow C$.
4. Note also that $\lceil X{<:}A,\Gamma'\rceil\equiv\lceil\Gamma'\rceil\equiv\lceil\Gamma'\{\!|{}^{B'}/_{X}|\!\}\rceil$, and similarly $\lceil B',\Delta'\rceil\equiv\lceil\Delta'\rceil$.
5. By Lemma A.9, $\Theta'\{\!|{}^{B'}/_{X}|\!\}\equiv\lfloor\Gamma'\{\!|{}^{B'}/_{X}|\!\}\rfloor_{B\{\!|{}^{B'}/_{X}|\!\}}$.
6. By 1, 3, and Lemma A.5, $E,\Gamma'\{\!|{}^{B'}/_{X}|\!\};K\cup\lceil\Gamma'\{\!|{}^{B'}/_{X}|\!\}\rceil\vdash b\{\!|{}^{B'}/_{X}|\!\}:B\{\!|{}^{B'}/_{X}|\!\}$.
7. By 3, 4, 5, 6, and the induction hypothesis, $C\equiv B\{\!|{}^{B'}/_{X}|\!\}\{\!|\Delta'/\Gamma'\{\!|{}^{B'}/_{X}|\!\}|\!\}$ and $E;K\cup\lceil\Delta'\rceil\vdash b\{\!|{}^{B'}/_{X}|\!\}\{\!|\Delta'/\Gamma'\{\!|{}^{B'}/_{X}|\!\}|\!\}:C$, from which we obtain the desired result.

CASE: ($\Gamma\equiv\alpha\prec:p,\Gamma'$)

1. ASSUME: $E,\alpha\prec:p,\Gamma';K\cup\lceil\alpha\prec:p,\Gamma'\rceil\vdash b:B$ and $E,K'\vdash(\Theta)(\Delta)\Rightarrow C$, where $\Theta\equiv\lfloor\alpha\prec:p,\Gamma'\rfloor_B$.
2. By the definition of $\lfloor\ \rfloor$, $\Theta\equiv\forall(\alpha\prec:p)\Theta'$, where $\Theta'\equiv\lfloor\Gamma'\rfloor_B$.
3. By (Arg Context $\prec:$), $\Delta\equiv q,\Delta'$, $E;K'\vdash q\prec:p$, and $E;K'\vdash(\Theta'\{\!|{}^{q}/_{\alpha}|\!\})(\Delta')\Rightarrow C$.
4. Note also that $\lceil\alpha\prec:p,\Gamma'\rceil\equiv\langle\alpha\rangle\cup\lceil\Gamma'\rceil\equiv\langle\alpha\rangle\cup\lceil\Gamma'\{\!|{}^{q}/_{\alpha}|\!\}\rceil$, and similarly $\lceil q,\Delta'\rceil\equiv\langle q\rangle\cup\lceil\Delta'\rceil$.
5. By Lemma A.9, $\Theta'\{\!|{}^{q}/_{\alpha}|\!\}\equiv\lfloor\Gamma'\{\!|{}^{q}/_{\alpha}|\!\}\rfloor_{B\{\!|{}^{q}/_{\alpha}|\!\}}$.
6. By 1, 3, and Lemma A.5, $E,\Gamma'\{\!|{}^{q}/_{\alpha}|\!\};K\cup\langle q\rangle\cup\lceil\Gamma'\{\!|{}^{q}/_{\alpha}|\!\}\rceil\vdash b\{\!|{}^{q}/_{\alpha}|\!\}:B\{\!|{}^{q}/_{\alpha}|\!\}$.

7. By 3, 4, 5, 6, and the induction hypothesis, $C \equiv B\{\!\{^q/_\alpha\}\!\}\{\!\{\Delta'/\Gamma'\{\!\{^q/_\alpha\}\!\}\}\!\}$ and $E;K \cup \langle q \rangle \cup \lceil \Delta' \rceil \vdash b\{\!\{^q/_\alpha\}\!\}\{\!\{\Delta'/\Gamma'\{\!\{^q/_\alpha\}\!\}\}\!\} : C$, from which we obtain the desired result.

CASE: $(\Gamma \equiv \alpha :\succ p, \Gamma')$ Similar to previous case.

□

The following simple lemma lifts (Val Subsumption) to configurations:

Lemma A.12 *If* $E;K \vdash (\Pi, \sigma, a) : A$ *and* $E;K' \vdash A<:A'$*, where* $E \vdash K \subseteq K'$*, then* $E;K' \vdash (\Pi, \sigma, a) : A'$

PROOF: Straightforward. (Val Config) followed by (Val Subsumption) followed by (Val Config) again. □

We can now prove type preservation, which is restated here:

Lemma A.13 (Type Preservation) *If* $E;K \vdash (\Pi, \sigma, a) : A$ *and* $(\Pi, \sigma, a) \Downarrow (\Pi', \sigma', v)$*, then there exists an environment* E' *such that* $E' \gg E$ *and* $E';K \vdash (\Pi', \sigma', v) : A$.

PROOF:

CASE: (Subst Value) Immediate.

CASE: (Subst Object) Let $o \equiv [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}]^p_q$.

1. ASSUME: $E;K \vdash (\Pi, \sigma_0, o) : A$.
2. By 1 and (Val Config),
   1. $E;K \vdash o : A$,
   2. $E \vdash \sigma_0$, and
   3. $\mathsf{dom}(E') = \mathsf{dom}(\sigma_0)$, where $E \equiv \Pi, E'$.
3. By 2:1 and (Val Subsumption),
   1. $E;\langle p \rangle \vdash o : [l_i : \Theta_i{}^{i \in 1..n}]^p_q$,
   2. $E;K \vdash [l_i : \Theta_i{}^{i \in 1..n}]^p_q <: A$, and $E \vdash \langle p \rangle \subseteq K$.
4. By 2:2 and (Val Store), $E;\langle p \rangle \vdash o : [l_i : \Theta_i{}^{i \in 1..n}]^p_q$, where $\iota : [l_i : \Theta_i{}^{i \in 1..n}]^p_q \in E$, for all $\iota \mapsto o \in \sigma_0$.
5. LET: 1. $E'_1 = E', \iota : [l_i : \Theta_i{}^{i \in 1..n}]^p_q$,
   2. $E_1 = \Pi, E'_1$, and
   3. $\sigma_1 = (\iota \mapsto o) :: \sigma_0$.
6. Clearly from 5 we have, $E_1 \gg E$ and $\mathsf{dom}(E'_1) = \mathsf{dom}(\sigma_1)$.
7. By 2:2, 4, 5, and (Val Store), liberally applying Lemma A.3, $E_1 \vdash \sigma_1$.
8. By 5:1, 3:2, (Val Location) and (Val Subsumption), $E_1;K \vdash \iota : A$.

9. Therefore, by 6, 7, 8, and (Val Config), $E_1; K \vdash (\Pi, \sigma_1, \iota) : A$.

CASE: (Subst Select)

1. ASSUME: $E; K^\dagger \vdash (\Pi_0, \sigma_0, \iota.l_j\langle\Delta\rangle) : A^\dagger$.
2. By (Val Config),
   1. $E; K^\dagger \vdash a.l_j\langle\Delta\rangle : A^\dagger$,
   2. $E \vdash \sigma_0$, and
   3. $\mathsf{dom}(E') = \mathsf{dom}(\sigma_0)$, where $E \equiv \Pi_0, E'$.
3. By 2:1 and (Val Subsumption),
   1. $E; K \vdash \iota.l_j\langle\Delta\rangle : C_j$, where
   2. $E; K^\dagger \vdash C_j <: A^\dagger$, and $E \vdash K \subseteq K^\dagger$.
4. By 3:1 and (Val Select),
   1. $E; K \vdash \iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q$ and
   2. $E; K \vdash (\Theta_j)(\Delta) \Rightarrow C_j$.
5. By 4:1 and (Val Subsumption),
   1. $E; \langle p\rangle \vdash \iota : [l_i : \Theta_i{}^{i\in 1..n+m}]^p_q$, where
   2. $E; K \vdash \iota : [l_i : \Theta_i{}^{i\in 1..n+m}]^p_q <: [l_i : \Theta_i{}^{i\in 1..n}]^p_q$, and
   3. $E \vdash \langle p\rangle \subseteq K$.
6. 2:2, 5:1, and (Val Store), $E; \langle p\rangle \vdash o : [l_i : \Theta_i{}^{i\in 1..n+m}]^p_q$, where $\iota \mapsto o \in \sigma_0$.
7. By 6 and (Val Object), $E, s_j : [l_i : \Theta_i{}^{i\in 1..n+m}]^p_q, \Gamma_j; \langle q\uparrow\rangle \cup \lceil\Gamma_j\rceil \vdash b_j : B_j$, where $\lfloor\Gamma_j\rfloor_{B_j} \equiv \Theta_i$.
8. By 5:1, 7, and Lemma A.5, $E, \Gamma_j; \langle q\uparrow\rangle \cup \lceil\Gamma_j\rceil \vdash b_j\{\!\{\iota/s_j\}\!\} : B_j$.
9. By 4:2, 8, Lemma A.11 Lemma A.2:2,
   1. $E; \langle q\uparrow\rangle \cup \lceil\Delta\rceil \vdash b_j\{\!\{\iota/s_j\}\!\}\{\!\{\Delta/\Gamma_j\}\!\} : C_j$ and
   2. $E; K \vdash C_j$.
10. By 2:2, 2:3, 9:1, and (Val Config), $E; \langle q\uparrow\rangle \cup \lceil\Delta\rceil \vdash (\Pi_0, \sigma, b_j\{\!\{\iota/s_j\}\!\}\{\!\{\Delta/\Gamma_j\}\!\}) : C_j$.
11. By 10 and the induction hypothesis, if $(\Pi_0, \sigma, b_j\{\!\{\iota/s_j\}\!\}\{\!\{\Delta/\Gamma_j\}\!\}) \Downarrow (\Pi_1, \sigma_0, v)$, then there exists an E_1 such that $E_1 \gg E$, and $E_1; \langle q\uparrow\rangle \cup \lceil\Delta\rceil \vdash (\Pi_1, \sigma_1, v) : C_j$.
12. By 11, and (Val Config),
   1. $E_1; \langle q\uparrow\rangle \cup \lceil\Delta\rceil \vdash v : C_j$,
   2. $E_1 \vdash \sigma_1$, and
   3. $\mathsf{dom}(E_1') = \mathsf{dom}(\sigma_1)$, where $E_1 \equiv \Pi_1, E_1'$.
13. By 9:2, 12:1, and the Permissibility Lemma (A.8) (and Lemma A.3), $E_1; K \vdash v : C_j$.
14. By 3:2, 13, and (Val Subsumption), $E_1; K^\dagger \vdash v : A^\dagger$.
15. By 12, 14, and (Val Config), $E_1; K^\dagger \vdash (\Pi_1, \sigma_1, v) : A^\dagger$.

CASE: (Subst Update)

1. ASSUME: $E;K^\dagger \vdash (\Pi,\sigma_0,\iota.l_j \Leftarrow \varsigma(s:A,\Gamma)b) : A^\dagger$.
2. By 1 and (Val Config),
   1. $E;K^\dagger \vdash \iota.l_j \Leftarrow \varsigma(s:A,\Gamma)b : A^\dagger$,
   2. $E \vdash \sigma_0$, and
   3. $\mathsf{dom}(E') = \mathsf{dom}(\sigma_0)$, where $E \equiv \Pi_0, E'$.
3. By 2:1 and (Val Subsumption),
   1. $E;K \vdash \iota.l_j \Leftarrow \varsigma(s:A,\Gamma)b : A$, where $A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_q$,
   2. $E;K^\dagger \vdash A <: A^\dagger$, and $E \vdash K \subseteq K^\dagger$.
4. By 3:1 and (Val Update),
   1. $E;K \vdash \iota : A$,
   2. $E, s:A, \Gamma; K' \vdash b : C_j$, where $\lfloor \Gamma \rfloor_{C_j} \equiv \Theta_j$,
   3. $E, \Gamma \vdash K' \subseteq \langle q\uparrow\rangle \cup \lceil \Gamma \rceil$, and
   4. $E, \Gamma \vdash K' \subseteq K \cup \lceil \Gamma \rceil$.
5. By 4:2, 4:3, and (Val Subsumption), $E, s:A, \Gamma; \langle q\uparrow\rangle \cup \lceil \Gamma \rceil \vdash b : C_j$.
6. By 4:1, (Val Location) and (Val Subsumption),
   1. $E;\langle p\rangle \vdash \iota : [l_i : \Theta_i{}^{i\in 1..n+m}]^p_q$,
   2. $E;K \vdash [l_i : \Theta_i{}^{i\in 1..n+m}]^p_q <: A$, and
   3. $E \vdash \langle p\rangle \subseteq K$.
7. By 5, 6:2, and Lemma A.4, $E, s : [l_i : \Theta_i{}^{i\in 1..n+m}]^p_q, \Gamma; \langle q\uparrow\rangle \cup \lceil \Gamma \rceil \vdash b : C_j$.
8. By 2:2 and (Val Store), $E;\langle p\rangle \vdash o : [l_i : \Theta_i{}^{i\in 1..n+m}]^p_q$, where $\iota \mapsto o \in \sigma_0$ and $o \equiv [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i\in 1..n+m}]^p_q$, (and similarly for the remainder of the store).
9. By 8 and (Val Object), $E, s_i : A, \Gamma_i; \langle q\uparrow\rangle \cup \lceil \Gamma_i \rceil \vdash b_i : C_i$, for all $i \in 1..n+m$.
10. LET: $o' = [l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i\in 1..j-1, j+1..n}, l_j = \varsigma(s:A,\Gamma)b]^p_q)$.
11. By 7, 9, 10, and (Val Object), $E;\langle p\rangle \vdash o' : [l_i : \Theta_i{}^{i\in 1..n+m}]^p_q$.
12. LET: $\sigma_1 = \sigma_0 + (\iota \mapsto o')$.
13. By 8, 11, 12, and (Val Store), $E \vdash \sigma_1$.
14. By 4:1, 3:2, and (Val Subsumption), $E;K^\dagger \vdash \iota : A^\dagger$.
15. By 2:3, 13, 14, and (Val Config), $E;K^\dagger \vdash (\Pi, \sigma_1, \iota) : A^\dagger$.

CASE: (Subst Let)

1. ASSUME: $E;K^\dagger \vdash (\Pi_0, \sigma_0, \mathbf{let}\ x : A = a\ \mathbf{in}\ b) : A^\dagger$.
2. By (Val Config),
   1. $E;K^\dagger \vdash \mathbf{let}\ x : A = a\ \mathbf{in}\ b : A^\dagger$,
   2. $E \vdash \sigma_0$, and
   3. $\mathsf{dom}(E') = \mathsf{dom}(\sigma_0)$, where $E \equiv \Pi_0, E'$.

3. By 2:1 and (Val Subsumption),
   1. $E;K \vdash \textbf{let}\ x : A = a\ \textbf{in}\ b : B$ where
   2. $E;K^\dagger \vdash B <: A^\dagger$ and $E \vdash K \subseteq K^\dagger$.
4. By 3:1 and (Val Let),
   1. $E;K \vdash a : A$, and
   2. $E, x : A; K \vdash b : B$.
5. By 2:2, 2:3, 4:1, and (Val Config), $E;K \vdash (\Pi_0, \sigma_0, a) : A$.
6. By 5 and the induction hypothesis, if $(\Pi_0, \sigma_0, a) \Downarrow (\Pi_1, \sigma_1, v)$, then there exists an E_1 such that $E_1 \gg E$ and $E_1; K \vdash (\Pi_1, \sigma_1, v) : A$.
7. By 6 and (Val Config),
   1. $E_1; K \vdash v : A$,
   2. $E_1 \vdash \sigma_1$, and
   3. $\mathsf{dom}(E_1') = \mathsf{dom}(\sigma_1)$, where $E_1 \equiv \Pi_1, E_1'$.
8. By 4:2 and Lemma A.3, $E_1, x : A; K \vdash b : B$.
9. By 8, 7:1, and Lemma A.5, $E_1; K \vdash b\{^v/_x\} : B$.
10. By 7:2, 7:3, 9, and (Val Config), $E_1; K \vdash (\Pi_1, \sigma_1, b\{^v/_x\}) : B$.
11. By 10 and the induction hypothesis, if $(\Pi_1, \sigma_1, b\{^v/_x\}) \Downarrow (\Pi_2, \sigma_2, u)$, then there exists an E_2 such that $E_2 \gg E_1$ and $E_2; K \vdash (\Pi_2, \sigma_2, u) : B$.
12. By 3, 11, Lemma A.3, and Lemma A.12, $E_2; K^\dagger \vdash (\Pi_2, \sigma_2, u) : A^\dagger$.

CASE: (Subst Unfold)

1. ASSUME: $E;K^\dagger \vdash (\Pi, \sigma, \textbf{unfold}(\textbf{fold}(A, v))) : B^\dagger$.
2. By 1 and (Val Config),
   1. $E;K^\dagger \vdash \textbf{unfold}(\textbf{fold}(A, v)) : B^\dagger$,
   2. $E \vdash \sigma$, and
   3. $\mathsf{dom}(E') = \mathsf{dom}(\sigma)$, where $E \equiv \Pi, E'$.
3. By 2:1 and (Val Subsumption),
   1. $E;K' \vdash \textbf{unfold}(\textbf{fold}(A, v)) : B'\{^{A'}/_Y\}$, where $A' \equiv \mu(Y)B'$,
   2. $E;K^\dagger \vdash B'\{^{A'}/_Y\} <: B^\dagger$, and $E \vdash K' \subseteq K^\dagger$.
4. By 3:1 and (Val Unfold), $E;K' \vdash \textbf{fold}(A, v) : A'$.
5. By 4:1 and (Val Subsumption),
   1. $E;K \vdash \textbf{fold}(A, v) : A$, where $A \equiv \mu(X)B$,
   2. $E;K' \vdash A <: A'$, and
   3. $E \vdash K \subseteq K'$.
6. By 5:1 and (Val Fold), $E;K \vdash v : B\{^A/_X\}$.
7. Step 5:2 is equivalent to stating $E;K' \vdash \mu(X)B <: \mu(Y)B'$.

8. By 7 and (Sub Rec),
   1. $E; K' \vdash \mu(X)B$,
   2. $E; K' \vdash \mu(Y)B'$, and
   3. $E, Y{<:}\mathsf{Top}^{K'}, X{<:}Y; K' \vdash B{<:}B'$,
   4. where $X \notin FV(B')$ and $Y \notin FV(B)$.
9. By 8:2 and (Sub Top), $E; K' \vdash \mu(Y)B' {<:} \mathsf{Top}^{K'}$.
10. By 8:3, 9, and Lemma A.5, $E, X{<:}\mu(Y)B'; K' \vdash B{<:}B'\{^{A'}/_{Y}\}$, recalling that $A' \equiv \mu(Y)B'$.
11. By 7, 10, and Lemma A.5, $E; K' \vdash B\{^{A}/_{X}\}{<:}B'\{^{A'}/_{Y}\}$, recalling that $A \equiv \mu(X)B$.
12. By 5:3, 6, 11, and (Val Subsumption), $E; K' \vdash v : B'\{^{A'}/_{Y}\}$.
13. By 12, 3:2, and (Val Subsumption), $E; K^\dagger \vdash v : A^\dagger$.
14. By 2:2, 2:3, 13 and (Val Config), $E; K^\dagger \vdash (\Pi, \sigma, v) : A^\dagger$.

CASE: (Subst Expose)

1. ASSUME: $E; K^\dagger \vdash (\Pi_0, \sigma_0, \textbf{expose}\ (\textbf{hide}\ p\ \textbf{as}\ \alpha \prec: p'\ \textbf{in}\ v{:}A)\ \textbf{as}\ \alpha \prec: p'', x{:}A'\ \textbf{in}\ b{:}B) : A^\dagger$.
2. By 1 and (Val Config),
   1. $E; K^\dagger \vdash \textbf{expose}\ (\textbf{hide}\ p\ \textbf{as}\ \alpha \prec: p'\ \textbf{in}\ v{:}A)\ \textbf{as}\ \alpha \prec: p'', x{:}A'\ \textbf{in}\ b{:}B : A^\dagger$,
   2. $E \vdash \sigma_0$, and
   3. $\mathsf{dom}(E') = \mathsf{dom}(\sigma_0)$, where $E \equiv \Pi_0, E'$.
3. By 2:1 and (Val Subsumption),
   1. $E; K \vdash \textbf{expose}\ (\textbf{hide}\ p\ \textbf{as}\ \alpha \prec: p'\ \textbf{in}\ v{:}A)\ \textbf{as}\ \alpha \prec: p'', x{:}A'\ \textbf{in}\ b{:}B : B$, where
   2. $E; K^\dagger \vdash B{<:}A^\dagger$, and $E \vdash K{<:}K^\dagger$.
4. By 3:1 and (Val Expose),
   1. $E; K \vdash \textbf{hide}\ p\ \textbf{as}\ \alpha \prec: p'\ \textbf{in}\ v{:}A : \exists(\alpha \prec: p'')A'$,
   2. $E; K \vdash B$, and
   3. $E, \alpha \prec: p'', x : A'; K \cup \langle\alpha\rangle \vdash b : B$.
5. By 4:1 and (Val Subsumption),
   1. $E; K' \vdash \textbf{hide}\ p\ \textbf{as}\ \alpha \prec: p'\ \textbf{in}\ v{:}A : \exists(\alpha \prec: p')A$,
   2. $E; K \vdash \exists(\alpha \prec: p')A {<:} \exists(\alpha \prec: p'')A'$, and
   3. $E \vdash K' \subseteq K$.
6. By 5:1 and (Val Hide),
   1. $E \vdash p \prec: p'$,
   2. $E; K' \vdash v\{^{p}/_{\alpha}\} : A\{^{p}/_{\alpha}\}$, and
   3. $E; K' \vdash \exists(\alpha \prec: p')A$.
7. By 5:2, (Sub Exists),

1. $E \vdash p' \prec: p''$ and
2. $E, \alpha \prec: p'; K \vdash A <: A'$.

8. By 4:3, 6:1, 7:1, (In Trans) and Lemma A.5, $E, x : A'\{^{p}/_{\alpha}\}; K \cup \langle p \rangle \vdash b\{^{p}/_{\alpha}\} : B$.
9. By 6:1, 7:2, and Lemma A.5, $E; K \vdash A\{^{p}/_{\alpha}\} <: A'\{^{p}/_{\alpha}\}$.
10. By 6:2, 5:3, (Val Subsumption), and 9 and (Sub Allow),
    1. $E; K \cup \langle p \rangle \vdash v\{^{p}/_{\alpha}\} : A\{^{p}/_{\alpha}\}$, and
    2. $E; K \cup \langle p \rangle \vdash A\{^{p}/_{\alpha}\} <: A'\{^{p}/_{\alpha}\}$.
11. By 6:2, 8, 10, Lemma A.5, $E; K \cup \langle p \rangle \vdash b\{^{v\{^{p}/_{\alpha}\}}/_{x}\}\{^{p}/_{\alpha}\} : B$.
12. By 2:2, 2:3, 11, and (Val Config), $E; K \cup \langle p \rangle \vdash (\Pi_0, \sigma_0, b\{^{v}/_{x}\}\{^{p}/_{\alpha}\}) : B$.
13. By 12 and the induction hypothesis, if $(\Pi_0, \sigma_0, b\{^{v}/_{x}\}\{^{p}/_{\alpha}\}) \Downarrow (\Pi_1, \sigma_1, u)$, then there exists an E_1 such that $E_1 \gg E$ and $E_1; K \cup \langle p \rangle \vdash (\Pi_1, \sigma_1, u) : B$.
14. By 13 and (Val Config),
    1. $E_1; K \cup \langle p \rangle \vdash u : B$,
    2. $E_1 \vdash \sigma_1$, and
    3. $\mathsf{dom}(E_1') = \mathsf{dom}(\sigma_1)$, where $E_1 \equiv \Pi_1, E_1'$.
15. By 4:2, 14:1, and the Permissibility Lemma (A.8) (and Lemma A.3), $E_1; K \vdash u : B$.
16. By 3:2, 15, and (Val Substitution), $E_1; K^\dagger \vdash u : A^\dagger$.
17. By 14:2, 14:3, 16, and (Val Config), $E_1; K^\dagger \vdash (\Pi_1, \sigma_1, u) : A^\dagger$.

CASE: (Subst New)

1. ASSUME: $E; K^\dagger \vdash (\Pi_0, \sigma_0, \mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ a) : A^\dagger$.
2. By 1 and (Val Config),
    1. $E; K^\dagger \vdash \mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ a : A^\dagger$,
    2. $E \vdash \sigma$, and
    3. $\mathsf{dom}(E') = \mathsf{dom}(\sigma_0)$, where $E \equiv \Pi_0, E'$.
3. By 2:1 and (Val Subsumption),
    1. $E; K \vdash \mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ a : A$,
    2. $E; K^\dagger \vdash A <: A^\dagger$, and $E \vdash K \subseteq K^\dagger$.
4. By 3:1 and (Val New),
    1. $E, \alpha \prec: p; K \cup \langle \alpha \rangle \vdash a : A$, and
    2. $E; K \vdash A$.
5. LET: 1. $\alpha' \notin \mathsf{dom}(\Pi_0)$, *i.e.*, α' is fresh,
    2. $\Pi_1 = \Pi_0, \alpha' \prec: p$, and
    3. $E_1 = \Pi_1, E'$. Clearly $E_1 \gg E$.
6. By 5, and (In $\prec:$) $E_1 \vdash \alpha' \prec: p$.
7. By 4:1, 6, and Lemma A.5 (and Lemma A.3), $E_1; K \cup \langle \alpha' \rangle \vdash a\{^{\alpha'}/_{\alpha}\} : A$, noting

from 4:2 that neither $\alpha \notin FV(K)$ nor $\alpha' \notin FV(A)$.

8. By 2:2, 5:3, 7, (Val Config), and Lemma A.3, $E_1; K \cup \langle \alpha' \rangle \vdash (\Pi_1, \sigma_0, a\{^{\alpha'}/_{\alpha}\}) : A$.
9. By 8 and the induction hypothesis, if $(\Pi_1, \sigma_0, a\{^{\alpha'}/_{\alpha}\}) \Downarrow (\Pi_2, \sigma_1, v)$, then there exists an E_2 such that $E_2 \gg E_1$, and $E_2; K \cup \langle \alpha' \rangle \vdash (\Pi_2, \sigma_1, v) : A$
10. By 8 and (Val Config),
    1. $E_2; K \cup \langle \alpha' \rangle \vdash v : A$,
    2. $E_2 \vdash \sigma_1$, and
    3. $\mathsf{dom}(E_2') = \mathsf{dom}(\sigma_1)$, where $E_2 \equiv \Pi_2, E_2'$.
11. By 10:1, 4:2, Lemma A.3, and the Permissibility Lemma (A.8), $E_2; K \vdash v : A$.
12. By 11, 3:2, and (Val Subsumption), $E_2; K^\dagger \vdash v : B^\dagger$.
13. Thus by 10:2, 10:3, 12, and (Val Config), $E_2; K^\dagger \vdash (\Pi_2, \sigma_1, v) : B^\dagger$. □

A.4 Proof of Lemma 5.11

The proof that well-typed expressions do not become stuck relies on the following Canonical Forms Lemma, which states that values of a given type have the expected form.

Lemma A.14 (Canonical Forms) *Let v be a closed value, that is, a value without free term or type variables, and assume that $E; K \vdash v : A$. Then:*

1. *If $A \equiv [l_i : \Theta_i{}^{i \in 1..n}]_q^p$, then $v \equiv \iota$. Moreover, if $E \vdash \sigma$, then $\sigma(\iota)$ has the form $[l_i = \varsigma(s_i : A_i, \Gamma_i) b_i{}^{i \in 1..n}, \ldots]_q^p$.*
2. *If $A \equiv \mu(X)B$, then $v \equiv \textbf{\textit{fold}}(A', v')$ for some A' and v'.*
3. *If $A \equiv \exists(\alpha \prec: p)B$, then $v \equiv \textbf{\textit{hide}}\ p'\ \textbf{\textit{as}}\ \alpha \prec: p''\ \textbf{\textit{in}}\ v' : A'$, for some p', p'', v' and A'. Moreover, $E \vdash p' \prec: p$.*

PROOF: By inspection of the typing rules for values. □

Lemma A.15 (Faulty Programs are Untypable (Restated)) *If $(\Pi, \sigma, a) \Downarrow$ WRONG, then there are no E, K, A such that $E; K \vdash (\Pi, \sigma, a) : A$.*

PROOF: By induction over $(\Pi, \sigma, a) \Downarrow$ WRONG. Proving the contrapositive:
Error Introduction Rules

CASE: (Error Select1)

1. ASSUME: $E;K \vdash (\Pi_0, \sigma_0, v.l_j\langle\Delta\rangle) : A$.
2. Following case (Subst Select) of the proof of Lemma A.13 until step 4, we obtain that $E;K \vdash v : [l_i : \Theta_i{}^{i\in 1..n}]^p_q$.
3. By Lemma A.14 $v \equiv \iota$, and that the $\sigma(\iota)$ has a method named l. Hence this rule cannot apply.

CASE: (Error Select2), (Error Update1), (Error Update2), (Error Unfold), (Error Unfold), and (Error Expose1) follow the pattern of (Error Select1), with straightforward application of Lemma A.14.

CASE: (Error Select3)

1. ASSUME: $E;K \vdash (\Pi_0, \sigma_0, \iota.l_j\langle\Delta\rangle) : A$.
2. Following case (Subst Select) of the proof of Lemma A.13 until step 9, where Lemma A.11 asserts the existence of the appropriate substitution, $\{\!\{^{\Delta}/_{\Gamma_j}\}\!\}_{\Pi_0}$. Therefore this error cannot occur.

CASE: (Error Update3)

1. ASSUME: $E;K \vdash (\Pi_0, \sigma_0, \iota.l_j \Leftarrow \varsigma(s : A, \Gamma)b) : A$
2. Following case (Subst Update) of the proof of Lemma A.13 until step 9. At this point we deduce that $\lfloor\Gamma\rfloor_{C_j} \equiv \Theta_j \equiv \lfloor\Gamma_j\rfloor_{C_j}$, because of α-renaming of bound variables. It is easy to see that $match(\Gamma_j; \Gamma)$ is true. Therefore this error cannot occur.

Error Propagation Rules

CASE: (Error Select4)

1. ASSUME: $E;K \vdash (\Pi_0, \sigma_0, \iota.l_j\langle\Delta\rangle) : A$
2. Following case (Susbt Select) of the proof of Lemma A.13 until step 10, we obtain $E;\langle q\uparrow\rangle \cup \lceil\Delta\rceil \vdash (\Pi_0, \sigma, b_j\{\!\{^{\iota}/_{s_j}\}\!\}\{\!\{\Delta/\Gamma_j\}\!\}) : C_j$.
3. By the induction hypothesis, $(\Pi_0, \sigma, b_j\{\!\{^{\iota}/_{s_j}\}\!\}\{\!\{\Delta/\Gamma_j\}\!\}) \not\Downarrow$ WRONG, that is it evaluates to some final configuration t or diverges.
4. Hence, $(\Pi_0, \sigma_0, \iota.l_j\langle\Delta\rangle) \not\Downarrow$ WRONG, by evaluating to the same configuration or diverging.

CASE: (Error Let1), (Error Let2), (Error New), and (Error Expose2) follow the same pattern as (Error Select4). □

A.5 Proof of Theorem 5.19

Again we state additional properties before we prove the main result.

Lemma A.16 *Assume $E \vdash p \prec: q$ and $\eta \models E$. Then:*

1. $[\![p]\!]_\eta \in [\![p]\!]_\eta{\downarrow} \subseteq [\![q]\!]_\eta{\downarrow}$, *and*
2. $[\![q]\!]_\eta \in [\![q]\!]_\eta{\uparrow} \subseteq [\![p]\!]_\eta{\uparrow}$.

PROOF: Clear from the definitions. □

The following simple equivalences save unfolding the definition of $[\![a]\!]_\eta$ in the proofs which follow.

Lemma A.17 (Simple Equivalences)

$$\begin{aligned}
[\![\textbf{\textit{fold}}(A,v)]\!]_\eta &= [\![v]\!]_\eta \\
[\![\textbf{\textit{hide}}\ p\ \textbf{\textit{as}}\ \alpha \prec: q\ \textbf{\textit{in}}\ v{:}A]\!]_\eta &= [\![v]\!]_\eta \\
[\![v.l\langle\Delta\rangle]\!]_\eta &= [\![v]\!]_\eta \cup [\![\Delta]\!]_\eta \\
[\![v,\Delta]\!]_\eta &= [\![v]\!]_\eta \cup [\![\Delta]\!]_\eta \\
[\![A,\Delta]\!]_\eta &= [\![\Delta]\!]_\eta \\
[\![p,\Delta]\!]_\eta &= [\![\Delta]\!]_\eta \\
[\![v.l \Leftarrow \varsigma(s:A,\Gamma)b]\!]_\eta &= [\![v]\!]_\eta \cup [\![b]\!]_\eta \\
[\![\textbf{\textit{let}}\ x:A = a\ \textbf{\textit{in}}\ b]\!]_\eta &= [\![a]\!]_\eta \cup [\![b]\!]_\eta \\
[\![\textbf{\textit{unfold}}(v)]\!]_\eta &= [\![v]\!]_\eta \\
[\![\textbf{\textit{new}}\ \alpha \lhd p\ \textbf{\textit{in}}\ a]\!]_\eta &= [\![a]\!]_\eta \\
[\![\textbf{\textit{expose}}\ v\ \textbf{\textit{as}}\ \alpha \prec: p,x{:}A\ \textbf{\textit{in}}\ b{:}B]\!]_\eta &= [\![v]\!]_\eta \cup [\![b]\!]_\eta
\end{aligned}$$

The next definition simplifies the statement of the lemma which follows.

Definition A.18 *If a has the form*

- $v.l_j \Leftarrow \varsigma(s:A,\Gamma)b$,
- ***new*** $\alpha \lhd p$ ***in*** b, *or*
- ***expose*** v ***as*** $\alpha \prec: p,x{:}A$ ***in*** $b{:}B$.

then b is called the immediate subterm *of a.*

Lemma A.19 *If $E;K \vdash a:A$, $\eta \models E$, and b is a immediate subterm of a, then $[\![b]\!]_\eta \subseteq [\![a]\!]_\eta$. Consequently, $[\![b]\!]_{\eta^+} = [\![b]\!]_\eta$, for any η^+ such that $\eta \subseteq \eta^+$.*

Recall that η is a map, hence a set of ordered pairs, so $\eta \subseteq \eta^+$ above simply states that η^+ is a proper extension of the map η.

The notation $\eta[\alpha \mapsto \kappa]$ represents the extension of the map η with the binding $\alpha \mapsto \kappa$. Now we can state the key lemma which allows context variables which are declared in method parameters, in **new** and **expose** to be ignored when interpreting those terms, since they will never be the owner of a location.

Lemma A.20 *If* $[\![b]\!]_{\eta[\alpha\mapsto\kappa]} \subseteq [\![K \cup \langle\alpha\rangle]\!]_{\eta[\alpha\mapsto\kappa]}$, $\alpha \notin \mathrm{FV}(K)$, *and* $[\![b]\!]_{\eta[\alpha\mapsto\kappa]} = [\![b]\!]_\eta$, *then* $[\![b]\!]_\eta \subseteq [\![K]\!]_\eta$.

PROOF SKETCH: Note that this says nothing about the well-formedness of $\eta[\alpha \mapsto \kappa]$. We can simply set κ to be some value, say $\bullet$, not in the range of η, and observe that $\bullet$ occurs neither in $[\![b]\!]_{\eta[\alpha\mapsto\bullet]}$ nor $[\![K]\!]_{\eta[\alpha\mapsto\bullet]}$, so removing $[\alpha \mapsto \bullet]$ affects neither definition. □

Lemma A.21 (Restated) *We have the following, where in each case* $\eta \models E$,

1. *If* $E \vdash K' \subseteq K$, *then* $[\![K']\!]_\eta \subseteq [\![K]\!]_\eta$;
2. *If* $E; K \vdash a : A$, *then* $[\![a]\!]_\eta \subseteq [\![K]\!]_\eta$;
3. *If* $E; K \vdash (\Theta)(\Delta) \Rightarrow C$, *then* $[\![\Delta]\!]_\eta \subseteq [\![K]\!]_\eta$.

PROOF SKETCH: By induction over judgments. Unless otherwise stated, assume that $\eta \models E$, where E is understood from context.

PROOF: **Part 1.**

CASE: (SubPerm p)

ASSUME: $E \vdash \langle p\rangle \subseteq \langle p\uparrow\rangle$. By definition, $[\![\langle p\rangle]\!]_\eta \subseteq [\![\langle p\uparrow\rangle]\!]_\eta$.

CASE: (SubPerm $\prec$:)

ASSUME: $E \vdash \langle p\uparrow\rangle \subseteq \langle q\uparrow\rangle$. By (SubPerm $\prec$:), $E \vdash q \prec: p$. By Lemma A.16, $[\![\langle p\uparrow\rangle]\!]_\eta \subseteq [\![\langle q\uparrow\rangle]\!]_\eta$.

CASE: (SubPerm Refl)

ASSUME: $E \vdash K \subseteq K$. By definition, $[\![K]\!]_\eta \subseteq [\![K]\!]_\eta$.

CASE: (SubPerm Trans)

ASSUME: $E \vdash K \subseteq K''$. By (SubPerm Trans), $E \vdash K \subseteq K'$ and $E \vdash K' \subseteq K''$. By the induction hypothesis, $[\![K]\!]_\eta \subseteq [\![K']\!]_\eta$ and $[\![K']\!]_\eta \subseteq [\![K'']\!]_\eta$.

CASE: (SubPerm Union-LB)

ASSUME: $E \vdash \bigcup[K_1..K_n] \subseteq K$. By (SubPerm Union-LB), $E \vdash K_i \subseteq K$ for all $i \in 1..n$. By the induction hypothesis, $[\![K_i]\!]_\eta \subseteq [\![K]\!]_\eta$ for all $i \in 1..n$. Therefore, $[\![\bigcup[K_1..K_n]]\!]_\eta \subseteq [\![K]\!]_\eta$.

CASE: (SubPerm Union-UB)

ASSUME: $E \vdash K_i \subseteq \bigcup[K_1..K_n]$, where $i \in 1..n$. By definition, $[\![\bigcup[K_1..K_n]]\!]_\eta = \bigcup_{i\in 1..n}[\![K_i]\!]_\eta$, therefore $[\![K_i]\!]_\eta \subseteq [\![\bigcup[K_1..K_n]]\!]_\eta$.

PROOF: **Part 2.**

We use Lemma A.17 without attributing it.

CASE: (Val x)

ASSUME: $E;K \vdash x : A$. By (Val x), $E;K \vdash A$. By definition, $[\![x]\!]_\eta = \emptyset \subseteq [\![K]\!]_\eta$.

CASE: (Val Location)

ASSUME: $E;\langle p\rangle \vdash \iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q$. By (Val Location), $\iota : [l_i : \Theta_i{}^{i\in 1..n}]^p_q \in E$. Thus $\eta(\iota) = [\![p]\!]_\eta \in [\![\langle p\rangle]\!]_\eta$.

CASE: (Val Object)

ASSUME: $E;\langle p\rangle \vdash [l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i\in 1..n}]^p_q : [l_i : \Theta_i{}^{i\in 1..n}]^p_q$.
By definition, $[\![[l_i = \varsigma(s_i : A_i, \Gamma_i)b_i{}^{i\in 1..n}]^p_q]\!]_\eta = \emptyset \subseteq [\![\langle p\rangle]\!]_\eta$.

CASE: (Val Select)

ASSUME: $E;K \vdash v.l_j\langle\Delta\rangle : C_j$. By (Val Select), $E;K \vdash v : [l_i : \Theta_i{}^{i\in 1..n}]^p_q$ and $E;K \vdash (\Theta_j)(\Delta) \Rightarrow C_j$. By the induction hypothesis, $[\![v]\!]_\eta \subseteq [\![K]\!]_\eta$ and $[\![\Delta]\!]_\eta \subseteq [\![K]\!]_\eta$. Thus $[\![v.l_j\langle\Delta\rangle]\!]_\eta = [\![v]\!]_\eta \cup [\![\Delta]\!]_\eta \subseteq [\![K]\!]_\eta$.

CASE: (Val Update)

ASSUME: $E;K \vdash v.l_j \Leftarrow \varsigma(s : A, \Gamma)b : A$, where $A \equiv [l_i : \Theta_i{}^{i\in 1..n}]^p_q$ and $\Theta_j \equiv \lfloor\Gamma\rfloor_{C_j}$. By (Val Update), $E;K \vdash v : A$, and $E, s : A, \Gamma; K' \vdash b : C_j$, and $E, \Gamma \vdash K' \subseteq \langle q\uparrow\rangle \cup \lceil\Gamma\rceil$, and $E, \Gamma \vdash K' \subseteq K \cup \lceil\Gamma\rceil$. By the induction hypothesis, $[\![v]\!]_\eta \subseteq [\![K]\!]_\eta$, $[\![b]\!]_{\eta^+} \subseteq [\![K']\!]_{\eta^+}$, $[\![K']\!]_{\eta^+} \subseteq [\![\langle q\uparrow\rangle \cup \lceil\Gamma\rceil]\!]_{\eta^+}$, and $[\![K']\!]_{\eta^+} \subseteq [\![K \cup \lceil\Gamma\rceil]\!]_{\eta^+}$, where $\eta^+ \models E, \Gamma$. Thus $[\![b]\!]_{\eta^+} \subseteq [\![K \cup \lceil\Gamma\rceil]\!]_{\eta^+}$. By Lemma A.19, $[\![b]\!]_\eta = [\![b]\!]_{\eta^+}$. From $E;K \vdash v : A$, we deduce from Lemma A.2 that $E \vdash K$, hence $\mathsf{dom}(\Gamma) \cap FV(K) = \emptyset$. By multiple applications of Lemma A.20, employing induction on the length of Γ, we obtain $[\![b]\!]_\eta \subseteq [\![K]\!]_\eta$. Therefore, $[\![v.l_j \Leftarrow \varsigma(s : A, \Gamma)b]\!]_\eta = [\![v]\!]_\eta \cup [\![b]\!]_\eta \subseteq [\![K]\!]_\eta$.

CASE: (Val Fold)

ASSUME: $E;K \vdash \mathbf{fold}(A, v) : A$. By (Val Fold), $E;K \vdash v : B\{{}^{A}/_{X}\}$. By the induction hypothesis, $[\![v]\!]_\eta \subseteq [\![K]\!]_\eta$. By definition, we have $[\![\mathbf{fold}(A, v)]\!]_\eta \subseteq [\![K]\!]_\eta$.

CASE: (Val Unfold)

ASSUME: $E;K \vdash \mathbf{unfold}(v) : B\{^{A}/_{X}\}$, where $A \equiv \mu(X)B$. By (Val Unfold), $E;K \vdash v : A$. By the induction hypothesis, $[\![v]\!]_\eta \subseteq [\![K]\!]_\eta$. By definition, $[\![\mathbf{unfold}(v)]\!]_\eta = [\![v]\!]_\eta$, from which the desired result follows.

CASE: (Val Let)

ASSUME: $E;K \vdash \mathbf{let}\ x : A = a\ \mathbf{in}\ b : B$. By (Val Let), $E;K \vdash a : A$ and $E,x : A;K \vdash b : B$. By induction hypothesis, $[\![a]\!]_\eta \subseteq [\![K]\!]_\eta$, and $[\![b]\!]_\eta \subseteq [\![K]\!]_\eta$. Thus by definition, $[\![\mathbf{let}\ x : A = a\ \mathbf{in}\ b]\!]_\eta = [\![a]\!]_\eta \cup [\![b]\!]_\eta \subseteq [\![K]\!]_\eta$.

CASE: (Val New)

ASSUME: $E;K \vdash \mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ a : A$. By (Val New), $E,\alpha \prec\!: p;K \cup \langle\alpha\rangle \vdash a : A$, and $E;K \vdash A$. By the induction hypothesis, $[\![a]\!]_{\eta[\alpha\mapsto\kappa]} \subseteq [\![K \cup \langle\alpha\rangle]\!]_{\eta[\alpha\mapsto\kappa]}$, where $\eta[\alpha \mapsto \kappa] \models E,\alpha \prec\!: p$. By Lemma A.19, $[\![a]\!]_\eta = [\![a]\!]_{\eta[\alpha\mapsto\kappa]}$. From $E;K \vdash A$, we deduce by Lemma A.2 that $E \vdash K$, and hence $\alpha \notin FV(K)$. Therefore by Lemma A.20, $[\![a]\!]_\eta = [\![\mathbf{new}\ \alpha \lhd p\ \mathbf{in}\ a]\!]_\eta \subseteq [\![K]\!]_\eta$, as required.

CASE: (Val Hide)

ASSUME: $E;K \vdash \mathbf{hide}\ p\ \mathbf{as}\ \alpha \prec\!: q\ \mathbf{in}\ v\!:\!A : \exists(\alpha \prec\!: q)A$. By (Val Hide), $E \vdash p \prec\!: q$, $E;K \vdash v\{^{p}/_{\alpha}\} : A\{^{p}/_{\alpha}\}$, and $E;K \vdash \exists(\alpha \prec\!: q)A$. By the induction hypothesis, $[\![v\{^{p}/_{\alpha}\}]\!]_\eta \subseteq [\![K]\!]_\eta$. By definition, $\mathsf{locs}(v\{^{p}/_{\alpha}\}) = \mathsf{locs}(v)$, thus $[\![v\{^{p}/_{\alpha}\}]\!]_\eta = [\![v]\!]_\eta$. By definition, $[\![\mathbf{hide}\ p\ \mathbf{as}\ \alpha \prec\!: q\ \mathbf{in}\ v\!:\!A]\!]_\eta = [\![v]\!]_\eta \subseteq [\![K]\!]_\eta$.

CASE: (Val Expose)

ASSUME: $E;K \vdash \mathbf{expose}\ v\ \mathbf{as}\ \alpha \prec\!: p,x\!:\!A\ \mathbf{in}\ b\!:\!B : B$. By (Val Expose), $E;K \vdash B$, $E;K \vdash v : \exists(\alpha \prec\!: p)A$, and $E,\alpha \prec\!: p,x : A;K \cup \langle\alpha\rangle \vdash b : B$. By the induction hypothesis, we have $[\![v]\!]_\eta \subseteq [\![K]\!]_\eta$, and $[\![b]\!]_{\eta[\alpha\mapsto\kappa]} \subseteq [\![K \cup \langle\alpha\rangle]\!]_{\eta[\alpha\mapsto\kappa]}$, where $\eta[\alpha \mapsto \kappa] \models E,\alpha \prec\!: p$. By Lemma A.19, $[\![b]\!]_\eta = [\![b]\!]_{\eta[\alpha\mapsto\kappa]}$. From $E;K \vdash B$, we deduce that $\alpha \notin FV(K)$. By Lemma A.20, $[\![b]\!]_\eta \subseteq [\![K]\!]_\eta$. Therefore, $[\![\mathbf{expose}\ v\ \mathbf{as}\ \alpha \prec\!: p,x\!:\!A\ \mathbf{in}\ b\!:\!B]\!]_\eta = [\![v]\!]_\eta \cup [\![b]\!]_\eta \subseteq [\![K]\!]_\eta$.

CASE: (Val Subsumption)

ASSUME: $E;K' \vdash a : B$. By (Val Subsumption), $E;K \vdash a : A$ and $E;K' \vdash A <\!: B$, and $E \vdash K \subseteq K'$. By the induction hypothesis, $[\![a]\!]_\eta \subseteq [\![K]\!]_\eta$, and $[\![K]\!]_\eta \subseteq [\![K']\!]_\eta$, from which the desired result follows.

PROOF: **Part 3.**

CASE: (Arg Empty)

ASSUME: $E;K \vdash (\emptyset)(\emptyset) \Rightarrow C$. By definition $[\![\emptyset]\!]_\eta = \emptyset \subseteq [\![K]\!]_\eta$.

CASE: (Arg Val)

ASSUME: $E;K \vdash (A \to \Theta)(v,\Delta) \Rightarrow C$. By (Arg Val), $E;K \vdash v : A$ and $E;K \vdash (\Theta)(\Delta) \Rightarrow C$. By the induction hypothesis, $[\![v]\!]_\eta \subseteq [\![K]\!]_\eta$ and $[\![\Delta]\!]_\eta \subseteq [\![K]\!]_\eta$. Therefore $[\![v,\Delta]\!]_\eta \subseteq [\![K]\!]_\eta$.

CASE: (Arg Type)

ASSUME: $E;K \vdash (\forall(X{<:}A)\Theta)(B,\Delta) \Rightarrow C$. By (Arg Type), $E;K \vdash B{<:}A$, and $E;K \vdash (\Theta\{^B/_X\})(\Delta) \Rightarrow C$. By the induction hypothesis, $[\![\Delta]\!]_\eta \subseteq [\![K]\!]_\eta$. By definition, $[\![B,\Delta]\!]_\eta = [\![\Delta]\!]_\eta$, from which the desired result follows.

CASE: (Arg Context $\prec$:)

ASSUME: $E;K \vdash (\forall(\alpha \prec: p)\Theta)(q,\Delta) \Rightarrow C$. By (Arg Context $\prec$:), $E;K \vdash (\Theta\{^q/_\alpha\})(\Delta) \Rightarrow C$. By the induction hypothesis, $[\![\Delta]\!]_\eta \subseteq [\![K]\!]_\eta$. By definition, $[\![p,\Delta]\!]_\eta = [\![\Delta]\!]_\eta$, from which the desired result follows.

CASE: (Arg Context $:\succ$) similar to (Arg Context $\prec$:).

□

Appendix B

When Owners are Dominators

The aim of this short appendix is to prove the claim that in the owners-as-dominators model, which underlies for example the **URDI**ς calculus, that owners are indeed dominators.

Firstly, in the owners-as-dominators model rep is 1:1 one, so we can write ι for $\mathsf{rep}(\iota)$ without ambiguity. Without loss of generality, we assume that there is a single constant context ε and that the object graph is given by the relation $\rightarrow \subseteq (O \cup \{\varepsilon\}) \times O$, where O is the collection of object ids. Edges $\varepsilon \rightarrow \iota$ are considered to be "from the root of the system." The partial order $\preceq \subseteq (O \cup \{\varepsilon\}) \times (O \cup \{\varepsilon\})$ is given by the reflexive transitive closure of the relation $\iota \lhd \mathsf{owner}(\iota)$, which corresponds to the property that the representation context for object ι is created directly inside ι's owner context. Since there is initially only a single context ε, we can deduce that $\iota \preceq \varepsilon$ for all ι. Additionally, we have that $\iota \preceq \iota' \preceq \mathsf{owner}(\iota)$ implies that either $\iota' = \iota$ or $\iota' = \mathsf{owner}(\iota)$. We call this property (P). Recall also that the containment invariant states $\iota \rightarrow \iota' \Rightarrow \iota \preceq \mathsf{owner}(\iota')$.

Now to prove that all paths from the root of the system to an object pass through that object's owner, we prove a more general proposition:

$$\text{If } \varepsilon \rightarrow \iota_0 \rightarrow \cdots \rightarrow \iota_n \text{ then } \iota' \in \{\varepsilon, \iota_0, \ldots, \iota_n\} \text{ for all } \iota' \succeq \iota_n.$$

We proceed by induction on the length of the path.

Base: Assume $\varepsilon \rightarrow \iota$. By the containment invariant $\varepsilon \preceq \mathsf{owner}(\iota)$ and hence $\mathsf{owner}(\iota) = \varepsilon$. Now consider ι' such that $\iota' \succeq \iota$. This ι' must be equal to either ι or ε by (P), hence it is in the set $\{\varepsilon, \iota\}$.

Inductive: Let $\varepsilon \rightarrow \iota_0 \rightarrow \cdots \rightarrow \iota_n$, where $n \geq 1$, be a path from the root of the system. By the inductive hypothesis, $\iota' \in \{\varepsilon, \iota_0, \ldots, \iota_{n-1}\}$ for all $\iota' \succeq \iota_{n-1}$. Con-

sider $\iota_{n-1} \to \iota_n$. By the containment invariant $\iota_{n-1} \preceq \mathsf{owner}(\iota_n)$, hence $\mathsf{owner}(\iota_n) \in \{\varepsilon, \iota_0, \ldots, \iota_{n-1}\}$. Also $\iota_n \in \{\varepsilon, \iota_0, \ldots, \iota_{n-1}, \iota_n\}$. Now consider ι' such that $\iota_n \preceq \iota' \preceq \mathsf{owner}(\iota_n)$. This ι' must be equal to either ι_n or $\mathsf{owner}(\iota_n)$ by (P), hence it is already in the set $\{\varepsilon, \iota_0, \ldots, \iota_{n-1}, \iota_n\}$. □

Bibliography

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[2] Martín Abadi. Baby Modula-3 and a theory of objects. *Journal of Functional Programming*, 4(2):249–283, April 1994.

[3] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[4] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, 15 March 1996.

[5] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *TAPSOFT '97. Theory and Practice of Software Development*, pages 682–696, April 1997.

[6] Douglas Adams. *The Hitchhikers Guide to the Galaxy*. Pan Books, 1985.

[7] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[8] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[9] Alexander Aiken, Manuel Fähndrich, and Ralph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, San Diego, California, June 1995.

[10] Paulo Sérgio Almeida. Balloon Types: Controlling sharing of state in data types. In *ECOOP Proceedings*, June 1997.

[11] Paulo Sérgio Almeida. *Control of Object Sharing in Programming Languages*. PhD thesis, Department of Computing, Imperial College of Science, Technology, and Medicine, University of London, June 1998.

[12] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions of Programming Languages and Systems*, 15(4):575–631, 1993.

[13] Henry G. Baker. Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages. In *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, pages 218–226, Nice, France, June 1990.

[14] Henry G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4), October 1993.

[15] Henry G. Baker. Iterators: Signs of weakness in object-oriented languages. *OOPS Messenger*, 4(3), July 1993.

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

[17] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science*, pages 88–97, 1999.

[18] Richard Bird and Phil Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[19] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, January 1996.

[20] G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Studentlitteratur, 1979.

[21] Bruno Blanchet. Escape analysis for object-oriented languages: Application to Java. In *OOPSLA Proceedings*, October 1999.

[22] Boris Bokowski. CoffeeStrainer — statically checking structural constraints in Java. Technical Report B-98-14, FU Berlin, Inst. f. Informatik, 1998.

[23] Boris Bokowski. Implementing "Object ownership to order". In *Intercontinental Workshop on Aliasing in Object-Oriented Systems*, At ECOOP'99, Lisbon, Portugal, June 1999.

[24] Boris Bokowski and Jan Vitek. Confined Types. In *OOPSLA Proceedings*, 1999.

[25] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[26] Viviana Bono and Kathleen Fisher. An imperative first-order calculus with object extension. In *ECOOP Proceedings*, 1998.

[27] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *ECOOP Proceedings*, June 1999.

[28] Viviana Bono, Amit Patel, Vitaly Shmatikov, and John C. Mitchell. A core calculus of classes and objects. In *15th Conf. Mathematical Foundations of Programming Semantics*, 1999.

[29] Grady Booch. *Object-oriented Design with Applications*. Benjamin-Cummings, 1991.

[30] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1999.

[31] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, 2000.

[32] John Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 2001. to appear.

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

[34] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA Proceedings*, 1998.

[35] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[36] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Sytems*, 1(3):221–242, 1995.

[37] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing Object Encodings. In *Theoretical Aspects of Computer Software (TACS'97)*, LNCS 1281, pages 415–438, 1997.

[38] Ciarán Bryce and Chrislain Razafimahefa. An approach to safe object sharing. In *OOPSLA Proceedings*, October 1999.

[39] Alex Buckley. Ownership types restrict aliasing. Master's thesis, Department of Computer Science, Imperial College of Science, Technology, and Medicine, Queen's Gate, London, June 2000.

[40] Cristiano Calcagno. Stratified operational semantics for safety and correctness of region calculus. In *28th ACM Symposium on Principles of Programming Languages*, January 2001.

[41] Cristiano Calcagno, Samin Ishtiaq, and Peter W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In *Principles and Practice of Declarative Programming*, 2000.

[42] K. H. S. Campbell, J McWhir, W. A. Ritchie, and I. Wilmut. Sheep cloned by nuclear transfer from a cultured cell line. *Nature*, 380:64–66, 1996.

[43] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[44] L. Cardelli. Obliq: A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995.

[45] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *LNCS*, pages 51–67, Sophia-Antipolis, France, June 1984.

[46] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 78:138–164, 1988.

[47] Luca Cardelli. *Type Systems*, chapter 103, pages 2208–2236. The Computer Science and Engineering Handbook. Allen B. Tucker (Ed.). CRC Press, 1997.

[48] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In *Theoretical Computer Science; Exploring New Frontiers in Theoretical Informatics. International Conference IFIP TCS 2000*, volume 1872 of *LNCS*, pages 333–347, 2000.

[49] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In *CONCUR 2000 – Concurrency Theory. 11th International Conference*, volume 1877 of *LNCS*, pages 365–379, August 2000.

[50] Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In *Foundations of Software Science and Computation Structures, European Joint Conferences on Theory and Practice of Software*, March 1998.

[51] Luca Cardelli and Andrew D. Gordon. Logical properties of name restriction. In *Typed Lambda Calculus with Applications (TLCA)*, September 2001.

[52] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In *Programming Concepts and Methods, IFIP State of the Art Reports.*, pages 479–504. North Holland, March 1990. Also available as Digital Systems Research Center Technical Report SRC-RR-90-56.

[53] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[54] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhäuser, 1997.

[55] Craig Chambers and Gary Leavens. Type soundness for an object-oriented language with multimethods, block structure, and modules. In *Fourth Workshop on Foundations of Object-Oriented Languages*, 1997.

[56] Craig Chambers and Gary T. Leavens. BeCecil, a core object-oriented language with block structure and multimethods: Semantics and typing. Technical Report UW-CSE-96-12-02, Department of Computer Science and Engineering, University of Washington, 1997.

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

[58] David Clarke. An object calculus with ownership and containment. In *Foundations of Object-oriented Programming (FOOL8)*, London, January 2001. Available from `http://www.cs.williams.edu/~kim/FOOL/FOOL8.html`.

[59] David Clarke, James Noble, and John Potter. Overcoming representation exposure. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems*, At ECOOP'99, Lisbon, Portugal, June 1999.

[60] David Clarke, James Noble, and John Potter. Simple ownership types for object containment. In *ECOOP Proceedings*, June 2001.

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

[62] David Clarke, Ryan Shelswell, John Potter, and James Noble. Object ownership to order. Microsoft Research Institute Internal Report, 1998.

[63] W. R. Cook. A proposal for making Eiffel type-safe. In *European Conference on Object-Oriented Programming*, pages 52–72, 1989.

[64] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, California, January 1990.

[65] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *Programming Design and Implementation*, 1999.

[66] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *1999 Symposium on Principles of Programming Languages*, 1999.

[67] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Technical Report SRC-RR-98-156, Compaq Systems Research Center, July 1998.

[68] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedigns of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[69] Jin Song Dong and Roger Duke. The geometry of object containment. *Object Oriented Systems*, 2:41–63, 1995.

[70] Sophia Drossopoulou and Susan Eisenbach. Java is type safe—probably. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 389–418, Jyväskylä, Finland, 9–13 June 1997. Springer.

[71] Jonathan Eifrig, Scott Smith, Valery Trifonov, and Amy Zwarico. An interpretation of typed OOP in a language with state. *Lisp and Symbolic Computation*, 8(4):357–397, 1995.

[72] Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[73] Extended Static Checking for Java home page, Compaq Systems Research Center. On the web `http://www.research.compaq.com/SRC/esc/`.

[74] Butler Lamspon et. al. Report on the programming language Euclid. *SIGPLAN Notices*, 12(2), 1977.

[75] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,* 1993, 26–38.

[76] Kathleen Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, Stanford University, 1996.

[77] Kathleen Fisher and John Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1996.

[78] Kathleen Fisher and John C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–25, 1998.

[79] Kathleen Fisher and John Reppy. Extending Moby with inheritance-based subtyping. In *ECOOP Proceedings*, June 2000.

[80] Cormac Flanagan and Martín Abadi. Types for Safe Locking. In *Programming Languages and Systems*, volume 1567 of *Lecture Notes in Computer Science*, pages 91–107, March 1999.

[81] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *25th ACM Conference on Principles of Programming Languages*, January 1998.

[82] Daniel P. Friedman and Amr Sabry. Recursion is a computational effect. Technical Report TR546, Computer Science Department, Indiana University, December 2000.

[83] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, Washington, 1999.

[84] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[85] Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

[86] David Gay and Alexander Aiken. Memory management with explicit regions. In *1998 Conference on Programming Design and Implementation*, Montreal, Canada, June 1998.

[87] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *9th International Conference on Compiler Construction (CC'2000)*, 2000. Published as Springer-Verlag LNCS 1781.

[88] Daniela Genius, Martin Trapp, and Wolf Zimmermann. An approach to improve locality using sandwich types. In *Types in Compilation '98*. Springer-Verlag, 1998. LNCS 1473.

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

[90] A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. *Journal of Functional Programming*, 9(4):373–426, July 1999.

[91] Andrew D. Gordon. Notes on nominal calculi for security and mobility. In *International Summer School on Foundations of Security Analysis and Design (FOSAD 2000)*, Lecture Notes in Computer Science, Bertinoro, September 2000. Springer.

[92] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *HLCL'98*, Elsevier ENTCS, 1998.

[93] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, July 2000.

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

[95] Peter Grogono and Patrice Chalin. Copying, sharing, and aliasing. In *Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94)*, Montreal, Quebec, May 1994.

[96] Peter Grogono and Markku Sakkinen. Copying and Comparing: Problems and Solutions. In *ECOOP Proceedings*, June 2000.

[97] Peter Grogono and Philip Santas. Equality in object-oriented languages. In *EastEurOOPe'93*, Bratislava, Slovakia, 1993.

[98] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming*. Foundations of Computing. MIT Press, 1994.

[99] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, January 1990.

[100] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[101] Joseph Heller. *Catch 22*. Vintage, 1961.

[102] Laurie J. Hendren and Guang R. Gao. Designing programming languages for analyzability: A fresh look at pointer data structures. In *1992 International Conference on Computer Languages*, pages 242–251, Oakland, California, April 1992.

[103] Ulrich Hensel, Marieke Huisman, Bart Jacobs, and Hendrik Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In *European Symposium on Programming*, number 1381 in LNCS, pages 105–121, 1998.

[104] Trent Hill, John Potter, and James Noble. Visualising implicit structure in object graphs. In *Proceedings of Software Visualisation Workshop, SofVis'99*, Sydney, Australia, December 3-4 1999.

[105] J. Roger Hindley and Johnathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1. Cambridge University Press, 1986.

[106] C. A. R. Hoare and He Jifeng. A trace model for pointers and objects. In *ECOOP'99*, pages 1–17, 1999.

[107] Martin Hofmann. Syntax and semantics of dependent types. In P. Dybjer and A. Pitts, editors, *Semantics of Logics of Computation*. Cambridge University Press, 1997.

[108] Martin Hofmann and Benjamin C. Pierce. A unified type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995.

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

[110] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), April 1992.

[111] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *8th International Parallel Processing Symposium*, April 1994.

[112] Hans Hüttel, Josva Kleist, Uwe Nestmann, and António Ravara, editors. *Workshop on Semantics of Objects As Processes (SOAP '99)*, Lisbon, Portugal, May 1999. BRICS-NS-99-2.

[113] Hans Hüttel and Uwe Nestmann, editors. *Workshop on Semantics of Objects As Processes (SOAP '98)*, Aalborg, Denmark, July 1998. BRICS-NS-98-5.

[114] Yuji Ichisugi and Akinori Yonezawa. Distributed garbage collection using group references. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990.

[115] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA Proceedings*, October 1999.

[116] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, 1999. Also in informal proceedings of the Sixth International Workshop on Foundations of Object-Oriented Languages (FOOL). Full version to appear in *Information and Computation*.

[117] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. In *European Conference on Object-Oriented Programming (ECOOP)*, 2000. Also in informal proceedings of the Seventh International Workshop on Foundations of Object-Oriented Languages (FOOL). To appear in Information and Computation.

[118] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM Symposium on Principles of Programming Languages*, January 2001.

[119] Stuart Kent and John Howse. Value types in Eiffel. In *TOOLS 19*, Paris, 1996.

[120] Stuart Kent and Ian Maung. Encapsulation and aggregation. In *TOOLS Pacific 18*, 1995.

[121] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *20th ACM Symposium on Principles of Programming Languages*, January 1993.

[122] Günter Kniesel and Dirk Theisen. JAC – Java with transitive readonly access control. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems*, At ECOOP'99, Lisbon, Portugal, June 1999.

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

[124] Gary T. Leavens and Olga Antropova. Acl — eliminating parameter aliasing with dynamic dispatch. Technical Report 98-08a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, February 1999.

[125] K. Rustan M. Leino. Data Groups: Specifying the Modification of Extended State. In *OOPSLA Proceedings*, 1998.

[126] K. Rustan M. Leino and Raymie Stata. Virginity: A contribution to the specification of object-oriented software. *Information Processing Letters*, 70(2):99–105, April 1999.

[127] Xavier Leroy and François Rouaix. Security properies of typed applets. In *25th ACM conference on Principles of Programming Languages*, January 1998.

[128] Henry Liebermann. Using prototypical objects to implement shared behaviour in object-oriented systems. *OOPSLA'86 Conference Proceedings, SIGPLAN Notices*, 21(11):214–223, November 1986.

[129] L. Liquori and G. Castagna. A typed lambda calculus of objects. In *Proc. of ASIAN '96, Int. Conf. on Concurrency and Parallelism, Programming, Networking, and Security*, volume 1212 of *LNCS*, Singapore, 1996. Springer-Verlag.

[130] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.

[131] Barbara Liskov and Jeanette Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[132] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 1988.

[133] B. J. MacLennan. Values and objects in programming languages. *ACM SIGPLAN Notices*, 17(12), December 1982.

[134] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kirsten Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[135] Karl Marx and Fredrick Engels. *The Communist Manifesto*. Mass Market, 1998.

[136] Vlada Matena and Beth Stearns. *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform*. Addison-Wesley, December 2000.

[137] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[138] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[139] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Compter and System Science*, 17:348–375, 1978.

[140] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100:1–77, September 1992.

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

[142] Greg Morrisett. Refining first-class stores. In *ACM SIGPLAN Workshop on State in Programming Languages*, June 1993.

[143] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[144] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.

[145] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[146] Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe—definitely. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 161–170, San Diego, California, 19–21 January 1998.

[147] James Noble. Iterators and Encapsulation. In *TOOLS Europe 33*, pages 431–442, Mont St-Michel, La Belle France, June 2000.

[148] James Noble, David Clarke, and John Potter. Object ownership for dynamic alias protection. In *TOOLS Pacific*, Melbourne, Australia, November 1999.

[149] James Noble, Antero Taivalsaari, and Ivan Moore, editors. *Prototype-based Programming: Concepts, Languages, and Applications*. Springer-Verlag, 1999.

[150] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98— Object-Oriented Programming*, volume 1445 of *Lecture Notes In Computer Science*, pages 158–185, Berlin, Heidelberg, New York, July 1988. Springer-Verlag.

[151] Robert O'Callahan and Daniel Jackson. Lackwit: a program understanding tool based on type inference. In *1997 International Conference on Software Engineering*, Boston, USA, May 1997.

[152] Martin Odersky. A functional theory of local names. In *21th ACM conference on Principles of Programming Languages*, January 1994.

[153] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[154] P. W. O'Hearn, A. J. Power, M. Takeyama, and R. D. Tennent. Syntactic control of interference revisited. In *Mathematical Foundations of Programming Semantics, Eleventh Conference*, 1995.

[155] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

[156] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[157] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[158] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of

Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[159] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[160] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.

[161] Alan Pope. *The CORBA Reference Guide*. Addison Wesley, 1998.

[162] John Potter, David Clarke, and James Noble. A mode system for flexible alias protection. In *Formal Methods Pacific '98*, Canberra, Australia, September 1998.

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

[164] Didier Rémy. From classes to objects via subtyping. In *European Symposium On Programming*, volume 1381 of *Lecture Notes in Computer Science*. Springer, March 1998.

[165] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[166] Didier Rémy and Jérôme Vouillon. The reality of virtual types for free! Available from http://pauillac.inria.fr/∼remy, October 1998.

[167] John C. Reynolds. Syntactic control of interference. In *5th ACM Symposium on Principles of Programming Languages*, January 1978.

[168] John C Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983.

[169] John C. Reynolds. Syntactic control of interference, part 2. In *Automata, Languages, and Programming: 16th International Colloquium*, number 372 in LNCS, pages 704–722, 1989.

[170] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*. Palgrave, 2000.

[171] Jon G. Riecke and Christopher A. Stone. Privacy via Subsumption. In *Fifth Workshop on Foundations of Object-Oriented Languages*, 1998.

[172] Dale Rogerson. *Inside COM*. Microsoft Press, 1996.

[173] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8), 1967.

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

[175] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *1992 ACM Conference on LISP and Functional Programming*, pages 288–298, San Francisco, CA, June 1992. ACM.

[176] Derek Santibáñez. Visualisation of dynamic ownership trees in evolving object graphs. Honours Thesis, School of MPCE, Macquarie University, 1998.

[177] A. J. H. Simons. Rationalising Eiffel's type system. In C. Miggins, R. Duke, and B. Meyer, editors, *18th Conference on Technology of Object-Oriented Languages and Systems*, Melbourne 1995.

[178] Anthony J. H. Simons. Borrow, copy or steal? Loans and larceny in the Orthodox Canonical Form. In *OOPSLA Proceedings*, 1998.

[179] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Program*, Berlin, Germany, March 2000.

[180] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *Proceedings of OOPSLA '86, ACM SIGPLAN Notices*, 21(11):38–45, 1986.

[181] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, ACM Press, 1998.

[182] Antero Taivalsaari. Kevo, a prototype-based object-oriented programming lnaguage based on concatenation and module operations. Technical Report Report LACIR 92-02, University of Victoria, 1992.

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

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

[185] Dirk Theisen. Enhancing encapsulation in OOP: A practical approach. Master's thesis, Institut für Informatik III, Römerst. 164, D-53117 Bonn, Universität Bonn.

[186] Mads Tofte and Lars Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):734–767, July 1998.

[187] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, January 1994.

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

[189] Mads Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, San Diego, CA, January 1998.

[190] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In J. Jouannaud, editor, *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architectures, Nancy, France*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, New York, NY, September 1985.

[191] David Ungar and Randall B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–205, 1991.

[192] Mark Utting. Reasoning about aliasing. In *The Fourth Australasian Refinement Workshop*, 1995.

[193] David Walker and Greg Morrisett. Alias types for recusive data structures. In *Workshop on Types in Compilation*, Montreal, Canada, September 2000. Avaliable as Carnegie-Mellon University Technical Report CMU-CS-00-161.

[194] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *28th ACM Symposium on Principles of Programming Languages*, January 2001.

[195] Wolfgang Weck and Clements Szyperski. Do we need inheritance? In *Workshop on Composibility Issues in Object Orientation (at ECOOP'96)*, 1996.

[196] Alan Wills. Reasoning about aliasing. In *ECOOP Proceedings*, 1993.

[197] Mario Wolczko. Semantics of Smalltalk-80. In Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, *ECOOP'87 European Conference on Object-Oriented Programming*, volume 276 of *Lecture Notes in Computer Science*, pages 108–120, Paris, France, 15–17 June 1987. Springer.

[198] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[199] Bennett Norton Yates. A type-and-effect system for encapsulating memory in Java. Master's thesis, Department of Computer and Information Science and the Graduate School of the University of Oregon, August 1999.

[200] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: A syntactic proof technique. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, Paris, France, September 1999.

[201] Silvano Dal Zilio and Andrew D. Gordon. Region analyis and a π-calculus with groups. In *25th International Symposium on Mathematical Foundations of Computer Science, MFCS 2000*, Bratislava, Slovak Republic, August 28–September 1 2000.