Inferring Ownership
Final Year Project Report

Alisdair Wren
Supervised by Dr Sophia Drossopoulou

Second Markers: Dr Paul Kelly and Dr Nobuko Yoshida

June 19, 2003
Ownership Types provide an effective means by which to restrict or specify aliasing in object oriented programs. While type-checking an ownership-typed program is well understood and while there is a large body of literature exploiting static ownership information, there is no method for inferring appropriate ownership types from untyped program text. The goal of this project was to produce a formal system for reasoning about ownership from programs and program data without any pre-determined ownership information.

Static ownership has a natural run-time definition in terms of paths, via references, in a program’s object heap. As a means for studying a program’s ownership characteristics, a formal system was developed for reasoning about the properties of object graphs derived from program heaps. Using this system, a number of results were obtained with important implications for ownership type inference, as well as for other potential uses of dynamic object ownership information. Most importantly, this project characterises the potential ownerships that will be acceptable for a program heap, and shows how to obtain the most precise – hence useful – object ownership information.

Subsequent adaptation of these run-time ownerships is then specified, followed by refinement through a system of constraints derived from the static ownership type system. Finally, this highly constrained set of potential ownerships is then expanded to statically-checkable program text, annotated with ownership types.
Acknowledgements

My particular gratitude to Dr Sophia Drossopoulou for her time, encouragement, patience and good humour, all of which I enjoyed in equally lavish measures.

Many thanks to those of the SLURP research group, especially to Dr Susan Eisenbach for her unwavering will and ability to advise, and to Dr John Potter and Dr Dave Clarke – founding members of the ownership types community – for their insights and comments when visiting the group. I am grateful also to the PhD Students of SLURP for their enthusiasm, impressions and esoteric vocabularies.

My thanks also to Dr Paul Kelly, who provided considerable inspiration at the outset of this project and who, as my personal tutor over the years, taught me anything that I know about good research.

Finally, thanks to all my colleagues in The MEng Ring of Knowledge, an unending source of diverse computer science expertise, and to Will for his intellect and forbearance.
# Contents

1 Introduction ........................................ 4
   1.1 Motivation ................................... 5
   1.2 Contribution ................................ 5
   1.3 This Report .................................. 6

2 Background ........................................ 7
   2.1 Object-Oriented Programming ................. 7
   2.2 Reasoning About Object Oriented Programs .... 8
   2.3 Aliasing ..................................... 9
      2.3.1 Aliasing in Imperative Languages ....... 9
      2.3.2 Aliasing in Object Oriented Languages ... 11
      2.3.3 Aliasing and Aggregate Data Structures ... 11
   2.4 Tackling Aliasing ............................... 13
      2.4.1 Alias Detection ........................... 13
      2.4.2 Program Annotations ...................... 15
   2.5 Type Systems ................................ 17
      2.5.1 Notation ................................ 19
      2.5.2 Typing Functional Programs ............... 19
      2.5.3 Typing Imperative Programs ............... 21
      2.5.4 Type Systems for Object-Oriented Languages .... 21

Alisdair Wren ........................................ 3
Inferring Ownership

2.5.5 Type Checking vs Type Inference .......................... 22
2.6 Summary ....................................................... 23

3 Ownership Types .............................................. 24
  3.1 Object Ownership ............................................ 24
    3.1.1 Object Heaps ............................................ 24
    3.1.2 Ownership Diagrams .................................... 25
  3.2 Ownership Contexts ......................................... 26
  3.3 The Ownership Invariant ................................. 27
    3.3.1 Owners as Dominators .................................. 27
  3.4 Using Ownership Types ................................. 28
  3.5 Context Substitution ................................. 31
    3.5.1 Run-time Context Substitution: \( \tau \) ............ 31
    3.5.2 Static Context Substitution: \( \sigma \) ............. 32
  3.6 Enforcing Owners-as-Dominators .......................... 32
    3.6.1 Type Rule Environments ............................. 32
    3.6.2 Valid Owners ........................................... 33
    3.6.3 The Update Rule ........................................ 33
  3.7 Summary ....................................................... 34

4 Motivation and Plan ........................................... 35
  4.1 Motivating Ownership Inference ............................. 35
  4.2 Motivating the Formal System ............................. 36
  4.3 Dynamic Inference ......................................... 37
  4.4 Inference Strategy ....................................... 38
    4.4.1 Overview .............................................. 38
    4.4.2 Obtaining Object Heaps ............................ 38
    4.4.3 Extended Object Graphs ............................ 40

Final Year Project
## Contents

4.4.4 Domination ........................................... 40  
4.4.5 Ownership ............................................ 40  
4.4.6 Ownership-Type Consistent Ownerships ................. 41  
4.4.7 Ownership Parameter Substitutions ....................... 41  
4.4.8 Type Propagation and Checking .......................... 41  
4.5 Summary .................................................. 42  

5 Introduction to the Formal System 43  
5.1 Notation .................................................. 43  
5.2 Object Graphs ............................................. 44  
5.2.1 Auxilliary Information in XObject ....................... 45  
5.2.2 Primitive Properties of Extended Object Graphs ....... 46  
5.3 Paths in Object Graphs ................................... 48  
5.3.1 Reachability and Well-formed Object Graphs ........... 50  
5.4 Graph Combination ....................................... 51  
5.5 Design Considerations ..................................... 53  

6 Domination in Object Graphs 55  
6.1 Specifying Domination ..................................... 55  
6.2 Properties of Domination ................................ 58  
6.2.1 dom is a Partial Order ................................. 58  
6.2.2 Root Dominates All ..................................... 64  
6.3 Dominator Functions ...................................... 65  
6.4 Domination under Graph Combination ....................... 65  
6.5 Design Considerations ..................................... 66  
6.5.1 Monotonicity of Domination under Graph Combination .... 66  
6.5.2 Reflexivity ............................................. 68  
6.6 Summary .................................................. 68  

Alisdair Wren 5
9.2.1 Instantiating this and owner ......................... 107
9.2.2 Parameter Counts ........................................ 107
9.2.3 Imported Contexts ....................................... 107
9.3 Inferring Static Substitutions ............................... 108
9.4 Inferring Types ............................................. 110
  9.4.1 Type Propagation ....................................... 111
9.5 Conclusions ................................................. 111
9.6 Summary ..................................................... 112

10 Conclusion .................................................... 113
  10.1 This Project’s Contribution ............................... 113
  10.2 Further Work ............................................. 114
  10.3 Remarks ................................................... 115

A  A Minimal Object-Oriented Language with Ownership 118
  A.1 Syntax ..................................................... 118
  A.2 Operational Semantics ................................... 121
  A.3 Soundness ................................................. 123

B Program Annotations ........................................... 128
Object oriented programming offers important opportunities to structure programs so that they better represent the environments in which they must operate. This improved, more natural structure should improve the power that program reasoning can offer when studying program properties such as safety and correctness. However, the potential for object aliasing is a fundamental obstacle to effectively reasoning about object oriented environments[15].

Ownership types[7] offer a means for expressing aliasing characteristics of a program in a similarly natural way to the underlying object-orientation of the language. The family of type systems proposed by Clarke et al[5, 6, 7] provides access control for member objects of an aggregate structure rather than the per-field access controls provided by traditional object-oriented languages.

Ownership type systems allow the declaration of aggregate structures with well-defined boundaries, through which references cannot pass. Thus an external object may not manipulate the contexts of an aggregate, for example a linked list, without the knowledge of the object which owns that structure: All heap references into that structure must go via the owner, a property that is invariant with program execution. Ownership types define a cheaply, statically checkable system of program annotations that enforce this property, so annotated programs are bestowed with information that immediately partitions its state into distinct portions. This separation dramatically cuts down on the aliasing possibilities that need to be considered and, when added with information about how code affects these distinct portions[6], the effort required for program reasoning is dramatically cut[3, 11].
1.1 Motivation

Current literature assumes that the programmer will annotate their code with ownership type information in order to reap the benefits offered by ownership type systems. However, there is an enormous body of legacy code for which the benefits of ownership typing could be exploited. It would be ideal, therefore, if it was possible to infer ownership types for code in much the same way as more traditional data-type information can be inferred already[10, 20].

Unlike traditional situations, however, we have an unbounded number of variables to consider, as the ownership type system relies on a system of parameters that have no immediate relation to the program text\(^1\). We therefore make use of information inferred from observations of program execution in order to determine how ownership is exhibited in a program’s object heap.

Inference of ownership types is not the only target for such work – there is a variety of uses for dynamic ownership information in object heaps, such as garbage collection[4] and program visualisation[14]. Many of these applications rely on absolute correctness and benefit from the best possible precision.

1.2 Contribution

This report, therefore, gives a detailed formal system for the inference of reliable ownership information from run-time object heaps, and goes on to show how such information might be used to infer ownership types for unannotated program text.

We present a system for reasoning about the invariants of ownership across multiple object heaps, and address the properties of domination in object graphs. As the system is developed, we show how domination and ownership relate, and how to calculate the most precise ownership invariant ownership from object graphs. In order to provide a solid foundation for the further work, all the required properties are proved on the formal system, which we also show is compatible with the intuitions underlying ownership types.

In order to use this information for the type inference, we define a mechanism to adapt the ownerships we derive so that they conform with requirements imposed by the type system. We narrow the search for a valid type through a series of constraints, each motivated by the type system and the program code for which types are required. Finally, we suggest a means for inferring annotations to the program text from the highly constrained selection of possible program typings.

\(^1\)Contrast this situation with, for example, determining the number of method parameters.
1.3 This Report

In Chapter 2, we discuss object oriented programming, introduce the problems caused by object aliasing and review some of the previous work on overcoming them. We also give a brief overview of type systems from the functional, imperative and object-oriented worlds. We focus on the details of ownership types in Chapter 3, where we show how ownership is specified and enforced, and give some explanation for the structures underlying the type system. Chapter 4 motivates the work in greater detail, and presents the overriding strategy for the formal work. We show how the inference of ownership has potential both for the inference of ownership types, and for a number of other separate applications. An overview of the type inference is presented, where we discuss how the inference is a set of progressively narrowing constraints on possibly valid ownerships, and how these constraints should be applied.

The body of the report presents the formal system, including its definition and some proofs of its most important properties. Chapter 5 introduces notation and the basis of the formal system, which is used in Chapter 6 to study the properties of domination in object graphs. A number of important properties are shown, including the difficulties associated with the use of domination for the inference of ownership. In Chapter 7, therefore, we give a more detailed examination of ownership, how it can be derived from domination information and how to obtain the most precise ownership information. Acknowledging that this ownership information will need to be adapted due to the inevitable loss of precision when moving from the dynamic world to static code, we present a means to do so in Chapter 8 and prove that it is correct. To conclude the formal work, we give suggestions for using these constrained ownerships to infer the type annotations for addition to the program text in Chapter 9.

The report is concluded in Chapter 10, where we summarise the formal work, evaluate its contribution and discuss the potential for the system’s development.
Chapter 2

Background

This chapter gives a brief overview of object oriented programming and the problems associated with reasoning about object-oriented programs. We observe how object aliasing can frustrate such reasoning, and present some of the solutions that have been proposed. Finally, we observe how type systems are used to place static constraints upon programs’ run-time behaviour and how these are applied in object-oriented situations.

2.1 Object-Oriented Programming

Many credit the first appearance of object-oriented ideas to Simula, a simulation language developed in the sixties by Dahl and Nygaard[8]. While the first version of the language, now known as Simula-I, may seem unusual to modern object-oriented programmers, the language’s next incarnation – Simula-67 – included the familiar class and new constructs[9, 19].

The work of Alan Kay et al[18] on Smalltalk took object-orientation to extremes – everything in the language was an object. Communication between objects was accomplished with messages, and the responses to these messages were defined by object methods[16]. These terms are all common to more recent object-oriented languages.

Over the following two decades, object oriented programming developed substantially, with an explosion of object-oriented languages. ‘Object-oriented’ became a buzz-phrase among programmers and these techniques have undoubtedly become one of the most popular paradigms for modern software development.

Object oriented programming has a definite advantage over other programming models: It allows software designers to represent more explicitly a program’s environment and applicability. Examples are countless: Personnel management systems can represent each person by a single object; stock markets can have an object for each company; more abstractly,
Inferring Ownership

objects can represent relationships between other objects such as marriage between people or contracts between companies. Object oriented ideas have percolated from software engineering through most other computing disciplines: Databases, networking, simulation, artificial intelligence and even theoretical computer science, which may have very little relation to any real program code. Most of those communities would regard the impact of object oriented techniques positively.

As an aside, to have had such wide-ranging applicability, there must be something fundamental about object-orientation: Indeed, an object is just a record (or C struct or, more formally, a tuple) which has methods associated with it – this seems little different from a procedure (or function) which takes a record as an argument. Thus, it may be that what other computing communities embraced from the software engineers and language designers is simply a revival and new notation for what already existed. Certainly, there must be something more to object-orientation than the use of data with associated methods. This project restricts its attention very much to the problems facing object oriented programming languages, but this object oriented property of a language is still poorly defined. Stroustrup, instrumental in the development of C++, succinctly defined object oriented programming as “programming with inheritance”[22], acknowledging that many languages already possessed the data abstraction\(^1\) and user-defined type facilities that might still be used to manipulate ‘objects’. In the same paper, he indicates that a programming language can only be object oriented if it encourages the definition of classes or objects and the use of inheritance to express the commonality between types so often found in the real world. This is in contrast to those languages which merely allow the use of object-like structures through esoteric use of language features. While Stroustrup makes much of inheritance, he presumably finds delegation acceptable in the case of object-based\(^2\) languages.

2.2 Reasoning About Object Oriented Programs

Reasoning about programs and programming languages is vital if the computing community is to provide reliable software. Unchecked software has already cost huge sums of money and, in several circumstances, a number of lives\(^3\). Subsequent analysis of the software has often detected preventable flaws in the system, yet this analysis is generally avoided due to its relatively high cost (as exemplified by Jackson in [17]).

Additionally, automatic program analysis underlies almost every compiler optimisation

---

\(^1\)By ‘data abstraction’, Stroustrup refers to the provision of abstract data type modules, which define procedures for creating and destroying members of the abstracted type, and performing operations on those structures[22]. In this case, user-defined types are treated very differently to built-in types.

\(^2\)Object-based languages are different from (though usually a subset of) object-oriented languages. This report assumes class-based languages.

\(^3\)There are a number of classic examples related to the costs of poorly assured software: Therac-25 was a computerised radiotherapy machine used in the mid-1980s; a software failure resulted in massive overdoses for a number of patients, resulting in death and serious injury. In 1996, the European rocket Ariane-5 was forced to self-destruct due to unchecked use of software transferred from another vehicle.
technique. Without the intuition of a human to guide its optimisations, a compiler must always make the most conservative assumptions about anything it cannot prove, in order to avoid changing the meaning of the code it is translating.

When analysing a program, of course, the presence of objects can be completely ignored, as any object oriented program can be converted automatically into a procedural program (compilation to machine code for example, where storage has no structure and everything is global). However, there are several advantages to acknowledging a program’s class or object structure when modelling behaviour. Object-orientation provides a convenient partitioning of behaviour – a system’s components can have their semantics or properties determined independently, such that their properties can be determined in whatever environment an object is used. This is not so different from function-by-function analysis based at each call site, but more structure generally yields more invariants, which tend to simplify analysis.

In the ideal case, objects completely encapsulate data. It would be preferable if, regardless of the ‘outside world’, invariants maintained by an object’s methods are never violated. Unfortunately, this is not the case in most real languages: Both C++ and Java provide access control on object data members, but there is no restriction to prevent an object’s method returning a reference to private data. This is a primitive example of aliasing, which is discussed more fully in the following section. Analysis of such objects, and therefore all objects in general, can no longer take place independently of their environment.

2.3 Aliasing

Aliasing is, at its simplest, the situation in which there exist two distinct references to the same piece of data.

2.3.1 Aliasing in Imperative Languages

Generally, data is never aliased by default. In C/C++, data can be declared in global variables, as local variables allocated in a function’s stack frame, or allocated by the new operator or a malloc-like function call. None of these allocation methods generate implicit aliases⁴; at least, not within the context of the program⁵.

Figure 2.1 shows some examples of how aliases might be generated. In figures 2.1(a)–2.1(c), there is likely to be a reason for the generation of the alias. For example, passing large structs by value in function calls yields poor performance, so passing a pointer provides

---

⁴Using malloc to allocate an n-byte buffer, then casting it to a pointer to a large (> n-byte) data structure and reading elements from that structure may in fact read from other program structures. These considerations are rather too detailed, and one would hope the program would raise a fault and be terminated in this case.

⁵The operating system presumably keeps track of a program’s allocated data, but we must rely on a reliable operating system that doesn’t modify a running program’s data.
Inferring Ownership

```c
int a = 5;
int *b;
b = &a;
(*b)++;
```

(a) Basic aliasing

```c
int *a, *b;
a = malloc(sizeof(int));
b = a;
(*b)++;
```

(b) Anonymous aliasing

```c
class A {
private:
    int i;
public:
    int *alias()
    {
        return &i;
    }
}
```

(c) Array aliasing

```c
int a[10];
int *b;
b = &a[5];
b[4]++;
```

(d) Private data aliasing

```c
A* a = new A;
*(a->alias())++
```

Figure 2.1: These examples, written in C/C++, illustrate how aliases might be generated. In (a), `(*b)++` affects the contents of both `*b` and `a`. There, `a` can be considered as the aliased data’s ‘name’. Code segment (b) demonstrates the same effect, but neither `a` nor `b` are names for the data which is changed by the final line, though they both alias it. In (c), `b[4]++` also affects `a[9]`. Detecting this type of aliasing can be particularly troublesome, especially when arrays are indexed by loop induction variables. Finally, (d) shows how private class data can be made mutable by the outside world – `*(a.alias())++` affects the internal, private data member `a.i`
a faster alternative. Array aliasing is often used when manipulating strings and for common algorithms from Quicksort to the Fast Fourier Transform. More importantly, passing pointers to specific functions or classes gives more fine-grained control over permissions on the data that is being referenced. This is particularly the case for code in the form of figure 2.1(d), where the data marked `private` must also be modified by a class, B, which has an intuitive relationship to class A, but where that relationship cannot be expressed by the language’s normal access modifiers. This is an object-oriented example, but C++’s access modifiers are similar to C’s global/local restrictions: Passing pointers into functions allows data to be modified by specific functions without making that data global.

2.3.2 Aliasing in Object Oriented Languages

While aliasing in imperative languages is common, there is an argument that the object-oriented programming style encourages the creation of aliases that are not necessary for the program’s operation. This is particularly the case for JAVA-like languages, where everything excluding primitive data types is an object reference. In that case, it becomes more intuitive to distribute object references without taking copies of the objects. C++ is only slightly different as it has the facility for variables to store objects directly, while pointers to objects must be explicitly declared. However, most objects are created using the `new` operator, and references can be implicitly created using C++’s pass-by-reference declarations for function parameters.

Polymorphism is key to fully exploiting object-oriented programming but also one of the greatest sources of object aliasing. Polymorphism allows variables with a type corresponding to one class to hold a pointer to an object derived from any of its subclasses. As an example, Stroustrup indicated in [22] that the use of virtual methods – which are exploited through polymorphism – can avoid the need for `type fields` for each object, specifying the specific behaviour of that object. This is a powerful design technique, and allows great generality for large parts of a software system, pushing specialised code into objects’ member functions. However, this technique can only be used with pointers or references to objects and naturally leads to an increase in potential aliases.

2.3.3 Aliasing and Aggregate Data Structures

With object-oriented programming comes the ability to build complicated arrangements of objects and object references designed to express structure in the objects’ data. Figure 2.2 demonstrates one of the simplest and most common aggregate data structures: a linked list for ordering objects of type `Item` (which can have any definition). Whenever an object of type `Item` (or a subclass) is added to a `LinkedList` object, it is aliased – both the caller of `add` and the `value` member of some `Link` object will hold a reference to that data.

This kind of behaviour is often desired; for example, when establishing several different orderings on one set of `Item` objects. However, suppose the `Item` objects held numbers
class Item {
  ...
}

class Link {
  public Link next;
  public Item value;

  public Link(Item i) {
    value = i;
  }

  public add(Item i) {
    if (next == null) {
      next = new Link(i);
    } else {
      next.add(i);
    }
  }
}

class LinkedList {
  protected Link first;
  protected void add(Item i) {
    if (first == null) {
      first = new Link(i);
    } else {
      first.add(i);
    }
  }

  public Item get() {
    if (first == null) {
      return null;
    } else {
      Item r = first.value;
      first = first.next;
      return r;
    }
  }
}

Figure 2.2: This example code illustrates how a linked list (first-in, first-out) might be implemented in a Java-like language. Any list constructed using this code will be an aggregate of objects derived from Link, LinkedList or Item, or any of their subclasses. Note that any object of type Item given to LinkedList.add will be implicitly aliased – if it is modified, the contents of the list will also change. If this code is in a class library, for example, the programmer may not be aware of this behaviour.
and some larger aggregate using the LinkedList assumed that all numbers in the list were
ordered – this would not be unreasonable if they were added to the list in order! If this larger
aggregate later exported one of these Item objects, perhaps in the manner of figure 2.1(d),
the receiving code could modify the object and break the assumed invariant on the list
without any knowledge or intervention by the enclosing data structure.

In more complicated systems, this modification of objects internal to an aggregate can be a
security hazard if modification is deliberate, or impact the correctness of the program where
an alias has been unwittingly made available. In our simple list example, the Item object
should be cloned before being exported to an untrusted object, which would preclude any
internal invariants of the aggregate being violated.

2.4 Tackling Aliasing

There is a variety of methods for dealing with the aliasing problem, which fall into two
broad categories: Firstly, compiler (or formal) analyses can be extended to calculate how
objects may be aliased during program execution, and the results fed into any subsequent
analyses. This is certainly the approach taken in general software production. Secondly,
programs may be annotated to indicate how aliasing might occur in the program. There
are some examples of this approach, but use is often limited to performance-critical pieces
of code where the programmer wishes to encourage compiler optimisation.

The two approaches, while different in implementation and, potentially, their accuracy, are
parts of the same idea – they both work to determine the aliasing relationships in the
program.

Hogg et al. proposed four approaches in their “Geneva Convention on the Treatment of
Object Aliasing”[15]: Detection, advertisement, prevention and control. Control is per-
formed at run-time and is not, therefore, of particular interest here. We consider detection
separately, while advertisement and prevention fall into the ‘program annotation’ category:

2.4.1 Alias Detection

Alias detection is an extremely important topic, and has a great deal of literature asso-
ciated with it. Even in a running system, aliasing is sometimes difficult to spot: Pointer
comparison is not always available to the programmer, and object mobility can make even
such comparisons unreliable. Determining aliasing relationships on static code – usually
through pointer alias analysis – is even more difficult.
Inferring Ownership

Pointer Alias Analysis

The goal for pointer alias analysis is to produce some kind of graph indicating which pointers are never, sometimes, or always aliases. In section 2.3.1, it was noted that a program’s data is usually completely un-aliased as it is allocated, so examining points at which aliases may be generated (and destroyed) is clearly key to determining the aliasing relationships of data in the system. Spotting aliases generated by statements of the form found in figures 2.1(a) and 2.1(d) is relatively easy. However, cases such as figure 2.1(b) are much more difficult, as it is unclear how to represent what a and b point to. In the case of arrays, as in figure 2.1(c), it is customary to treat the entire array as a single piece of data – precision is lost, but the cost of analysis is greatly reduced.

There are a number of schemes for resolving this naming problem[21], based on allocation site or procedure naming, but these attempts have not always been successful. An alternative, shape analysis, has been used to obtain a more abstract impression of how data structures are shaped on the heap, but there is some concern about how these analyses might scale to real programs[13].

Flow sensitivity and context sensitivity are two aspects which have a profound effect on the expense of pointer alias analysis. A flow sensitive analysis will take control-flow within procedures into account when calculating its results, while a context sensitive analysis will take the various calling-contexts of each method into account. Sensitivity to these aspects of the program can sometimes increase the precision of the result, with fewer spurious aliasing relationships, but may be intractable. However, Whaley at al have presented a context-sensitive analysis which analyses each method only once, producing a parameterised aliasing scheme for each method[23]. Rather than re-analyse the method for each of its calling contexts, the parameterised result can be fleshed-out with details of the context in order to produce the desired result.

Escape Analysis

Escape analysis is particularly useful for circumstances such as that in figure 2.1(d), where i can be said to escape the class A. Objects can escape classes where either they are readable by other objects, or where a reference to the object is returned from a method call. Objects can escape methods where the object was declared as a local variable to the method, and subsequently returned.

The Whaley-Rinard analysis[23] also provides an escape analysis, and there are strong ties between this and pointer alias analysis. However, where it is possible to prove that objects fail to escape their surroundings, some of the modularity of analysis offered by object encapsulation reappears. Applications of escape analysis have included:

6This can include references assigned to objects passed into the method by reference and other exotic variations on the same theme.
**Removal of object locks:** Where objects are shown never to escape their thread, synchronisation locks imposed in concurrent Java programs can be removed, along with their processing overheads.

**Stack-allocation of objects:** Where objects are shown never to escape their method, those objects can be allocated on the stack rather than waiting for Java’s garbage collection.

Thus, results of escape analysis are of particular interest to both the performance optimiser and the formal analyser.

### 2.4.2 Program Annotations

There are a number of schemes to annotate programs in a way that gives an analysis more information about aliasing, or which provide easily, statically checkable constraints on aliasing.

**C99: restrict and noalias**

The best known of these is perhaps the `restrict` keyword, officially introduced in the C99 standard[1] but available in C and C++ when using gcc and other optimising compilers. `restrict` is a qualifier on pointer types, and specifies that no reference to the pointer’s target can be made without using, directly or indirectly, the pointer’s value[1]. In other words, there are no aliases for pointers qualified with `restrict`.

The `restrict` keyword was actually preceded in the C99 committee’s deliberations by `noalias`, which was an object qualifier rather than a part of a pointer type. It specified that only one pointer existed to the qualified object, but was dropped after a number of protestations from various quarters about the impracticality of supporting library code annotated with `noalias`.

Both `noalias` and `restrict` have the disadvantage that they can give rise to ‘undefined’ behaviour when misused, and the standard does not imply that a compiler should try to check that the code obeys the constraints expected for restricted pointers. For example, where a function of type `void f (int * restrict p, int * restrict q)` is called using `f(a, a)`, the behaviour is undefined. Moreover, compilers are free to completely ignore these annotations after parsing.

**LCLint**

A stronger kind of annotation for aliasing exists using the structured-commenting system of LCLINT (now SPLINT). LCLINT provides annotations for expressing a range of pointer
Inferring Ownership

behaviour, such as the uniqueness of pointers, whether data allocated inside methods is returned to the caller and how data is referenced through mutable function parameters[12].

The LCLint program can be used as a preprocessor, and statically checks all of the assertions made using the annotations (as well as some other properties which may indicate bugs). As a preprocessor, of course, it suffers the disadvantage that the results cannot be used to optimise the code in any way.

Additionally, while the addition of such annotations is in many ways incremental, experience shows that changes may be necessary throughout a program when annotating just one function; usually the annotation will impact those required on other functions occurring in the annotated function’s call-stack. This is due to the static checks which prevent unconstrained pointers being assigned to those which are more specifically constrained. restrict, on the other hand, can be applied to individual routines to give the compiler hints specific to a single algorithm without impact on other code.

Data Types and Aliasing

Most programming languages – SMALLTALK is a notable exception – have their variables and expressions annotated with a data type. This data type is used to constrain the kind of data that a variable can hold in order to enforce some basic and desirable invariants which prevent arbitrary operations on inappropriate data. This is extended in object-oriented languages to ensure that the object contained in a variable implements a method for the messages that the variable may be expected to receive. This is particularly important in the case of dynamically allocated objects, where increased indirection (via pointers or references) further dissociates a variable’s type from the methods implemented by its target. This topic is discussed in much greater detail in section 2.5.

In the context of aliasing, the type system can help determine where aliases are impossible: In a “strictly” typed language, where variables can hold items of a single type and nothing else and where type conversion is disallowed, it can be safely assumed that a variable of one type will never alias a variable of another type. However, such a language would not be particularly interesting, nor, according to Stroustrup[22], may it even be considered object-oriented.

In most real object-oriented languages, however, subclass polymorphism can make these assumptions more difficult. Figure 2.3 illustrates the relationship more clearly, showing how languages with such polymorphism accept the use of a subclass where a variable or parameter is typed with its superclass. In JAVA-like languages, we can assume that variables of type B cannot alias variables of type C or A, but that variables of type A may alias variables of type A, B and C. As aliasing is, to all intents and purposes, a symmetric relationship – even given the implicit priority one might award to named variables rather than heap pointers – it is usually accepted that B can actually alias references of type A as well.

C++ complicates matters further, with much more liberal type conversion and pointer
A a; B b; C c;

a = new A // trivially legal
a = new B // legal: B subclass of A
a = new C // legal: C subclass of A
b = new A // illegal
c = new B // illegal

Figure 2.3: A class inheritance structure, where B and C are subclasses of A. The code on the left is in a Java-like language, where everything is assigned by reference. The various assignments demonstrate how subclasses can be assigned to variables of their own or a superclass’ type, but that all other assignments are illegal.

arithmetic available, such that the type system gives far fewer assurances about object aliasing.

2.5 Type Systems

Data types are the most common program annotation commonly in use and, as mentioned in the previous section, have already been used to aid alias detection. This section examines data types in more detail, as they form the basis of the ownership type system on which this relies.

In the previous section, types were briefly mentioned as a means for getting more information from the program when analysing it statically. There is a huge body of literature and theory on the subject of type systems, so only a small subset will be summarised here.

In many cases, it is helpful to ensure that data in a program is of a certain type. For example, when writing a mathematical function, it is helpful to ensure that the function can only be given numerical arguments rather than strings. To this end, programs can be annotated with type information (the program can be typed) specifying the types of data, variables, function parameters and so on. These annotations may be completely arbitrary, so a typed program should be type checked to ensure that the typing is valid – validity is specified using a collection of rules known as type system. In the example given above, one might annotate the function’s parameters with a numeric type and then hope that the type system contained a rule prohibiting the use of a string for such an argument. A program that type-checks successfully is said to be type-correct.

Type systems are based on the syntax of the program, so even the most liberal type system will have only the set of syntactically-correct programs in its set of type-correct programs. In reality, the type system is much more restrictive than this, as demonstrated by figure 2.4, as there will be syntactically correct programs that ‘work’, but which do not follow the typing rules of the system. A classic example, adapted here from [20], shows a program which
Inferring Ownership

(a) The relationship between syntax and typing rules

```plaintext
var v : Integer
v := 0
if v == 0 then v := 1 else v := 'c'
```

(b) A syntactically-correct, possibly type-incorrect program

Figure 2.4: For any type system, the set of type-correct programs will be a subset of ‘acceptable’ programs which would otherwise work perfectly.
may not be type-checkable in the common situation where the type system requires both
consequents of an if-statement to have the same type. Such a rule cannot be satisfied,
though the program is clearly correct as the else-branch is never executed – it is dead code.
Dead code contributes nothing to the program’s operation and yet routinely interferes with
typing. However, its detection is undecidable in the general case, so type systems cannot
be constructed to accommodate it without losing type precision on code which is executed.

By constraining acceptable programs in this way, type systems give assurances about the
safety of the program: A type-checked program should, in general, encounter many fewer
run-time errors than an unchecked program with only the cost of static type checking.

2.5.1 Notation

When discussing the typing of a program, there is an important distinction between syntactic
elements – the keywords, expressions, keyboard characters that make up the program
– and the semantics of those elements, which may have the same name. At the most
basic level, there is a difference between the characters 12 and the number 12, which is
the usual meaning for that pair of characters, as well as 0x0C and 014 in C, for example.
Program syntax will, therefore, be typeset in fixed-pitch type in order to distinguish it
from semantic elements.

Type systems are usually expressed in rules such as \( A_1, A_2, \ldots \vdash B \), which expresses that statement
\( B \) holds if the premises \( A_1, A_2, \ldots \) also hold. \( B \) will often take the form of \( \Gamma \vdash \text{term} : T \),
expressing that \( \text{term} \) has type \( T \) given some kind of context or environment, \( \Gamma \).

2.5.2 Typing Functional Programs

A great deal of work has been done on type systems for functional languages, particularly
the \( \lambda \)-calculus – a minimal functional programming language. This work demonstrates how
even such a simple language can exhibit many of the issues underlying program typing;
issues which extend to the richest object-oriented languages.

\( \lambda \)-calculus terms, \( L \), are constructed from only three productions: \( L ::= x \mid (\lambda x.L) \mid (L_1 L_2) \),
where \( x \) is a variable – the second and third productions are known as abstraction and
application respectively. Execution is by the reduction of redexes, which are composed of
an application of an abstraction to some other term; for example \( (\lambda x.L_1)(L_2) \). This is
‘executed’ to give \( L_1[L_2/x] \), which represents \( L_1 \) with occurrences of \( x \) replaced by \( L_2 \).\footnote{Actually, only free occurrences of \( x \) are substituted in \( L_1 \). So, either free occurrences should be checked,
or, as is more usual, we consider every distinct bound variable modulo \( \alpha \)-conversion. In this whirlwind tour
of \( \lambda \)-calculus, there is no space to cover this kind of detail.}

Despite its extreme simplicity, all programs can be encoded in this way.
Inferring Ownership

Typing the $\lambda$-calculus

Typing of $\lambda$-expressions was first studied by Haskell B. Curry, who based his system on the observation that $\lambda$-abstractions of the form $\lambda x.L$ took a value of some type (as $x$), and then returned something of another type (as $L$ with some substitution for $x$). This was expressed by assigning such abstractions a type like $\sigma \rightarrow \tau$, where $\sigma$ and $\tau$ could themselves be more complicated types: A very large $\lambda$-expression with a similarly complicated type could be passed into the expression, but those types are unknown until it is possible to examine both $L$ and the expression accompanying the abstraction in a redex.

Where applications of the form $((\lambda x.L_1)L_2)$ were to be typed, it is intuitively necessary that the abstraction on the left should take an input of the same type as that given to $L_2$. So, if $(\lambda x.L_1)$ has type $\sigma \rightarrow \tau$, and $L_2$ has type $\rho$, then we can see that $\rho$ and $\sigma$ must actually represent the same type. This constraint on types is generalised to all applications of the form $(L_1L_2)$.

Properties of the type system

These constraints or rules are those which govern Curry’s type system for $\lambda$-calculus, one of the simplest type systems designed, yet which provided a basis for many type systems used with larger, real programming languages. Despite its simplicity, there are a number of interesting questions which must be asked, to which answers can be found in the literature:

- Can more than one valid type be given to the same $\lambda$-expression? If so, is there a ‘basic’ type which represents all the others (the principle type property)?
- Can every $\lambda$-expression be given some valid type?
- If a $\lambda$-expression is executed, as demonstrated above, will the result have the same type as the original expression? (The subject reduction property)
- If two $\lambda$-expressions have the same type, do they do the same thing? Are they the same expression?
- Given a $\lambda$-expression and a type, what is the complexity of an algorithm trying to check it for validity?
- Given just a $\lambda$-expression, is it possible to find an algorithm which will assign it a type? What is its complexity?

These questions all apply to the much richer languages which are used in practice, though they may be increasingly difficult to answer.
2.5.3 Typing Imperative Programs

The primary difference between the statements found in imperative programming languages and expressions from functional languages is the presence of side-effects. A statement itself may not produce any result at all, but in order to be useful, it will have some effect on the machine state.

Typing of imperative programming constructs has been investigated by encoding their properties in λ-calculus and investigating their properties in this familiar environment, as well as by building type systems on their syntax directly[20]. In some systems, statements are given the special empty type \( \text{void} \), familiar as the \( \text{void} \) declaration in C/C++. More interesting type rules only apply within the syntax of individual statements, such as where the language requires the left-hand side of an assignment to share a type with its right-hand side[2].

Functions can be typed based on their parameters: For example, the C function \( \text{int strcmp(char *s1, char *s2)} \) can be assigned the type \( \text{ptr(character)} \times \text{ptr(character)} \rightarrow \text{integer} \), indicating that it takes two character pointers and returns an integer. The typing rules for the language should specify that wherever this function is called, it must occur in a context where an integer is expected, and must be given two character pointers as arguments. The types of declared functions and variables are usually carried in the type system’s environment (see section 2.5.1) in order to enable these checks to take place.

2.5.4 Type Systems for Object-Oriented Languages

Further extending type systems to object-oriented languages adds yet new challenges. The programmer has almost complete control over the types available in the programming language, through their definition of new classes. Additionally, subclass polymorphism allows objects of one class to be used in the place of other class where they are related by inheritance. Combined with this, a type system designed to give the programmer some assurance of safety should check that no object can be sent a message for which it does not implement a method, avoiding SMALLTALK’s “message not understood” error.

Palsberg and Schwartzbach presented a general object-oriented type system where types in an object-oriented programming language were simply sets of classes[20]. In this way, polymorphism can be immediately represented by ensuring that any type containing a class also contains all of its subclasses. They express their type system directly in terms of constraints, for example given an expression \( \text{exp} \), the type of that expression, \( [\text{exp}] \), was specified by set relations on \( \text{exp} \)'s components. For example, for the expression \( v := \text{exp}_2 \), they specify the requirement that \( [\text{exp}_2] \subseteq [v] \). This insists that the right-hand side of an assignment cannot have a larger domain than the item it is being assigned.

Of particular interest is the constraint which arises from method calls: Here, expressions of the form \( \text{exp}.m(\text{exp}_1, \text{exp}_2, \ldots, \text{exp}_n) \) give rise to a number of constraints on the parameters,
Inferring Ownership

```plaintext
// a, b, c have no declared type
a = new A
b = new B
c = new C
```

Figure 2.5: More than one possible C++ type for variables b and c exist, given the declared class structure on the left. As with all programs with more than one class, there is more than one sets-of-classes type for all the variables in the program.

but also a constraint specifying that \([\text{expr}] \subseteq \{ x \mid \text{m is a method of } x \}\). Thus, Palsberg and Schwartzbach explicitly preclude the possibility that a message receiver does not implement an appropriate method.

This ‘types as sets of classes’ system is a generalisation of the type systems in use for languages like JAVA and C++. In C++, all types which are not class pointer types can be expressed in Palsberg-Schwartzbach types as a set containing just themselves. Where a class pointer type is used, the equivalent Palsberg-Schwartzbach type is the set containing pointers to that class as well as to all of its subclasses. Similarly in JAVA, primitive types are generalised to sets containing themselves, and class references types are sets implicitly expanded with the class type’s subclasses.

Palsberg and Schwartzbach also examine monotone inheritance, which specifies that subclasses should be extensions of their superclass. This prohibits the activity in SMALLTALK where methods can be ‘cancelled’ after inheritance from a super-object. Given the implicit expansion from JAVA or C++ types to sets-of-classes types, as well as their use of monotone inheritance, it becomes obvious that objects assigned to class variables will answer any given message as long as the class in the declared type is able to do so.

2.5.5 Type Checking vs Type Inference

Type checking has already been introduced as a method for checking the validity of a program already annotated with types. Type checking is normally done by matching the rules of a type system with the specific program and types given, or by finding a solution to the constraints imposed in the examples from the previous section.

Type inference is a process by which an unannotated or partially annotated program is typed. While an algorithm for type checking can usually be derived directly from the type rules, more ingenuity is often required for a type inference algorithm in order to avoid an exhaustive search through all the potential type derivations for the program. Certainly inference tends to have a greater complexity than checking, and is undecidable in many circumstances.

Additionally, more than one typing can often be obtained for the same program, demon-
...strated in figure 2.5. With sets-of-classes typing, the smallest types assignable are \{A\} for a, \{B\} for b and \{C\} for c. However, these types can be expanded with any other class in the program without violating the constraint on assignments that the type of the right-hand side must be a subset of the left. However, even with JAVA types, b could be assigned the type B or the more general type A, keeping the assignment legal. Similarly with c, though variable a can only be assigned the type A.

One of the challenges when constructing an inference system is, therefore, to determine whether there is a ‘best’ type and then to design an algorithm that finds it. The ‘best’ type might be the most general, the most restrictive, the smallest, the one requiring least time to find or may be measured by some other property. In terms of Curry’s system from section 2.5.2, the principle type was that from which all other valid types could be obtained by substitution\(^8\). In our example above, a principle type might be given by the most specific assignment of types, based on the class name given to the new operator. Other types can then be obtained by substituting class names with those further up the inheritance tree – this does not work in general, of course, as method calls and field names start to disappear!

There are also type inference systems where the optimal type – by whatever metric – may not be inferred automatically. Instead, the typing resulting from inference may be an approximation of the optimal type, which may otherwise be obtained only through manual annotation. Palsberg and Schwartzbach point out that their inference algorithm can only approximate the optimal type, which may itself even fail to satisfy the constraints generated on the types during inference\(^20\).

### 2.6 Summary

In this chapter we showed how object aliasing hinders effective reasoning about object-oriented programs in a number of situations. We examined a variety of methods used to alleviate the burden of aliasing to improve the effectiveness of program analysis in the presence of aliasing.

A brief overview of type systems for functional, imperative and object-oriented languages was given, in which we described how sound type systems are a necessarily restrictive constraint on the possible operations of a program. We summarised some of the classic issues for type checking and inference on functional programs, and we demonstrated how some of these reflect on object-oriented type systems.

In the following chapter, we give much more detailed background on the specifics of the ownership type systems upon which this work is based.

---

\(^{8}\)For example \(\sigma \rightarrow \tau\) is the principle type of a single \(\lambda\)-abstraction. Types such as \(a \rightarrow b, (a \rightarrow b) \rightarrow c\) and \((a \rightarrow b) \rightarrow (c \rightarrow d)\) can be obtained by substituting larger elements for \(\sigma\) and \(\tau\).
Chapter 3

Ownership Types

This chapter gives an introduction to the primary motivation behind this project – ownership types. We discuss the properties of ownership, how the static type system relates to run-time properties, and give some justification for how the invariants are maintained.

3.1 Object Ownership

The focus of this project is on ownership types, a system which draws together the ideas of aliasing in object oriented languages and their type systems. Ownership types are meant to reflect the programmer’s intuition about the structure of aggregate data structures, and how their constituent objects refer to each other. Through the type system, ownership types impose statically-checkable restrictions on aliasing, encouraging true enforced encapsulation of objects within aggregate structures[6]. They are, therefore, much more powerful than traditional alias annotations which only represent programmer beliefs without any kind of checking.

3.1.1 Object Heaps

Traditionally, we regard the state of an object-oriented program as a graph of nodes, which represent objects, and edges, which represent references between them. Such an object graph is shown in Figure 3.1, which illustrates part of a program heap showing the objects used to represent a list of values. While the object of the LinkedList class, list, is the object which manages the list, the representation of the list comprises all the displayed objects, which includes those of the Link and Value classes.
3.1.2 Ownership Diagrams

Suppose that an object external to list were to reference one of the Link objects. Then that object would be able to modify the list without the knowledge of the managing object, list – the representation of the list is not truly encapsulated from outside interference. The idea of an object owner protects aggregate objects from being modified without knowledge of their managing object. Each object is assigned an owner, such that all references to owned objects must pass through their owner.

Figure 3.2 serves to illustrate how access to owned objects – in this case the Link and Value objects – must be controlled by their owner. It also illustrates how ownership is commonly represented in the literature. Each object has an ownership boundary, shown in the diagram with a dotted line, through which references cannot pass. The reference from the ext object is not permitted, as ext is outside the ownership boundary of list, while object 3 is inside the boundary. The area inside the diagram is the object’s context or representation, the implication being that objects inside another object’s ownership boundary make up an aggregate structure managed by the owning object. In the figure above, the Link and Value objects make up the representation of a LinkedList object. Each object sits on the box representing its ownership boundary: It is both inside and outside its own box, as external objects can reference it and it can reference objects within its own box.

Ownership diagrams provide an ideal means by which to reason about specific examples of ownership, though the usual caveat applies about making general assumptions based on the examination of single, specific examples.
Inferring Ownership

Figure 3.2: The same linked list object, but where list is the owner of all the other objects. Therefore, all references to those objects must pass through their owning list object. This requirement creates an ownership boundary (shown as a dotted line) around those objects, with the owning object – list – as the gatekeeper. The reference from the external object, ext, from the outside of this boundary to an object inside is disallowed. If ext wishes to access object 3, it must do so indirectly via a reference to list, which may then provide access control for the list’s representation.

3.2 Ownership Contexts

Let us generalise the picture away from the linked-list program heap of Figure 3.1 to the object graph shown in Figure 3.3, a graph which will form many of the examples in this report. The ownership boundaries are marked with dashed lines, and mark out each object’s ownership context. The context of A contains two objects directly, B and F. In turn, B and F have their own contexts, and so on. We say that the context of B is inside that of A or, in mathematical notation, B ≺ A.

The ≺ operator is reflexive and transitive by definition, so E ≺ B and H ≺ H. It should also be clear that it is anti-symmetric, that is o₁ ≺ o₂ ∧ o₂ ≺ o₁ ⇒ o₁ = o₂. For a full definition of ≺, see the type-system of the Jœ₀ language in Appendix A.

There is also a global context, usually called world (though it has been written norep, for ‘no representation’ and root in other literature), such that every object address is contained by world. Therefore, A ≺ world. Alternatively, of course, we could regard A’s context as the global context.

These contexts may also be regarded as a tree of objects, where each object’s parent is its owner. Therefore owner(B) = A and owner(A) = world. If we regard A as the root or global object, then we refine this slightly so that owner(A) = A.
3.3 The Ownership Invariant

The ownership invariant, given by Clarke in his thesis on ownership types[5], specifies that an object, \( o_1 \), referencing another, \( o_2 \), must always be inside the owner of \( o_2 \):

\[
o_1 \text{ references } o_2 \Rightarrow o_1 \prec \text{owner}(o_2)
\]

The effect of this invariant is to disallow references from outside of an object’s ownership context to the inside. Figure 3.4 shows some object references which are disallowed given the ownership boundaries shown. The reference from \( D \) to \( E \), for example, is not permitted as \( E \)'s owner, \( C \), is not inside \( D \): \( D \not\prec \text{owner}(E) \).

Figure 3.5 gives more examples of when the ownership invariant allows references to occur. An object can always reference other objects within the same context, as in Figure 3.5(a), because they have a common owner. References from an object to its owner are also always permitted, Figure 3.5(b). This idea may be extended so that objects can always reference to objects indirectly outside their own, as in Figure 3.5(d). Finally, as in Figure 3.5(c), an owner may always reference objects within its own context but cannot reference inside those objects’ contexts.

3.3.1 Owners as Dominators

It emerges that object owners are always dominators in the object graphs representing program heaps[5, 7, 6]. An object, \( o_1 \), dominates another, \( o_2 \), whenever \( o_1 \) is on every path from the graph’s root (so, in Figure 3.4, the root is \( A \)) to \( o_2 \). Recall Figure 3.4, and notice...
Figure 3.4: References may pass outwards through object boundaries, for example between G and A, but may never pass inwards through an object boundary. Some references that would be disallowed are shown on the diagram, for example from D to E. References are allowed if and only if the referencing object is inside the owner of the referenced object. So, the reference from G to A is allowed as G is inside the owner of A, which is the global context.

that, for example, every path to E from A goes via C. The disallowed reference from D to E would have by-passed C by creating a new path that would not have gone via C. It would have broken the ownership invariant imposed by C’s ownership of E.

This property is known as the Owners-as-Dominators property, and is one of the fundamental properties underlying ownership types. Note, however, that ownership is an artificial relationship between objects, and domination is a property of object graphs – it is the responsibility of the type system to ensure that object owners are always dominators throughout the execution of a program.

3.4 Using Ownership Types

The owners-as-dominators property is a run-time property of object heaps, and does not represent any kind of type system. Conversely, the ownership type system specifies how ownership is bestowed upon objects during program execution, but does not specify individual object relationships.

The ownership type system annotates class declarations with zero or more ownership parameters, e.g. class A<o, p1, p2, ...>. The first ownership parameter, if present, is the owner of that object; where it is absent, the object has the special owner world, indicating that it is owned only by the system. Within classes, class reference types are annotated with values from the parameters of the surrounding class, or with world, this or owner. world has already been introduced: A may have been created with world as its owner in
Figure 3.5: Various allowable referencing situations. In every case, whenever \( o_1 \) references \( o_2 \), \( o_1 \) is inside the owner of \( o_2 \): \( o_1 \prec \text{owner}(o_2) \).
class X<owner> {
    Y<this, owner> fxy;

    void mX() {
        fxy = new Y<this, owner> ;
        fxy.mY();
    }
}

class Y<owner, p1> {
    X<this> fyx;
    Z<p1> fyz;

    void mY() {
        fyx = new X<this> ;
        fyz = new Z<p1> ;
    }
}

class Z<owner> { ... }

Figure 3.6: This figure illustrates how some simple code instantiates objects with ownership (though, of course, the ownership is just a property of the object heap – it is not actually stored in any way at run-time). The field fxy holds a Y object with an owner of this – therefore, any instance of Y assigned to fxy is owned by the instance of X holding that field. Y is also passed a parameter which, in X, is assigned the X-instance’s owner. Y uses the value of this parameter to assign the ownership of a new Z-instance, stored in fyz, which is placed in the same context as x1. Y also holds another X-instance, to which it gives the this parameter and which it therefore owns.
Figure 3.4. An owner of this indicates that the new object is owned by the object creating or referencing that object. owner indicates that the new object’s owner is the same as that of the creating or referencing object – this creates ‘siblings’ in the ownership tree.

The use of these ownership types is demonstrated in Figure 3.6, along with a sample of the ownership it enforces. X takes a single parameter, its owner, and has a single field, fxy. Y takes two parameters according to its class definition, and X passes this, which represents the instantiating X-instance’s own context, and owner, the context that encloses it. The Y-instance, y1, is therefore allocated inside x1’s context when x1.mX() is called. When y1.mY() is called, it allocates new X object inside its own context, by passing this as its ownership argument, and allocates Z instance to the context it was passed as a parameter – specifically, the context enclosing x1.

3.5 Context Substitution

The ownership parameters form paths through with object contexts are propagated through the program. This propagation is performed both at run-time, in the operation semantics, and statically, when type-checking an ownership-typed program.

3.5.1 Run-time Context Substitution: \( \tau \)

Recall Figure 3.6, when the x1.mX() call was made. At this point, the newly-created instance of Y, y1, was passed two parameters: its owner, x1, and a second parameter, owner(x1). It then used these parameters to instantiate objects into those contexts.

Therefore, we can define a run-time substitution, \( \tau_{y1} \), which maps the formal parameters of Y to the ownership contexts they should have in the instance y1.

\[
\begin{align*}
\text{owner} & \mapsto \text{owner}(x1) \\
\text{p1} & \mapsto x1
\end{align*}
\]

Similarly, there is a substitution, \( \tau_{x2} \), which maps the parameters of class X to ownership contexts when we instantiate it to produce object x2:

\[
\begin{align*}
\text{owner} & \mapsto \text{owner}_x1
\end{align*}
\]

Notice that the context owner(x1) only ‘reaches’ x2 by virtue of the two substitutions above – \( \tau_{y1} \) instantiates y1’s parameter, p1, such that it can then be used to instantiate x2’s ownership parameter, owner.
3.5.2 Static Context Substitution: $\sigma$

Concurrently, there is a static substitution required when examining how contexts flow in the static program text. Again, examine the program text from Figure 3.6. There is a substitution which links the formal parameters of $X$ to those of $Y$ at the instantiation in $X.mX()$. Here the substitution, $\sigma_{(X,Y)}$, is:

$$
\begin{align*}
\text{owner}_Y & \mapsto \text{this}_X \\
\text{p1}_Y & \mapsto \text{owner}_X
\end{align*}
$$

The parameters have been annotated with their enclosing class in subscript to distinguish between different instances. There is a similar substitution, $\sigma_{(Z,X)}$, for the instantiation of $X$ in $Y.mY()$:

$$
\begin{align*}
\text{owner}_X & \mapsto \text{p1}_Y
\end{align*}
$$

The composition of these substitutions, $\sigma_{(Z,X)} \circ \sigma_{(X,Y)}$, specifies a ‘channel’ through which the context of $X$’s owner is passed into $Y$ for use in $Z$.

3.6 Enforcing Owners-as-Dominators

In this section, some formalism is necessary to demonstrate how the dynamic property of domination and ownership can be enforced statically. This is accomplished through the rules of the type system; a less formal subset of the rules from $\mathcal{J}\text{o}e_0[6]$ is presented here.

3.6.1 Type Rule Environments

When statically checking an ownership type system, an environment, $E$, is required to give types to term variables. This environment is used in the following situations in $\mathcal{J}\text{o}e_0$, a language derived from that used in earlier work by Clarke and Drossopoulou[6] (see Appendix A):

1. The use of $\mathcal{J}\text{o}e_0$’s single-assignment let construct, where the variable it binds is assigned the type of the assigned data.

2. Type checking of methods, where variables used as formal parameters are bound to their declared type.

3. When checking class definitions, the environment has constraints added for context parameters in order to enforce the owners-as-dominators property. These are discussed in section 3.6.2, below. Additionally, this is given a type corresponding to the class that is being checked.
A set of bindings, $B$, is also present in the type rules of [6] but it is inert in the type rules, required only for the formal type preservation proof. This report instead presents an informal argument for preservation of the ownership property, so the bindings are omitted.

The judgements of the type system are, therefore, of the form $E \vdash s : t$, stating that some syntax, $s$, has type $t$ given the environment $E$.

### 3.6.2 Valid Owners

An ownership type is valid if the owner parameter is somewhere inside all other parameters in the type. When explaining the necessity of this property, it must be assumed that all ownership parameters will be used as the owner of some object in the aggregate in question – if this is not the case, then the parameter is superfluous and can be deleted. This property prevents the exportation of ownership outside the aggregate that is being declared by prohibiting the flow of context outside the class that generates that context.

Formally, for the type $c\langle p_1, p_2, \ldots, p_n \rangle$, the requirement $\forall i : 2..n. [p_1 \prec p_i]$ must hold.

With this property, the full combination of legal ownership is allowed without the use of class variables as context values, simply by annotating every class’ definition with a parameter which takes this from the class that created it.

### 3.6.3 The Update Rule

Clearly, flow of references through the system is responsible for the potential violation of the owners-as-dominators property, as references may be created which bypass owners on paths from the heap root. Therefore, field update\(^1\) must be at the heart of the static ownership check.

The rule, simplified from Clarke[6], is:

\[
\frac{(1)E \vdash z : c(\sigma) \quad (2)F_c(f) = t_{fld} \quad (3)E \vdash y : \sigma_z(t_{fld})}{(4)E \vdash z.f = y : \sigma_z(t_{fld})}
\]

Each numbered part of the rule expresses the following:

1. For code such as $z.f_{1d} := y$, assume that the type of $z$ is known to be $c_{z< v_1^z, v_2^z, \ldots, v_n^z >}$, where $v_1^z \ldots v_n^z$ are context values replacing parameters in $z$’s type – this replacement was accomplished by some substitution, $\sigma$.

\(^1\)Method-local variables are ignored here, but the idea is simpler in the language of [6] as method-locals are single-assignment.
Inferring Ownership

2. The type of field `fld` in class `cz` must be retrieved from the program text with the function $F_c$, which will be of type $t_{fld}$.

3. By applying the substitution $\sigma_z$ to $t_{fld}$, we instantiate all of $t_{fld}$’s context parameters with values, including `this` which we instantiate with $cz$’s context value: `this` must be substituted in this way as the type $t_{fld}$ was declared inside the body of $cz$, but the type for $y$ that is to be constructed must be valid outside that class body.

4. This newly instantiated type, represented by $\sigma_z(t_{fld})$ is the type that we require $y$ to possess in order that the ownership properties are preserved.

In summary, the above definition ensures that the ownership of the assigned data must match the ownership type of the class field. Combined with the valid owners property from the previous section, which disallows field names to be declared with undesirable ownerships, this intuitively demonstrates the preservation of ownership as references flow around the system. The full proof is, of course, a great deal more involved and is fully explained in earlier literature[6].

3.7 Summary

Ownership types provide a system for assigning owners to individual objects using a statically checkable type system. We discussed how owners suggest ownership boundaries in the graph, through which references cannot pass, and saw how owners defined in code are always dominators whenever the system is ownership type-correct.

The two species of ownership parameter substitutions – dynamic and static – were introduced, and we saw how these are vital when considering how ownership contexts flow around a running program, and for explaining how the type system functions. We also demonstrated how the ownership invariant is maintained according to a rule from the type system.

For those familiar with type systems and semantics, Appendix A provides the full operational semantics and type system of a simplified target language with ownership, $Joe_0$, as well as a formal description of how dynamic program state may conform with the ownership type system.
Chapter 4

Motivation and Plan

In this chapter, we discuss the reasons for inferring ownership types and why such a system is desirable. We discuss how such an inference might proceed, and provide justification for the approach presented in this report.

4.1 Motivating Ownership Inference

We have already discussed how object aliasing significantly hinders program reasoning, optimisations, debugging and maintenance. Ownership types provide some relief from these problems, but there has not yet been an attempt to infer ownership types for program code without annotations. In this section, we discuss how such a system may be useful:

Legacy code There are large bodies of code in use which have no ownership annotations, so while it may be desirable to exploit ownership typing to improve optimisation or to highlight unexpected effects, the need to modify old code may outweigh the benefits. Where ownership typed code may be used in a system without full annotation, the conservative assumption must be made that all unannotated classes generate world-owned objects. However, with an ownership inference system, we might hope that more informative ownerships can be determined for unannotated objects. These inferred ownerships could even be fed back into the code so that the system maintainers might benefit from the additional information.

Moreover, there is potential for ownership types to be inferred for entirely untyped code so that future additions can be consistently typed with legacy code.

Optimization and Verification A Pointer Alias Analysis is often performed – to some extent – by many program verification tools and high-performance optimising compilers. These analyses are often expensive, but must be performed every time the
Inferring Ownership

program is re-verified or compiled. Ownership types provide a mechanism for annotating programs with the results of such an analysis. However, as a static type system, it is only able to represent context- and flow-insensitive pointer aliasing information. Once ownership information has been added, verifying the properties remain consistent with the type system is a matter of a static check[7].

The annotations may then be used to perform optimisations by virtue of the reduced cost of program reasoning with ownership types[11]. Additionally, ownership types have been applied to the problems of data locks[3] in a similar way, though with different emphasis, as pointer alias analysis[23].

Program Comprehension Ownership has been shown to be useful when visualising object-oriented programs[14], and the inference of ownership types offers a static equivalent to well-organised run-time visualisation. The use of ownership types may allow more efficient comprehension of program text, by highlighting how different areas of code may access and modify each other’s invariants.

Garbage Collection If ownership is known at run-time, either from ownership annotations or on-line calculation of ownership in the object heap, then a garbage collector can be greatly optimized[4]. Whenever an object becomes unreferenced, then all the objects that it owns also become immediately disposable. Normally, the garbage collector will have to examine other objects individually if they were referenced by the disposed object, but a much greater return (in terms of disposed objects) can be obtained per garbage collection cycle if ownership is known.

Clearly, the correctness of the garbage collection is vital, which further underlines the importants of a formal treatment of run-time ownership information.

4.2 Motivating the Formal System

Many of the benefits of ownership types are predicated on their ability to make guarantees about program behaviour, the absence of aliasing, and the correctness of the ownership invariants upon which they are based. Therefore, in any representation of the type system and its properties, we must be sure that the representation is faithful and that the properties are rigorously investigated. Formal systems provide the machinery necessary to reason about all the possible conditions and cases without exhaustive investigation, which would be impossible in the general case.

Therefore, this report presents detailed definitions and, where necessary, proofs that the expected properties hold to show that our representation is faithful or sound with respect to our intuition. With this foundation, we show how properties emerge from this system that will later prove useful for algorithms or additional formal work.

Without this formal work, there could be no guarantees of the system’s correctness and, therefore, the correctness of decisions based upon it.
class V<owner> {
    W<this, this> fV;
}

class W<owner, pW> {
    X<this, pW> fW;
}

class X<owner, pX> {
    Y<this> fXY;
    Z<this, pX> fXZ;
}

class Z<owner, pZ> {
    R<pZ> fZ;
}

Figure 4.1: The Z object references V’s ownership context as it references an object contained by that context. In order to do this, the field of Z which holds the reference must be annotated with a context parameter which, at run-time, holds V’s context. Thus, there must be a chain of parameters (represented with dashed lines) passing V’s context down to Z through the type system. The program text gives an example for how such a program might be typed.

4.3 Dynamic Inference

Traditional type inference systems have focused on inferring types from the program text alone. However, there are a number factors which make such a strategy particularly difficult for ownership types; most notably the issue of ownership parameters. As discussed in Section 3.5, ownership parameters provide access for classes to contexts it cannot directly reference with owner, this or world. When considered in the context of the entire object system, a chain of ownership parameters is required from every referenced context to the object which references it – an example is illustrated by Figure 4.1. Thus, the complexity of type inference for ownership types seems much greater than for data type systems, where there is far less coupling between typing decisions made for different portions of code.

This statement requires justification: Suppose one were to try typing the return value of a statement “if b then e₁ else e₂”, where b, e₁ and e₂ are expressions. The resulting type is clearly some combination of the types of e₁ and e₂, respectively[20]. We observe, therefore, that typing information tends to flow outwards from smaller program portions to larger portions.

On the other hand, Figure 4.1 demonstrates how there is a flow of typing information in both directions: At a local level, we may find extra context parameters are required which
must ‘ripple’ outwards through class declarations and the inheritance tree. In turn, these changes will require adjustment of local typing decisions made elsewhere, and so on. It is not even clear how to determine the number of ownership parameters required!

In this project, therefore, we investigate the use of object graphs to reason about ownership, in the hope that it provides a means to investigate the problems associated with a static inference, to improve our understanding and to provide a solid foundation for future work based on ownership. Furthermore, ownership from object graphs is, in itself, an important topic which requires careful formulation.

### 4.4 Inference Strategy

#### 4.4.1 Overview

Figure 4.2 shows a block diagram representing the various processes we imagine might be required for the inference of ownership types, given the use of object graphs. The idea behind the overall strategy is to place successively more stringent constraints upon the potential ownerships as the data derived from program execution passes through the system.

#### 4.4.2 Obtaining Object Heaps

Given program text, we propose that the code be run for some time (many execution steps) through some abstract machine, which records the program heaps in every program state. These will provide a means by which to examine references that occur during program execution and, perhaps more importantly, those which do not tend to occur. Of course, any finite set of program runs will not, in general, provide a complete characterisation of its behaviour.

We propose the use of some kind of virtual machine to extract this information during program execution – the information required is, as we shall see, more detailed than is maintained by a standard run-time environment. Formally, some trivial extensions to the operational semantics will be required, in order to hold multiple ‘snapshots’ of the running program’s heap.

The details of this kind of collection are relatively uninteresting and provide more of an implementation challenge than a concern about correctness. They are therefore omitted from this discussion.
Figure 4.2: This figure demonstrates how we intend to use ownership inference to arrive at suggestions for ownership typed program text. See Section 4.4 for a detailed account of the overall strategy for the inference.
4.4.3 Extended Object Graphs

The Extended Object Graphs are used to store the information from several distinct program heaps in a single structure. This is important if the contents of all the heaps are to be considered at once, in order to discover properties invariant with program execution. Chapter 5 introduces the formal system with a discussion of the representation used for the object heap data, and of how to produce such a representation from a set of distinct program heaps.

4.4.4 Domination

The owners-as-dominators property is key to ownership types (see Section 3.3.1), so domination would appear to be an appropriate first step on the way to examining the inference of run-time ownerships. Therefore, from the extended object graphs, we derive domination information that is invariant with program execution.

In Chapter 6, therefore, we introduce a system for reasoning about domination, and show that it follows our intuitive definitions of domination. We also see that domination is not sufficient to entirely characterise ownership because of various "dynamic constraints".

4.4.5 Ownership

From the domination characteristics previously inferred, dynamic ownerships must be constructed. Chapter 7 discusses the means by which to derive ownerships reliably from domination information and discusses how to find the best of a large number of potential ownerships.

When discussing potential ownerships, precision is vital if we are to find useful ownerships. An ownership where all objects are owned by the world global context is not informative, but is always valid. However, we must hope to do considerably better than this when finding invariant ownerships if we hope to use the ownerships to improve program reasoning. Therefore, a good portion of Chapter 7 is devoted to work based on the precision of ownership.

While discussing precision, it is important to remember that the invariants under consideration can be cheaply and statically checked – this is not a situation such as is found in program analysis where overly conservative assumptions must be made for safety. Therefore, we consider precision relative to a set of run-time references between objects; we leave, until the following chapter, observations about overly restrictive precision with respect to the static type system.
4.4.6 Ownership-Type Consistent Ownerships

This part of the inference process restricts the set of possible ownerships to those which are appropriate for consideration as static ownerships.

As with virtually every static type system, the constraints it imposes on the run-time state of a program are usually overly conservative. Therefore, when going from run-time state to static types, some dynamic information may be inappropriate with respect to the type system. These “static constraints” must be used, therefore, to select possible ownerships based on their suitability as the result of an ownership-typed program.

When an ownership is consistent with a type system, we say it is ownership-type consistent (OT-consistent). Again, the goal here is to restrict the number of potential ownerships that move to the next phase of the type inference.

It is in this portion that extra information is required, which is not normally stored in object heaps. In general, ownership and creator information is vital when inferring ownership types particularly for the this and owner ownership values, which can only be considered relative to the context of an object’s creator. Additionally, such creator information can impose constraints on ownerships that are valid in the type system: For example, ownership parameters cannot change values during run-time. Therefore, objects allocated by the same new statement and by the same object must have precisely the same ownership characteristics. However, as run-time ownership is considered for each individual object, this requirement forms an additional constraint on the inferred ownerships.

4.4.7 Ownership Parameter Substitutions

As a first move towards the annotated program text, the substitutions described in Section 3.5 must be inferred. From the domain of these substitutions, then, we obtain the number of parameters required in each class definition and in each reference to that class name throughout the program. From the substitutions themselves, we are also able to determine how the parameters of an enclosing class are used in each type declaration within that class, particularly for the class’s field variables. We give suggestions for how this inference may proceed in Chapter 9.

4.4.8 Type Propagation and Checking

From the parameter substitutions and the field typings that are implied, it should then be possible to type the formal parameters of methods and the type declarations used for new statements. Therefore, field types should be propagated along the data flow paths in the program to ensure such expressions are well-typed. For example, when a statement \( f := \text{new } A \) allocates to a field of type \( A\langle p_1, p_2 \rangle \), then the statement can clearly be annotated with the type \( \text{new } A\langle p_1, p_2 \rangle \) as, according to the type system, the left and right hand sides...
Inferring Ownership

of the assignment must have precisely equal ownership parameters.

4.5 Summary

In this chapter, we discussed how ownership type inference is a valuable tool if the benefits of ownership types are to be fully exploited. We also justified the use of a dynamic inference – relying on run-time sampling to obtain data for the inference – as a means for exploring a static inference, as well as for the other potential uses of ownership inference on object graphs. A number of problems were also identified that may impede static inference.

We gave an overview of our strategy for the inference, including the idea that the system successively restricts the number of potential ownerships that may be considered acceptable. There may be a number of potentially valid typings, thus it may be the job of the user or of other constraints to finally decide on which is the most appropriate.

In the following chapters, we give in complete detail a study of the inference of precise ownership information from object graphs, which may be used both to infer ownership types and for a variety of other applications such as visualisation, debugging and invariant detection. From there, we make a number of suggestions and assertions about the complete inference process, which represents the full breadth of further research possible based on this work.
Chapter 5

Introduction to the Formal System

This chapter introduces the fundamental definitions for the discussion of dominators and ownership in the following chapters. Extended Object Graphs are presented as extensions of the classical object heaps found in object-oriented semantics literature, references and paths are discussed, and we give a function for combining object graphs so that the properties of several object graphs may be considered simultaneously. The chapter concludes with some notes on the design of these structures.

5.1 Notation

A number of typographical conventions are used to indicate the kind of structures represented by different names. Variables are simply presented in italics, sometimes annotated with distinguishing marks \((a, a_1, p', O)\). Named sets, relations and functions are set in sans-serif \((\text{Address, ref})\). Binary relations are normally written as infix operators. As new sets are introduced, one or more variable names that range over that set will be specified – subscripts and other marks distinguish between variables ranging independently over the same set.

Normally sets are defined by equality or as a product of other sets. However, where a set, \(M\), is a mapping, it shall be defined by \(M \overset{\text{def}}{=} A \mapsto B\). This is equivalent to \(M \subset P(A \times B)\), where \(M\) contains only those sets \(m\) for which \((a, b_1) \in m\) \(\land\) \((a, b_2) \in m \Rightarrow b_1 = b_2\) holds.

Members of \(M\) can be used in functional notation such that \(m(a) = b\) where \((a, b) \in m\), or \(\text{Udef}\) – the undefined value – otherwise. Also, the domain of \(m\) is given by \(\text{dom}(m \in M) = \{ a \mid (a, b) \in m \}\). Notice that \(\text{dom}\) is written in italics, and should not be confused with the binary relation \(\text{dom}\), defined later in the discussion of the formal system.

The familiar turnstyle operator, \(\vdash\), is used to indicate contexts for statements: \(G \vdash A\) should be read as “\(A\) is true in \(G\)”, or “\(G\) shows that \(A\)”, where \(A\) is not a logical formula.
Inferring Ownership

$G \vdash A \Rightarrow G \vdash B$ is not, therefore, a rule of inference, but instead relates the property $B$ to property $A$: The $\vdash$ binds more tightly to $G$ and $A$ than $\Rightarrow$.

The $\Diamond$ symbol is used to indicate that the context of the assertion, $G$, is well-formed in some way. For example, $G \vdash \Diamond$. Different types of good form are distinguished by their subscript.

Finally, the system contains a number of inference rules. The rules are specified using the usual notation for derivation systems as follows:

\[
\text{(Rule-Example)} \quad \frac{\text{requirement}}{\text{consequence}}
\]

Rules are always referenced in small, capitalised type (Rule-Example). There may be many rules with the same consequence portion but with different requirement portions, $R_1, R_2, \ldots, R_n$. Together these are equivalent to $R_1 \land R_2 \land \ldots \land R_n \iff \text{consequence}$. In other words, derivations according to inference rules grow upwards.

Free variables in logical statements are, by convention, universally quantified ($\forall$). Existential quantification ($\exists$) is always explicit.

5.2 Object Graphs

Object graphs are usually used to represent a snapshot in a program’s execution, where nodes are objects in the program heap, and where edges represent the references stored in object fields. These graphs are therefore limited to consideration of run-time characteristics of the object graphs. If the graphs are to be used as a means for reasoning about the run-time incarnation of static properties – in this case, run-time invariants – it must be possible to express all the potential links that may be present in the program heap at any time in program execution.

Perhaps the most important structure in the formal system is the Extended Object Graph – throughout the rest of the report, references to ‘object graph’ are to extended object graphs; object graphs from a single snapshot of program execution are known simply as ‘heaps’. We now introduce the formal definition of Extended Object Graphs, contained in the set $X\text{ObjectGraph}$.
Definition 1

\[
\begin{align*}
\text{ProgPoint} & \overset{\text{def}}{=} \mathbb{N} \\
\text{Address} & \overset{\text{def}}{=} \mathbb{N} \\
\text{Address}^N & \overset{\text{def}}{=} \text{Address} \cup \{\text{nil}\} \\
\text{XObject} & \overset{\text{def}}{=} \text{ClassName} \times \text{XFieldMap} \times \text{ProgPoint} \times \text{Address}^N \\
\text{XFieldMap} & \overset{\text{def}}{=} \text{FieldName} \mapsto \mathcal{P}(\text{Address}) \\
\text{XObjectMap} & \overset{\text{def}}{=} \text{Address} \mapsto \text{XObject} \\
\text{XObjectGraph} & \overset{\text{def}}{=} \text{XObjectMap} \times \text{Address}
\end{align*}
\]

Rather than objects mapping field names to values, field names are associated with a set of object addresses. For an extended field map, \( F \in \text{XFieldMap} \), \( F(f_1) = \{1, 2, 4\} \) indicates that the field \( f_1 \) has held the values 1, 2 and 4 during a period of some program’s run. Notice that there is no indication of the sample period, nor is there any information about when the value of \( f_1 \) was 1, or when it was 4. It will later be shown that such information is unhelpful for reasoning about ownership.

While the difference between ‘traditional’ object graph representations and this extended definition is restricted to the definition of \( \text{XFieldMap} \), the definitions depending on this are also annotated with \( X \) to distinguish them from their traditional counterparts, such as might be used in an object-oriented semantics: \( \text{XObject}, \text{XObjectMap}, \text{XObjectGraph} \).

The following variables range over the structures defined in Definition 1:

\[
p_n \in \text{ProgPoint}, a \in \text{Address}^N, o \in \text{XObject} \\
F \in \text{XFieldMap}, f \in \text{FieldName}, H \in \text{XObjectMap}, G \in \text{XObjectGraph}
\]

Notice that extended object graphs, \( G = \langle H, a_r \rangle \), have a heap-like map of addresses to extended objects, \( H \), and a distinguished address, \( a_r \), which is the root of the extended object graph. For brevity, where \( G = \langle H, a_r \rangle \), \( G(a) = H(a) \) for \( a \in \text{Address} \). In a running system, this root address \( a_r \) may be the initial object instantiated by the language’s runtime environment, or some other address chosen to represent the root of the object graph. It corresponds to the world ownership context from the static definition of ownership types.

5.2.1 Auxilliary Information in \text{XObject}

It was noted in Section 4.4.6 that some extra information, not normally stored in object heaps, is required for ownership type inference. Therefore, individual objects, \( o = (c, F, p_n, a_c) \in \text{XObject} \) contain extra information about the object’s creation, alongside the standard information about the object’s class and field values. The object’s allocation
Figure 5.1: These are example extended object graphs which will illustrate many of this work’s features. References are marked between each object (→). Formally, for example, one might say \( G_A \ldots \text{ref} \ldots \). \( a \) is the root object of both \( G_A \) and \( G_B \).

point, \( p_n \), indicates the \texttt{new} statement in the program text by which the object was instantiated. The object’s creator address, \( a_c \), specifies which object in the graph executed the \texttt{new} statement indicated by \( p_n \).

Notice that \( a_c \) ranges over \( \text{Address}^N \) as it can take a \texttt{nil} value. This can be used to indicate that the object was created by the run-time environment, or by some other means outside the context of the running program.

5.2.2 Primitive Properties of Extended Object Graphs

The following rules define primitive judgements on the object graph, derived directly from objects in the extended object graph.

\[
(G(a') = (c, F, p_n, a)) \quad \frac{G \vdash_{og} a \texttt{crtd} a'}{(\text{Rule-Created})}
\]

\[
G(a) = (c, F, p_n, a_c) \quad \exists f \in \text{FieldName}[a' \in F(f)] \quad \frac{G \vdash_{og} a \texttt{ref} a'}{(\text{Rule-Refs})}
\]
Figure 5.2: The diagram above displays the creator information (in dotted arrows) which is also stored in the extended object graphs and links each object to its creator – the object which allocated it using \texttt{new}. For example, \texttt{B} created \texttt{F}. Formally, we write $G_A \vdash_{\text{crtd}} B \text{ crtd } F$. The creator information must be the same in both graphs if they are to be combined (see Section 5.4).
Inferring Ownership

**Rule-Created** specifies the `crtd` relation, linking objects to their creators. Notice that objects are not referenced directly in the relation, but that their addresses are used instead; this is a convenience used for all relations between objects, and which arises from the use of addresses rather than objects as field values.

**Rule-Refs** is possibly the most important rule of the system, and underlies most of the reasoning in this report. It simply states, for object addresses \( a, a' \in \text{Address} \), that \( a \text{ ref } a' \) in object graph \( G \) if and only if there is some field, \( f \), in \( a \) that contains \( a' \) as a value.

### 5.3 Paths in Object Graphs

There are two portions to the definition of paths in extended object graphs. The first provides the syntax, while the second relates the existence of paths to references in the object graphs.

**Definition 2** *Paths, ranged over by \( p \), are simply lists of addresses – paths may be of zero length. We add some syntactic sugar for paths, including a ‘cons’ operator, \( : \), for constructing them.*

\[
\begin{align*}
a & \in \text{Address} \\
p \in \text{Path} & ::= \, [\, ] \, | \, p : a
\end{align*}
\]

*The following equivalence is used for convenience, much like in many functional languages, and paths will invariably be considered modulo this equivalence.*

\[
[a_0, a_1, a_2, \ldots, a_{n-1}, a_n] \overset{\text{def}}{=} [a_0, a_1, a_2, \ldots, a_{n-1}] : a_n
\]

The above definition specifies an inductive construction for paths, familiar to functional programmers (though the paths grow rightward here, while functional lists usually grow leftwards).

There are two extra basic definitions which become useful when demonstrating properties of paths, which should again be familiar to functional programmers.

**Definition 3** *The length of a path is determined with length:*

\[
\begin{align*}
\text{length} & : \text{Path} \rightarrow \mathbb{N} \\
\text{length}([\, ]) & \overset{\text{def}}{=} 0 \\
\text{length}(p : a) & \overset{\text{def}}{=} 1 + \text{length}(p)
\end{align*}
\]
Figure 5.3: This figure illustrates a path in the object graph, \([A, B, C, D, F, G]\), which goes from \(A\) and to \(G\). Notice that there is a reference between \(G\) and \(H\), so this path can be extended to create a new path from \(A\) to \(H\).

**Definition 4** Two paths may be concatenated with the \(+ +\) operator:

\[
\begin{align*}
++ & : (\text{Path} \rightarrow \text{Path}) \rightarrow \text{Path} \\
p ++ [] & \overset{\text{def}}{=} p \\
p_1 ++ (p_2 : a) & \overset{\text{def}}{=} (p_1 ++ p_2) : a
\end{align*}
\]

We state without proof that \(+ +\) is associative: \((p_1 ++ p_2) ++ p_3 = p_1 ++ (p_2 ++ p_3)\). The superfluous brackets will be omitted accordingly. Additionally, \(\:\overset{\text{def}}{=}\) binds more tightly than \(+ +\), so that \((p_1 : a) ++ p_2 \neq p_1 : (a ++ p_2)\), which would make no sense given \(+ +\)'s definition on paths rather than addresses.

Now that paths are defined, the conditions under which they arise in extended object graphs must be specified.

\[
\begin{align*}
\text{(Rule-PathBase)} & \quad \frac{a \in \text{dom}(G)}{G \vdash \text{path} \ [a]} \\
\text{(Rule-PathInd)} & \quad \frac{G \vdash \text{path} \ (p : a_1) \quad G \vdash_{\text{og}} a_1 \ \text{ref} \ a_2}{G \vdash \text{path} \ (p : a_1) : a_2}
\end{align*}
\]

**Rule-PathBase** states that the unit-length path containing only the address \(a\) is present in the object graph \(G\) only if \(a\) is an address in \(G\). **Rule-PathInd** extends paths along
Inferring Ownership

chains of references in the object graph: A path ending at address $a_1$ may be extended to $a_2$ if and only if $a_1$ references $a_2$, illustrated on Snapshot B in Figure 5.3.

Notice that the empty path, [], is never present in any object graph but it still a path that can be used with path operations. This becomes particularly convenient for reasons which will become clear when proofs are required. Also notice that paths may be cyclic.

To complete the path definitions, it will be convenient to express the following properties of paths in object graphs:

\[ G \vdash \text{path } [a] ++ p \]
\[ G \vdash \text{path } ([a] ++ p) \text{ from } a \]

\[ G \vdash \text{path } p : a \]
\[ G \vdash \text{path } (p : a) \text{ to } a \]

The meaning of Rule-PathFrom and Rule-PathTo should be clear – paths starting at $a$ are from $a$ and paths ending at $a$ are to $a$.

\[ G \vdash \text{path } (p_1 : a) ++ p_2 \]
\[ G \vdash \text{path } ((p_1 : a) ++ p_2) \text{ via } a \]

Rule-PathVia states that a path $p$ goes via $a$ whenever it can be split into two portions, the first of which ends at $a$. Note that singleton paths, $[a]$, go from $a$, to $a$ and via $a$.

For shorthand, Rule-PathFrom and Rule-PathTo are combined in Rule-PathFromTo:

\[ G \vdash \text{path } p \text{ from } a_0 \]
\[ G \vdash \text{path } p \text{ to } a_n \]
\[ G \vdash \text{path } p \text{ from } a_0 \text{ to } a_n \]

In proofs, the consequences of this rule are not usually shown explicitly, relying instead on the properties defined in Rule-PathFrom and Rule-PathTo.

5.3.1 Reachability and Well-formed Object Graphs

Many of the definitions and rules only make sense when considering addresses which are connected to the root address in some way. We now define the set of such addresses:

**Definition 5** The set of reachable addresses in object graph $G$ is defined by:

\[ a \in \text{reachable}(G) \iff \exists p \in \text{Path}[G \vdash \text{path } p \text{ from } a_r \text{ to } a] \]
Chapter 5. Introduction to the Formal System

We now impose a constraint on well-formed object graphs so that only reachable addresses need to be considered. We say that an extended object graph $G$ is well-formed, $G \vdash \triangleleft_{\text{xog}}$, according to the following rule:

\[
\text{(Rule-WFGraph)} \quad \forall a \in \text{Address} [a \in \text{dom}(G) \Rightarrow a \in \text{reachable}(G)]
\]

From this point on, all graphs under consideration are assumed to be well-formed.

The domains of the $\text{ref}$ and $\text{crtd}$ relations can now be specified precisely:

\[
G \vdash_{\text{xog}} \text{ref} : \text{reachable}(G) \times \text{reachable}(G)
\]

\[
G \vdash_{\text{xog}} \text{crtd} : \text{reachable}(G) \times \text{Address}^N
\]

The domain of $\text{crtd}$ is somewhat more general than that of $\text{ref}$, as there is no constraint on the value that any object’s creator’s address can take. However, when $G \vdash_{\text{xog}} a_1 \text{ ref } a_2$, $a_1$ must be in the domain of $G$ by the top of Rule-Refs, while $a_2$ must be in $\text{reachable}(G)$ as there is a path to $a_2$ by virtue of the $a_1$’s reference.

5.4 Graph Combination

A means is required for generating the extended object graphs used in this report. Clearly, mapping all field values in a program heap to a singleton set containing only that value would be such an extended graph. However, there must be some facility for combining graphs so that it is possible to reason about how properties of extended graphs impact on the underlying program heaps from which they are constructed.

Firstly, it is not true that all graphs are suitable for combination: A primary example is graphs with different root addresses – how should two different roots be presented? Finding a solution to such problems is not useful in this particular context, as the representation of root is not, in general, part of any program’s mutable state, and only graphs from single programs are useful when examining its object-graph’s behaviour. The predicate, $\text{compatible}_{\text{xog}}$, specifies when graphs are suitable for combination:

**Definition 6** It is only appropriate to consider the combination of graphs $G_1 = (H_1, a_{r1})$ and $G_2 = (H_2, a_{r2})$ under the following conditions:

\[
\text{compatible}_{\text{xog}}((H_1, a_{r1}), (H_2, a_{r2})) \iff a_{r1} = a_{r2} \land \forall a \in \text{dom}(H_1) \cap \text{dom}(H_2) [H_1(a) = (c, F, p_n, a_c) \land H_2(a) = (c, F', p_n, a_c)]
\]

Definition 6 specifies that graphs can only be combined when they share a common root address, and when only the value of each object’s field map varies between the two graphs. Each object’s class, allocation point and creator should be identical in both graphs.
Inferring Ownership

Figure 5.4: This figure illustrates the function of the \( \cup_{\text{xog}} \) operator on the two example snapshots introduced in Figure 5.1. The resulting graph contains exactly the union of references from the original snapshots. Thus, every path occurring in either \( G_A \) or \( G_B \) is also present in the result.

The combination itself is performed by the \( \cup_{\text{xog}} \) operator:

**Definition 7** Graphs \( G_1 = (H_1, a_{r_1}) \) and \( G_2 = (H_2, a_{r_2}) \) can be combined according to the following definition:

\[
\cup_{\text{xog}} : (\text{XObjectGraph} \times \text{XObjectGraph}) \rightarrow \text{XObjectGraph}
\]

\[
(H_1, a_{r_1}) \cup_{\text{xog}} (H_2, a_{r_2}) \overset{\text{def}}{=} \begin{cases} 
(H_3, a_{r_3}) & \text{if } \text{compatible}_{\text{xog}}((H_1, a_{r_1}), (H_2, a_{r_2})) \\
\text{Udf} & \text{otherwise}
\end{cases}
\]

where \( H_3 \overset{\text{def}}{=} \{ (a \in \text{Address} \mapsto o \in \text{XObject}) \mid o = G_1(a) \cup_{\text{xobj}} G_2(a) \} \)

and \( a_{r_3} \overset{\text{def}}{=} a_{r_1} \) (Note that \( a_{r_1} = a_{r_2} \))

\[
\cup_{\text{xobj}} : (\text{XObject} \times \text{XObject}) \rightarrow \text{XObject}
\]

\[
 o_1 \cup_{\text{xobj}} o_2 \overset{\text{def}}{=} \begin{cases} 
 o_1 & \text{if } o_2 = \text{Udf} \\
 o_2 & \text{if } o_1 = \text{Udf} \\
 (c, F_3, p_n, a_c) & \text{if } o_1 = (c, F_1, p_n, a_c) \text{ and } o_2 = (c, F_2, p_n, a_c) \\
\text{Udf} & \text{otherwise}
\end{cases}
\]

where \( F_3 \overset{\text{def}}{=} \{ (f \mapsto as) \mid f \in (\text{dom}(F_1) \cup \text{dom}(F_2)) \land as = F_1(f) \cup F_2(f) \} \)

The definition above specifies two functions: \( o_1 \cup_{\text{xobj}} o_2 \) returns a new object where each of its fields occurs in at least one of its input objects and holds the union of the field's values from both \( o_1 \) and \( o_2 \). \( G_1 \cup_{\text{xog}} G_2 \) simply distributes the \( \cup_{\text{xobj}} \) operator across each of \( G_1 \)'s and \( G_2 \)'s objects. Of course, where the object occurs only in \( G_1 \) or only in \( G_2 \), the object appears unmodified in the result.
Intuitively, this graph combination operator yields a graph with all the references of \( G_1 \) and \( G_2 \) and no more. Diagrammatically, overlaying one graph with another gives the correct result (Figure 5.4).

### 5.5 Design Considerations

Even for the foundation structures already presented, a great deal of effort was required in order to provide natural, concise definitions. Where one definition is built upon another, qualifying statements and special cases should be avoided – poor definitions at the lower levels usually yield complexity higher up.

These complexities are not always immediately obvious, therefore a degree of iteration and refinement is required when assembling a formal system. Examples of refinements applied to the structures introduced in this section include:

- The definition of the ‘root’ object in an extended object graph is required to ‘anchor’ paths when considering domination. There were four ideas for the representation of the graph root:
  
  1. An object whose creator is \( \text{nil} \). This approach would model an initial object instantiated by the run-time environment, rather than by another object which exists in the system. However, such a representation would require that only one such object existed in the graph – there must be only one root – and would impose an undesirable constraint on the graph contents.
  
  2. A pseudo-object that has a path from itself to any objects with some property. Developing the previous idea, we tried this approach with the root having paths to all objects whose creator was \( \text{nil} \). This removed the constraints that the previous idea imposed, but complicated some of the reasoning, particularly as the root object was not a part of the object graph.
  
  3. Alternatively, we might regard any object without any incoming references as a root, though it seems reasonable that one might reference the top-level objects (so one can call-back, for example). There is no guarantee that a referenceless object would exist in an arbitrary graph without additional constraints.
  
  4. The final idea, and the approach taken here, is for the root simply to be a distinguished object in the graph. This has the advantage that no constraint is imposed on the structure of the graph, while the root is addressable. This simplifies the treatment of the root in the formal description, but comes at the cost of an extra clause to determine the compatibility of graphs for combination.

- The definitions and constraints on paths can have an effect on the consideration of domination. A number of refinements to path definitions were made to simplify reasoning:
Inferring Ownership

1. Paths are considered entirely in the context of an object graph. Clearly, a specific path, \( p \), such that \( G \vdash_{\text{path}} p \to a \) goes to \( a \) regardless of the context of \( G \). A more appropriate notation might therefore be \( G \vdash_{\text{path}} p \land \vdash p \to a \). However, such a decomposition of the statements offers little useful flexibility to the reasoning, and decouples paths from the graphs in which they occur.

2. Paths can occur which do not necessarily begin at the object root. It is sometimes useful to consider paths which do not start at the graph root – this is particularly true when reasoning about sub-portions of paths. Initially, it was the case that all paths started at root, as domination is only concerned about such paths. Formally: \( G \vdash_{\text{path}} p \Rightarrow G \vdash_{\text{path}} p \text{ from } a_r \), where \( G = \langle H, a_r \rangle \). However, it later became apparent that sub-paths may need to be considered, for example as a result of Rule-Via which splits paths into concatenated sections, thus paths starting at any object address are valid.

3. The notation \([a_1, a_2, \ldots, a_n]\), regarded now as syntactic sugar, could have potentially been considered as the fundamental notation for paths. However, induction over paths becomes much clearer with the ‘:’ (cons) operator than with the less-precise sugared notation. Furthermore, two rules were required to specify the \( \text{via} \) and \( \text{from} \) relations in terms of sugared paths; the inductive power of these pairs of rules now stems from the definition of \( +\), which is well-defined with the \([\] \) and ‘:’ constructors.
Chapter 6

Domination in Object Graphs

This chapter describes the formal system’s consideration of domination in object graphs – known to be a key property to enforcing static ownership. We define some structures for reasoning about domination, demonstrate that they behave intuitively and discuss how domination behaves under the graph combination operator defined in the previous chapter. Finally, some of the considerations influencing the system’s construction are reviewed.

6.1 Specifying Domination

As discussed in Chapter 3, it has been established that owners are always dominators[7]. Thus, when attempting to infer ownership in object graphs, the properties of domination are of obvious interest.

Informally, \( x \) dominates \( y \) when, in some graph, \( x \) occurs on every path through the graph from its root to \( y \). The prerequisites for formalising this property – specifically, graphs, graph roots and paths – were all specified in Chapter 5, so domination can be formalised immediately:

\[
\begin{align*}
\text{(Rule-Dom} & \quad a, a' \in \text{dom}(H) \quad \langle H, a_r \rangle \vdash_{\text{path}} p \text{ from } a_r \text{ to } a' \Rightarrow \langle H, a_r \rangle \vdash_{\text{path}} p \text{ via } a \\
& \quad \langle H, a_r \rangle \vdash_{\text{og}} a \text{ dom } a' 
\end{align*}
\]

The rule simply states that \( a \) dominates \( a' \) if and only if, for every path \( p \) to \( a' \) from the graph’s root, \( a_r \), that \( a \) must occur in that path. There is an additional condition that both \( a \) and \( a' \) must be addresses in the object graph. From this, by the assumption that all graphs, \( G \), are well-formed, it must be that \( a \) and \( a' \) are in \text{reachable}(G).
Figure 6.1: This figure shows examples of the paths from the graph root, A, to the E object. The longer path contains cycles – there are infinitely many paths to E, as every path can be extended with another trip around A, F and G. Suppose, however, that these were the only paths to E, then noticing that B and C were on all such paths would allow the conclusion that B and C both dominate E: $G \vdash_{og} B \text{ dom } E$, $G \vdash_{og} C \text{ dom } E$. This also illustrates why we expect the graph root to dominate everything – root is clearly on every path from root to any other object.
(a) Dominators for $G_1$

(b) Dominators for $G_2$

Figure 6.2: The dashed arrows (----) link dominators to their subordinate objects. For example, A dominates B in both graphs. The references are also present in the graph, to illustrate how these domination relationships are derived – recall that the root of the graph is A. Transitive edges are not shown: A also dominates C in both graphs, even though only edges from A to B and B to C are shown.
Inferring Ownership

Firstly, therefore, the domination relation exhibited by a graph has the following type:

\[ G \vdash_{og} \text{dom} : \mathcal{P}(\text{reachable}(G) \times \text{reachable}(G)) \]

Additionally, this condition avoids the possibility that there is no path from the graph’s root to \( a' \). If this were the case, then the top of \textsc{Rule-Dom} would hold without any other constraints, so every address \( a \) would dominate \( a' \).

6.2 Properties of Domination

This section proves some properties of the \text{dom} relations exhibited by extended object graphs.

6.2.1 \text{dom} is a Partial Order

It is valuable to show that new structures fit into familiar classifications and the expectation that \text{dom} should be a partial order is just one example. There is a danger when defining new structures that the formal definition differs subtly from the informal expectations. We would certainly hope that domination orders objects into ‘higher’ objects, such as the root object, and ‘lower’ objects, which occur deep inside aggregate structures. Therefore, it is important to check that these properties hold to ensure that we are not led astray by unproven assumptions, and that we exploit established results and intuition when proving more complicated properties later in the work.

To show that a relation is a partial order, it is sufficient to show that it is reflexive, transitive and anti-symmetric. Formal definitions of these properties can be found in the theorems below.

The first proof of this report is relatively simple, and serves as much to illustrate some of the formalism as it does to show that the \text{dom} relation is reflexive. Firstly, the theorem is introduced formally:

\textbf{Theorem 1} \ \forall G \in XObjectGraph, a \in \text{reachable}(G)[G \vdash_{og} a \text{ dom } a].

The proof proceeds simply by the rules of logical inference, and by those that have been defined in this and the previous chapter:

\textbf{Proof of Theorem 1:}

1: \ \langle H, a_r \rangle \vdash_{\text{path}} p \text{ from } a_r \text{ to } a \ \ (G \vdash \Diamond_{xog}, \text{ arbitrary } p)
Figure 6.3: Theorem 2 demonstrates that the \texttt{dom} relation (shown with dashed arrows as usual) is transitive: \[ G \vdash_{\text{og}} a_1 \text{ dom } a_2 \land G \vdash_{\text{og}} a_2 \text{ dom } a_3 \Rightarrow G \vdash_{\text{og}} a_1 \text{ dom } a_3 \]

Given the use of arbitrary \( G \) and \( a \), it can be concluded that \( \forall G, a [G \vdash_{\text{og}} a \text{ dom } a] \), which was to be shown. 

\textbf{Explanation of the proof:} As we wish to show the property for all object graphs and addresses, we choose arbitrary \( G \) and arbitrary address \( a \) from \( G \)'s domain. By virtue of \( G \)'s assumed good form, we can infer that there is a path from the graph's root, \( a_r \), to \( a \) – of course there may be many such paths, so we choose arbitrary path \( p \). There is then some formal manipulation of such a path, which essentially establishes that any path finishing at \( a \) also goes via \( a \). As \( p \) was arbitrary, we infer that all paths to \( a \) go via \( a \). As \( a \) and \( G \) were arbitrary, conclude that \( G \vdash_{\text{og}} a \text{ dom } a \) and that \( \text{dom} \) is reflexive.

\textbf{Notes to the proof:} This proof was originally not possible with early definitions of paths, when domination was irreflexive for all addresses other than the root (see Section 6.5). A special case was used to ensure that the root was dominated by itself, so that every object address had at least one dominator.

We now show that \( \text{dom} \) is transitive (Figure 6.3). Again, this is something we would intuitively expect, but that requires careful demonstration. Firstly, however, there are two properties of paths that must be shown:

\textbf{Lemma 1} \( G \vdash_{\text{path}} p_1 \text{ ++ } p_2 \land p_1 \neq [] \Rightarrow G \vdash_{\text{path}} p_1 \)

This will prove consistently important whenever a path property depends on properties of its prefixes and we make regular use of this lemma in proofs completed by induction on the length of paths. This proof, too, proceeds by such an induction:
Proof:

CASE 1: $p_2 = []$ (BASE CASE)

1: $G \vdash \text{path} p_1 ++ []$ \hspace{1cm} (Assumption)
2: $p_1 \neq []$ \hspace{1cm} (Assumption)
3: $G \vdash \text{path} p_1$ \hspace{1cm} (1, Defn ++)
4: $G \vdash \text{path} p_1 ++ [] \land p_1 \neq [] \Rightarrow G \vdash \text{path} p_1$ \hspace{1cm} (1, 3, \Rightarrow-introduction)

Case complete

CASE 2: $p_2 = p'_2 : a'$ (INDUCTIVE CASE)

5: $G \vdash \text{path} p_1 ++ (p'_2 : a')$ \hspace{1cm} (Assumption)
6: $p_1 \neq []$ \hspace{1cm} (Assumption)
7: $G \vdash \text{path} (p_1 ++ p'_2) : a'$ \hspace{1cm} (5, Defn ++)

If $p'_2 = []$:

8: $G \vdash \text{path} (p_1 ++ []) : a'$ \hspace{1cm} (7, Case assumption)
9: $G \vdash \text{path} p_1 : a'$ \hspace{1cm} (8, Defn ++)
10: $p_1 = p'_1 : a''$ \hspace{1cm} (6, 9)
11: $G \vdash \text{path} p_1$ \hspace{1cm} (9, 10, RULE-PATHIND)
12: $G \vdash \text{path} p_1 ++ p_2 \land p_1 \neq [] \Rightarrow G \vdash \text{path} p_1$ \hspace{1cm} (5, 6, 11, \Rightarrow-introduction)

Otherwise, if $p'_2 = p''_2 : a''$:

13: $G \vdash \text{path} (p_1 ++ (p''_2 : a'')) : a'$ \hspace{1cm} (7, Case assumption)
14: $G \vdash \text{path} ((p_1 ++ p''_2) : a'') : a'$ \hspace{1cm} (13, Defn ++)
15: $G \vdash \text{path} (p_1 ++ p''_2) : a''$ \hspace{1cm} (14, RULE-PATHIND)
16: $G \vdash \text{path} p_1 ++ (p''_2 : a'')$ \hspace{1cm} (15, Defn ++)
17: $G \vdash \text{path} p_1 ++ p''_2$ \hspace{1cm} (16, Case assumption)
18: $G \vdash \text{path} p_1$ \hspace{1cm} (17, Inductive hypothesis)
19: $G \vdash \text{path} p_1 ++ p_2 \land p_1 \neq [] \Rightarrow G \vdash \text{path} p_1$ \hspace{1cm} (5, 6, 18, \Rightarrow-introduction)

Case complete.

Therefore, $G \vdash \text{path} p_1 ++ p_2 \land p_1 \neq [] \Rightarrow G \vdash \text{path} p_1$, which was to be shown.

Explanation of the proof: Given a path in graph $G$ composed of two concatenated portions, $p_1 ++ p_2$, it must be shown that the earlier portion, $p_1$, is present in the graph. We use an induction over the length of the latter portion, $p_2$. For the base case, this latter portion may be empty – therefore $p_1$ must be the same as the original path, which we already know is present in the object graph, and that case is complete.

For the inductive case, we simply remove the last address in the path (which will come off the $p_2$ portion), which will be $p_1$ concatenated with some shortened version...
of \( p_2 \). By induction, it immediately follows that \( p_1 \) is in the object graph, and the proof is complete.

Most of the detail in this proof is simply manipulation of the path to put it in the forms expected by the rules.

Lemma 1 offers a corollary that will also prove useful in later work. It states that any prefix of a path in \( G \) has the same starting point as the original path:

**Corollary 1** \( G \vdash p_1 \quad \text{from} \quad a \) \( \land \quad p_1 \neq [] \) \( \Rightarrow \) \( G \vdash p_1 \quad \text{from} \quad a \)

This clearly follows directly from the definition of Rule-PathFrom and Lemma 1.

The following lemma states that all paths go via any addresses that their prefixes go via; the via relation for any path is a superset of the via relations for its prefixes.

**Lemma 2** \( G \vdash \text{path} \ p_1 \quad \text{from} \quad a \) \( \land \quad G \vdash \text{path} \ p_2 \quad \text{via} \quad a \) \( \Rightarrow \) \( G \vdash \text{path} \ p_1 \quad \text{from} \quad a \)

The proof is a straight-forward application of the path construction rules and of the rule which defines the via relation.

**Proof:**

1. \( G \vdash \text{path} \ p_1 \quad \text{from} \quad a \) \hspace{2em} (Assumption)
2. \( G \vdash \text{path} \ p_1 \quad \text{via} \quad a \) \hspace{2em} (Assumption)
3. \( G \vdash \text{path} \ ((p_1' : a) \quad \text{from} \quad a) \quad \text{from} \quad a \) \hspace{2em} (1, 2, Rule-PathVia)
4. \( G \vdash \text{path} \ ((p_1' : a) \quad \text{from} \quad a) \quad \text{via} \quad a \) \hspace{2em} (3, associativity of \( ++ \))
5. \( G \vdash \text{path} \ ((p_1' : a) \quad \text{from} \quad a) \quad \text{via} \quad a \) \hspace{2em} (4, Rule-PathVia)
6. \( G \vdash \text{path} \ ((p_1' : a) \quad \text{from} \quad a) \quad \text{via} \quad a \) \hspace{2em} (5, associativity of \( ++ \))
7. \( G \vdash \text{path} \ ((p_1 \quad \text{from} \quad a) \quad \text{from} \quad a) \) \hspace{2em} (2, 6, Rule-PathVia)

Proof complete.

The two preceding lemmata are now used to show that the \textit{dom} relation is transitive:

**Theorem 2** The \textit{dom} relation is transitive: \( G \vdash \text{og} \ a_1 \quad \text{dom} \quad a_2 \) \( \land \ G \vdash \text{og} \ a_2 \quad \text{dom} \quad a_3 \) \( \Rightarrow \) \( G \vdash \text{og} \ a_1 \quad \text{dom} \quad a_3 \)

**Proof of Theorem 2:**

Take an arbitrary graph \( G = \langle H, a_r \rangle \), then suppose that \( \langle H, a_r \rangle \vdash \text{og} \ a_1 \quad \text{dom} \quad a_2 \) and \( \langle H, a_r \rangle \vdash \text{og} \ a_2 \quad \text{dom} \quad a_3 \) then, to show transitivity, it must be shown that \( \langle H, a_r \rangle \vdash \text{og} \ a_1 \quad \text{dom} \quad a_3 \).
Inferring Ownership

1: \( \langle H, a_r \rangle \vdash_{og} a_1 \text{ dom } a_2 \) (Assumption)
2: \( \langle H, a_r \rangle \vdash_{og} a_2 \text{ dom } a_3 \) (Assumption)
3: \( \forall p \langle H, a_r \rangle \vdash_{\text{path}} p \text{ from } a_r \text{ to } a_2 \Rightarrow \langle H, a_r \rangle \vdash_{\text{path}} p \text{ via } a_1 \) (1, Rule-Dom)
4: \( \forall p \langle H, a_r \rangle \vdash_{\text{path}} p \text{ from } a_r \text{ to } a_3 \Rightarrow \langle H, a_r \rangle \vdash_{\text{path}} p \text{ via } a_2 \) (2, Rule-Dom)
5: \( \langle H, a_r \rangle \vdash_{\text{path}} p' \text{ from } a_r \text{ to } a_3 \) (5, instance of 4)
6: \( p' = (p'_1 : a_2) ++ p'_2 \) (6, Rule-PathVia)
7: \( \langle H, a_r \rangle \vdash_{\text{path}} (p'_1 : a_2) \text{ from } a_r \) (5, 7, Corollary 1)
8: \( \langle H, a_r \rangle \vdash_{\text{path}} (p'_1 : a_2) \text{ from } a_r \text{ to } a_2 \) (8, Rule-PathTo)
9: \( \langle H, a_r \rangle \vdash_{\text{path}} (p'_1 : a_2) \text{ via } a_1 \) (9, instance of 3)
10: \( \langle H, a_r \rangle \vdash_{\text{path}} (p'_1 : a_2) ++ p'_2 \text{ via } a_1 \) (10, Lemma 2)
11: \( \langle H, a_r \rangle \vdash_{\text{path}} p' \text{ via } a_1 \) (7, 11)
12: \( \langle H, a_r \rangle \vdash_{\text{path}} p' \text{ from } a_r \text{ to } a_3 \Rightarrow \langle H, a_r \rangle \vdash_{\text{path}} p' \text{ via } a_1 \) (5, 12, \( \Rightarrow \)-introduction)
13: \( \forall p \langle H, a_r \rangle \vdash_{\text{path}} p \text{ from } a_r \text{ to } a_3 \Rightarrow \langle H, a_r \rangle \vdash_{\text{path}} p \text{ via } a_1 \) (13, \( p' \) arbitrary)
14: \( \langle H, a_r \rangle \vdash_{og} a_1 \text{ dom } a_3 \) (14, Rule-Dom)

Which was to be shown. □

Explanation of the proof: The proof follows by patient application of the rules defining domination and the associated path properties. The intuition behind the proof is simple: Any path to \( a_3 \) must go via \( a_2 \) by the fact that \( G \vdash a_2 \text{ dom } a_3 \). Thus, any path to \( a_3 \) has a path to \( a_2 \) as a prefix. This prefix must go via \( a_1 \) because \( G \vdash a_1 \text{ dom } a_2 \). By lemma 2, the full path goes via all the addresses this prefix does, so all paths to \( a_3 \) go via \( a_1 \) and \( G \vdash a_1 \text{ dom } a_3 \) as required for transitivity.

Notes to the proof: Like many other proofs on the object graph, the assumption that the graph is well-formed (for example, on line 5) reduced the number of cases required. Consideration of the unreachable cases fails to tell us anything interesting about the graph or its transitivity: Either unreachable nodes are undominated, so the antecedent of the theorem does not hold, or unreachable nodes are dominated by everything, in which case the consequence of the theorem always holds.

The final stage for showing that \( \text{dom} \) is a partial order requires that we show \( \text{dom} \) is anti-symmetric – that is, if \( a_1 \) and \( a_2 \) mutually dominate one another, then we must check that \( a_1 \) and \( a_2 \) are identical:

**Theorem 3** \( \forall a_1, a_2 \left[ \langle H, a_r \rangle \vdash_{og} a_1 \text{ dom } a_2 \land \langle H, a_r \rangle \vdash_{og} a_2 \text{ dom } a_1 \Rightarrow a_1 = a_2 \right] \)

This proof is intuitively more complicated than preceding proofs, as it follows by a double induction over the lengths of the arbitrary paths which underly the domination property. See the ‘explanation of the proof’ for more details.

**Proof of Theorem 3:**
### Chapter 6. Domination in Object Graphs

1. \( \langle H, a_r \rangle \vdash_{\text{og}} a_1 \text{ dom } a_2 \)  
   (Assumption)
2. \( \langle H, a_r \rangle \vdash_{\text{og}} a_2 \text{ dom } a_1 \)  
   (Assumption)
3. \( \langle H, a_r \rangle \vdash \text{path } p_1 \text{ from } a_r \text{ to } a_1 \)  
   \( (G \vdash \diamond_{\text{og}}) \)
4. \( \langle H, a_r \rangle \vdash \text{path } p_2 \text{ from } a_r \text{ to } a_2 \)  
   \( (G \vdash \diamond_{\text{og}}) \)
5. \( \forall p [\langle H, a_r \rangle \vdash \text{path } p \text{ from } a_r \text{ to } a_2 \Rightarrow \langle H, a_r \rangle \vdash \text{path } p \text{ via } a_1 ] \)  
   \( (1, \text{ Rule-Dom}) \)
6. \( \forall p [\langle H, a_r \rangle \vdash \text{path } p \text{ from } a_r \text{ to } a_1 \Rightarrow \langle H, a_r \rangle \vdash \text{path } p \text{ via } a_2 ] \)  
   \( (2, \text{ Rule-Dom}) \)
7. \( \langle H, a_r \rangle \vdash \text{path } p_1 \text{ via } a_2 \)  
   \( (3, \text{ instance of 6}) \)
8. \( \langle H, a_r \rangle \vdash \text{path } p_2 \text{ via } a_1 \)  
   \( (4, \text{ instance of 5}) \)

It must then be shown that \( a_1 = a_2 \) by induction over the lengths of paths \( p_1 \) and \( p_2 \):

**Case 1:** \( p_1 = [a_r], \ p_2 = [a_r] \) (Base Case)

9. \( p_1 = ([] : a_2) ++ [\ ] \)  
   \( (7, \text{ Rule-PathVia}) \)
10. \( p_1 = ([] : a_2) \)  
    \( (9, \text{ Defn } ++ ) \)
11. \( p_1 = ([] : a_1) \)  
    \( (3, \text{ Rule-PathTo}) \)
12. \( a_1 = a_2 \)  
    \( (10, 11) \)

Case complete.

**Case 2:** \( p_1 = [a_r], \ p_2 = p_2 \ : a' \) (Base Case)

Proof as above.

**Case 3:** \( p_1 = p_1' : a', \ p_2 = [a_r] \) (Base Case)

Again, proof as above, but with the roles of \( p_1 \) and \( p_2 \) reversed.

**Case 4:** \( p_1 = p_1' : a_1', \ p_2 = p_2' : a_2' \) (Inductive Case)

13. \( p_1 = (p_1' : a_2) ++ p_2' \)  
    \( (7, \text{ Rule-PathVia}) \)
14. \( p_2 = (p_2' : a_1) ++ p_2' \)  
    \( (8, \text{ Rule-PathVia}) \)
15. \( \langle H, a_r \rangle \vdash \text{path } (p_1' : a_2) \text{ from } a_r \)  
    \( (3, 13, \text{ Corollary 1}) \)
16. \( \langle H, a_r \rangle \vdash \text{path } (p_2' : a_1) \text{ from } a_r \)  
    \( (4, 14, \text{ Corollary 1}) \)
17. \( \langle H, a_r \rangle \vdash \text{path } (p_1' : a_2) \text{ from } a_r \text{ to } a_2 \)  
    \( (15, \text{ Rule-PathTo}) \)
18. \( \langle H, a_r \rangle \vdash \text{path } (p_2' : a_1) \text{ from } a_r \text{ to } a_1 \)  
    \( (16, \text{ Rule-PathTo}) \)
19. \( \langle H, a_r \rangle \vdash \text{path } (p_1' : a_2) \text{ via } a_1 \)  
    \( (17, \text{ instance of 5}) \)
20. \( \langle H, a_r \rangle \vdash \text{path } (p_2' : a_1) \text{ via } a_2 \)  
    \( (18, \text{ instance of 6}) \)
21. \( a_1 = a_2 \)  
    \( \text{ (Inductive Hypothesis) } \)

Inductive case complete.

Therefore, for all addresses \( a_1 \) and \( a_2 \) in any well-formed graph, \( \langle H, a_r \rangle \vdash_{\text{og}} a_1 \text{ dom } a_2 \wedge \langle H, a_r \rangle \vdash_{\text{og}} a_2 \text{ dom } a_1 \Rightarrow a_1 = a_2 \), which was to be shown. \( \square \)

**Explanation of the proof** Firstly, the intuition behind the proof: Of course, it is possible that the paths to \( a_1 \) and \( a_2 \) are identical, in which case the constituent addresses must be equal, including \( a_1 \) and \( a_2 \). Otherwise, if \( a_1 \) dominates \( a_2 \) then \( a_1 \) must be on every path to \( a_2 \). Similarly if \( a_2 \) dominates \( a_1 \). Then, clearly, both the path to \( a_1 \) and \( a_2 \)
Inferring Ownership

can be truncated at points ending at \( a_2 \) and \( a_1 \) respectively. By iterating this process, we must eventually reduce the paths to the single-address paths which contain only the root address, \( a_r \). At this stage, both \( a_1, a_2 \) and \( a_r \) must all be equal.

The induction arises as a result of this iteration.

Notes to the proof Originally this proof was shown on graphs without the restrictions imposed by good form (Rule-WFGraph), but restricted to the reachable portion. Such restriction is now implicit in the definitions of graphs.

Outside of the reachable portion, if unreachable nodes are dominated by everything – an earlier definition – then the \( \text{dom} \) relation is symmetric.

We now have that \( \text{dom} \) is reflexive, transitive and anti-symmetric and have therefore shown the following:

**Corollary 2** Theorems 1, 2 and 3 imply that, for any extended object graph \( G \), \( \text{dom} \) is a partial order on the addresses in an extended object graph.

6.2.2 Root Dominates All

This system is intended to model the run-time version of ownership types. In the ownership type system, there is a global context – variously known as \textit{world}, \textit{root} or \textit{norep} – such that all objects may freely reference objects in that context. Thus, \textit{world} owns everything. This translates into the expectation here that the graph root should dominate all other objects in the graph: If this property does not hold, then we should assume that our definition is wrong and should be corrected.

**Theorem 4** \( \forall G = \langle H, a_r \rangle, a [G \vdash_{\text{og}} a_r \text{ dom } a] \).

**Proof:**

1. \( \langle H, a_r \rangle \vdash_{\text{path}} p \text{ from } a_r \text{ to } a \) \( (G \vdash \diamond_{\text{og}}) \)
2. \( p = [a_r]_{+} \vdash p' \) \( (1, \text{Rule-PathFrom}) \)
3. \( p = ([] : a_r)_{+} \vdash p' \) \( (2, \text{Defn}:) \)
4. \( \langle H, a_r \rangle \vdash_{\text{path}} ([] : a_r)_{+} \vdash p' \)
5. \( \langle H, a_r \rangle \vdash_{\text{path}} ([] : a_r)_{+} \vdash p' \text{ via } a_r \)
6. \( \langle H, a_r \rangle \vdash_{\text{path}} p \text{ via } a_r \)
7. \( \langle H, a_r \rangle \vdash_{\text{path}} p \text{ from } a_r \text{ to } a \Rightarrow \langle H, a_r \rangle \vdash_{\text{path}} p \text{ via } a_r \) \( (2, 6, \Rightarrow\text{-introduction}) \)
8. \( \langle H, a_r \rangle \vdash_{\text{og}} a_r \text{ dom } a \)
9. \( \forall a, G = \langle H, a_r \rangle [\langle H, a_r \rangle \vdash_{\text{og}} a_r \text{ dom } a] \) \( (7, \text{Rule-Dom}) \)

\( a, G \) arbitrary, \( \forall\text{-introduction} \)
Chapter 6. Domination in Object Graphs

Proof complete. \[ \square \]

Notes to the proof Once again, we look back at earlier definitions of domination and paths. In an earlier variant of the formal system, discussed in Section 6.5, it was the case that the root object had no dominator. Thus, root did not dominate everything. Such behaviour was ‘patched’ by adding a special rule specifying simply that \( \forall G = (H, a_r) \mid G \vdash a_r \text{ dom } a_r \), increasing the complexity of proofs without improving the elegance of the reasoning or the intuitive quality of the work.

6.3 Dominator Functions

When considering ownership, it only makes sense to consider a single owner for every address. Therefore, if domination is to underlie our investigation of ownership, it will later become convenient to consider domination dictated by functions that return an address’s dominator, rather than use of the \textit{dom} relation:

Definition 8 A dominator function is a total function, \( D : \text{reachable}(G) \rightarrow \text{reachable}(G) \), and is well-formed with respect to an extended object graph, \( G \), under the following rule:

\[
\frac{\forall a \in \text{dom}(G) \mid G \vdash \text{og } d(a) \text{ dom } a}{G \vdash d \Diamond \text{dom}} \tag{Rule-DomFunc}
\]

Thus the application of a dominator function, \( D \), to an address \( a \) must return an address that dominates \( a \). Therefore, dominator functions are simply projections of the \textit{dom} relation into functions, so there is a large space of dominator functions for any single object graph. Also notice that the identity function on addresses, \( \text{id} \) is a dominator function, as the \textit{dom} relation is reflexive. Additionally, by \textit{dom}’s anti-symmetry, if \( D(a) = a' \land D(a') = a \) then \( a = a' \). Finally, root will always be a fixed point of dominator functions, \( D \) such that \( D(a_r) = a_r \), as root is the only object to dominate the root.

6.4 Domination under Graph Combination

There is clearly no guarantee that a dominator function for an object graph, \( G_1 \), will be a valid dominator function for its union with a second graph, \( G_1 \cup_{\text{og}} G_2 \). New paths can always be produced in the union, constructed from portions of the path present in \( G_1 \) and \( G_2 \) but without having been linked in the individual graphs.

Alisdair Wren
Inferring Ownership

Figure 6.4: Domination according to Rule-Dom, where X is the root object. In the top pair of diagrams, the addition of a new reference (solid arrows) increases the size of the dom relation (shown in dashed arrows), as Z is now reachable and is dominated by X. However, in the lower pair of diagrams, the addition of a new reference to (c) shrinks the domination relation, as Y is no longer on every path from X to Z.

Additionally, however, a dominator function for the union, $G_1 \cup_{xog} G_2$, may not be valid on $G_1$ or $G_2$. In particular, the function, $D$, may specify that $D(a_1) = a_2$ but $a_2$ may not be present in both graphs.

Thus, the effect of graph combination on domination is a little chaotic. In the following section, we see how earlier versions of the formalism behaved more desirably, but in Chapter 7, we see that this behaviour is sacrificed in order to provide a cleaner formalism when dealing with ownership.

6.5 Design Considerations

6.5.1 Monotonicity of Domination under Graph Combination

As dom is applicable only to addresses in reachable($G$) according to Rule-Dom, an addition of a reference (and, therefore, the combination of graphs) may either increase or decrease the size of the dom exhibited by a graph, as illustrated in Figure 6.4, depending on the
Chapter 6. Domination in Object Graphs

Figure 6.5: This is how Figure 6.4 would appear had domination been defined according to Rule-Dom\textsubscript{old}. This old rule does not restrict domination only to reachable addresses, so both X and Y dominate Z, which has no path linking it to root. Diagrams (c) and (d) from Figure 6.4 remain the same, so the dom relation always shrinks with the old definition of domination.

structure of the graph.

An earlier definition of Rule-Dom was:

\[(\text{RULE-Dom}_{\text{old}}) \quad \forall p[G \vdash \text{path} p \text{ from } a_r \text{ to } a'] \Rightarrow G \vdash \text{og } a \text{ dom } a']\]

With this definition, every address dominates any other not reachable from the root: From \(-\exists p[G \vdash \text{path} p \text{ from } a_r \text{ to } a']\), \(G \vdash \text{og } a \text{ dom } a')\) can be immediately derived from Rule-Dom\textsubscript{old}, thus \(\forall a_1[\forall a_2 \notin \text{reachable}(G)[G \vdash a_1 \text{ dom } a_2]]\). Therefore, as illustrated in Figure 6.5, Z would be dominated by both X and Y in the situation shown in 6.4(a), and the dom relation would shrink in both cases: Reference addition would be monotonic with respect to the domination relation\(^1\). This implies that, if the references of \(G_1\) are a subset of those in \(G_2\), then the domination relation of \(G_1\) contains that of \(G_2\).

We originally thought this property would be desirable to simplify the reasoning – finding familiar and well-understood attributes for new definitions invariably brings with it a number of useful, general results and improved intuition. Despite this, Rule-Dom\textsubscript{old} was abandoned in favour of the new version: While the unreachable portion gives rise to good behaviour for the dom relation, divisions between the reachable and unreachable portions of the graph gave rise to more cases in the proofs, as well as undefined portions in later functions on object addresses. Additionally, a more attractive form of monotonicity can be achieved when we examine ownership in the next chapter.

Thereorefore, a number of modifications were made so that only the reachable portion of an

\(^1\text{This is not the classic monotonicity, as the dom relation shrinks as the set of references grows – it might be better to consider the growing set of domination links not in the graph as the number of references increases}\)
Inferring Ownership

object graph was considered. As such, we adopted the newer definition of Rule-DOM and added the constraints imposed on object graphs by good form (Rule-WFGraph), so that only reachable addresses are present in object graphs.

6.5.2 Reflexivity

While not a decision related to the definition of the dom relation, the definition of paths from Section 5.3 had some important effects on its characteristics. Early in the work, paths were lists of references rather than lists of addresses: For example \([a_1, a_2, a_3, \ldots, a_{n-1}, a_n]\), where \(G \vdash a_1 \text{ref} a_2, G \vdash a_2 \text{ref} a_3,\) and so on. The via relation was also defined differently, so that paths only went via any address occurring in the first element of any link in the path – so in the previous example, the path goes via \(a_2\) but not \(a_n\), which occupies only the second position in a path link. This meant that, in general, no path went via its destination. Thus, with the exception of the root address, no object was on all paths to itself and, therefore, no object dominated itself. At the same time, it was possible for the root address to have no dominator if there were no path cycling back to the root. (This is, of course, limited by the restriction of dom to reachable addresses, as discussed earlier. The root object would not have been considered reachable under this old definition of paths and the via relation!)

Whilst it was possible to add a special case for the root address such that root always dominated itself, it seemed more appropriate to avoid special cases and allow domination to be reflexive. By redefining paths to be object addresses, albeit constrained to those which referenced one another in the object graph, all paths to any object address \(a\) go via \(a\), thus \(\forall a \in \text{dom}(G) \mid G \vdash \text{og} a \text{ dom} a\) – dom is reflexive. This property is shown formally in Theorem 1.

6.6 Summary

Domination is important in the treatment of ownership types, as the type system ensures that static owners are always run-time dominators in program object heaps[7]. We have established that domination is a partial order on objects in an object graph and that root dominates all addresses according to our definitions – both properties that are important if the definitions are to be sound with respect to our intuitions about domination.

In this chapter, we also showed how the design of formal systems, like software, must evolve as subtle problems and inadequacies are exposed. Indeed, a great deal of effort has been expended respecifying the system in order to allow clear, elegant reasoning later in the work.

In the following chapter, we will observe how domination does not fully describe object ownership, and will examine more of the formal system to address these shortcomings.
Chapter 7

Ownership in Object Graphs

In this chapter we show how domination is insufficient to characterise ownership, and propose some solutions. We demonstrate how to obtain a valid ownership from the domination characteristics of an object graph and, having given a formal description of ownership precision, continue to show that this is the most precise possible ownership. We conclude by demonstrating that this is a means for obtaining the most precise invariant ownership across a set of program heaps.

7.1 Dominators Are Not Owners

As discussed in Section 4.4.5, there is a need to seek the most precise ownership when inferring this information from object graphs. Otherwise, the power of ownership types will be diluted. However, when examining the precision of domination under combination, we discovered that domination does not fully characterise the ownership in object graphs, especially when considering more than one graph.

Recall Figure 6.2, duplicated in Figure 7.1. It is well known that owners should always be dominators, so it seems reasonable to believe that valid ownerships may consist of those dominations which persist over both graphs. Thus, taking the intersection of the dominations from 7.1(a) and 7.1(b), we observe that A dominates B, which in turn dominates C in both graphs. C dominates D only in $G_A$, so we ‘fall back’ to B as a dominator for D in the intersection. C dominates E in both graphs. If we use this intersection domination relation as an ownership, then the diagram in 7.1(c) is the result. However, notice that the reference from D to E breaks the ownership variant, despite being a valid dominator.

We are therefore forced to conclude that dominators are not, in general, valid owners, and consider other alternatives for characterising ownership in object graphs.
Figure 7.1: The dashed arrows (\(\rightarrow\)) link dominators to their subordinate objects. Notice that \(C\) dominates \(E\) in both object graphs, so it seems reasonable that \(C\) should own \(E\), as shown in (c). However, the highlighted reference from \(G_A\) breaks \(C\)'s ownership boundary, so we conclude that dominators are not always valid owners.
7.2 Ownership Functions

In the previous section, it was described how dominator functions – functions \( f \) for which \( G \vdash f \Diamond_{\text{dom}} \) – cannot be used in general as ownerships. To characterise ownership, therefore, we introduce the definition of an ownership function:

**Definition 9** A total function, \( O : \text{reachable}(G) \rightarrow \text{reachable}(G) \) is an ownership function with respect to an extended object graph \( G \) (written \( G \vdash O \Diamond_{\text{own}} \)), according to the following rule:

\[
\frac{O(a_r) = a_r}{\langle H, a_r \rangle \vdash \text{og} \ a \text{ ref } a' \Rightarrow \exists k \geq 0 [O^k(a) = O(a')]}
\]

The constraint \( G \vdash \text{og} \ a \text{ ref } a' \Rightarrow \exists k \geq 0 [O^k(a) = O(a')] \) was first described by Clarke[5, page 48] in a thesis originally describing ownership and the owners-as-dominators property.

Ownership functions specify the owner address for every address in the object graph, which implies locations for the ownership boxes seen earlier in Chapter 3, through which references cannot pass. **Rule-OwnFunc** specifies that references can point from \( a \) to \( a' \) if:

- \( a' \) is inside \( a \)'s ownership box so that \( a = O(a') \) and \( k = 0 \) – for example, the reference from \( B \) to \( D \) in Figure 7.2;
- \( a \) and \( a' \) are inside the same ownership box so that \( O(a) = O(a') \) and \( k = 1 \) – for example, the reference from \( C \) to \( D \) in Figure 7.1(c);
- or \( a \)'s ownership box is inside that of \( a' \). Several applications of the ownership function are then required to take \( a \)'s context into that of \( a' \), hence \( k > 1 \). This situation is exemplified by the reference from \( G \) to \( A \) in Figure 7.2.

The equivalence between the ownership function and the ownership boxes is illustrated in Figure 7.2. Perhaps the only unexpected trait of the function is the reflexive arc for the graph root, \( A \). This is specified by the \( O(a_r) = a_r \) clause of **Rule-OwnFunc**, and serves to ensure that the root object is assigned an owner – it is the only object in the graph which may not have an incoming reference, as all other graphs must be linked to the root in order to be in \( \text{reachable}(G) \).

As every object must have an owner, and as the root should be above all other objects, we expect ownership functions to specify a spanning tree on the addresses of an object graph. Therefore, from any address, \( a \), we should be able to apply the ownership function, \( O \), a certain number of times, \( k \) – written \( O^k(a) \) – such that we arrive at the root address. This property, not only vital for confirming this intuition but also for use in future proofs, is confirmed by the proof of Theorem 5:
Inferring Ownership

(a) Ownership boxes for $G_B$

(b) Ownership function for $G_B$

Figure 7.2: An ownership function for $G_B$, also presented in the more familiar ownership boxes. We have rather presumptuously taken the domination tree from 7.1(b) as an ownership tree, even though this was shown to be generally unsound in Section 7.1. That approach in this case is justified later, in Theorem 10.

**Theorem 5** $G = \langle H, a_r \rangle \vdash O \diamond_{\text{own}} \forall a \in \text{dom}(G) \exists k[O^k(a) = a_r]$

**Proof:** By induction on paths from the root, $a_r$, to arbitrary address $a$

1: $G \vdash O \diamond_{\text{own}}$ \hspace{1cm} (Assumption)
2: $G \vdash \text{path } p \text{ from } a_r \text{ to } a$ \hspace{1cm} ($G \vdash \diamond_{\text{log}}$)

**Case 1:** length($p$) = 1 (Base Case)

3: $p = [a_r]$ \hspace{1cm} (2, Case assumption, Rule-PathFrom)
4: $a_r = a$ \hspace{1cm} (2, 3, Rule-PathTo)
5: $O^1(a) = a_r$ \hspace{1cm} (1, 4, Rule-OwnFunc)
6: $\exists k[O^k(a) = a_r]$ \hspace{1cm} (5, $\exists$-introduction)

Case complete.

**Case 2:** length($p$) = $n$ (Inductive Case)

In this inductive step, we can assume that all addresses, $a'$, in any prefix of a path to $a$ have a $k$ such that $O^k(a') = a_r$. Note that the induction is not over $k$, but over the length of these possible paths.

7: $p = (p_2 : a_1) : a$ \hspace{1cm} (3, Case assumption, Rule-PathTo)
Chapter 7. Ownership in Object Graphs

8: \( G \vdash_{og} a_1 \text{ ref } a \)  

9: \( \exists j [O^j(a_1) = O(a)] \)  

10: \( O^j(a_1) = O(a) \)  

11: \( G \vdash_{path} p_2 : a_1 \text{ from } a_r \)  

12: \( G \vdash_{path} p_2 : a_1 \text{ from } a_r \text{ to } a_1 \)  

13: \( \exists l [O^l(a_1) = a_r] \)  

14: \( O^l(a_1) = a_r \)  

If \( j_1 \geq l_1 \) then \( O^{j_1}(a_1) = O^{l_1}(a_1) = a_r \) by virtue of \( a_r \) being a fixed point of \( O \), and as \( O(a) = O^{j_1}(a_1) \) then \( \exists k [O^k(a) = a_r] \).

If \( j_1 < l_1 \) then:

15: \( l_1 = j_1 + m \)  

16: \( O^{j_1+m}(a_1) = a_r \)  

17: \( O^{j_1+m}(a_1) = O^m(O(a)) \)  

18: \( O^{m+1}(a) = a_r \)  

19: \( \exists k [O^k(a) = a_r] \)  

Case complete.

Therefore, for every address in \( G \) – and hence in \( \text{reachable}(G) \) – \( O \) is a spanning tree starting from the graph root for \( G \) wherever \( O \bowtie_{\text{own}} \).

Explanation of the proof: We wish to show that there is some \( k \) such that \( O^k(a) = a_r \), the root of the object graph. We show this property by induction on the length of the path to address \( a \) from the root, \( a_r \) – this may make more sense when placed in the context of Figure 7.2, where it can be observed that the lengths of paths from the root give an upper bound on the ‘depth’ in the ownership tree.

The base case, when the path is of unit length should be clear – the address \( a \) must be \( a_r \), so the case is completed.

For the inductive case, we take advantage of the fact that the penultimate address in the path to \( a \), called \( a_1 \), must reference \( a \). As \( O \) is an ownership function, the constraint on \( O \) from \( \text{RULE-OWNFUNC} \) must apply, so \( O^j(a_1) = O(a) \) for some \( j \). By the inductive hypothesis and the fact that the path to \( a_1 \) from the root is shorter than the path to \( a \), we know that there is an \( l \) such that \( O^l(a_1) = a_r \). These two equalities relate \( a_r \) to \( O(a_1) \) and in turn to \( O(a) \), thus the result follows from simple manipulation based on the fact that \( O \) is a function – especially that it is ‘deterministic’, \( O(a_1) = O(a_2) \Rightarrow O(O(a_1)) = O(O(a_2)) \).

Notes to the proof: The proof uses the fact that, for valid ownership functions on graph \( \langle H, a_r \rangle \), \( O(a_r) = a_r \). The spanning-tree property would hold here even if this were not the case, but the proof would be more complicated. There are later theorems where it is unclear how their properties could be shown without this constraint on ownership functions.
Inferring Ownership

Figure 7.3: Lemma 3 states that any path to \( a \) must go via all the (possibly indirect) owners of \( a \): \( O(a), O(O(a)), \ldots, O^n(a) \) for \( 0 \leq n \leq \infty \). The light arrows represent the references making up the path, with the heavy arrows reflecting subsequent applications of the ownership function up the path towards the root.

The spanning tree theorem was motivated by later proofs, which relate ownership to domination, but it is usually the immediate consequence in Corollary 3 that is referenced.

A fixed point of a function, \( f \), is an \( x \) such that \( f(x) = x \). The following corollary of Theorem 5 is often used in later proofs:

**Corollary 3** Theorem 5 implies, for some ownership function \( O \) such that \( G \vdash O \diamond \text{own} \), that \( a_r \) is the only fixed point for \( O \). If this were not the case, a chain of applications of \( O \) would fall into a repeating cycle for some \( a \), and there would be no \( k \) such that \( O^k = a_r \), contradicting Theorem 5.

In general, \( G \vdash O \diamond \text{own} \wedge \exists n \geq 1 [O^n(a) = a] \Rightarrow a = a_r \).

7.3 Owners as Dominators

The literature introducing ownership types\([7, 6]\) makes constant references to the “owners-as-dominators” property. It is therefore important to check that ownership functions are indeed dominator functions (we have already shown, in Section 7.1, that dominators are not owners).

First, a lemma is required, which shows that a path to any address, \( a_1 \), must go via all of its (possibly indirect) owners, \( O^a \) for \( 0 \leq n \leq \infty \), as illustrated by a path in Figure 7.3.
Chapter 7. Ownership in Object Graphs

The proof proceeds by a double induction on both the length of such a path, and on the number, \( n \), of applications of the ownership function:

**Lemma 3**  \( G \vdash O \odot \text{own} \land G \vdash \text{path } p \text{ from } a_r \text{ to } a_1 \Rightarrow \forall n \geq 0 [G \vdash p \text{ via } O^n(a_1)] \)

**Proof:**

1. \( G \vdash O \odot \text{own} \) (Assumption)
2. \( G \vdash \text{path } p \text{ from } a_r \text{ to } a_1 \) (Assumption)

**Case 1:** \( p = [a], n = 0 \) (Base Case for \( p \) and \( n \))

3. \( p = [a_1] \) (2, Rule-PathTo, Case assumption)
4. \( O^0(a_1) = a_1 \) (Defn of \( O^n \))
5. \( p = ([] : a_1) ++ [] \) (3, Defn ++ , :)
6. \( G \vdash p \text{ via } a_1 \) (3, 5, Rule-PathVia)
7. \( G \vdash p \text{ via } O^0(a_1) \) (4, 6)

Case complete.

**Case 2:** \( p = [a], n > 0 \) (Base Case for \( p \), Inductive Case for \( n \))

8. \( p = [a_r] = ([] : a_r) ++ [] \) (2, Rule-PathFrom, Case assumption)
9. \( G \vdash p \text{ via } O^{n-1}(a_1) \) (1, 2, Inductive Hypothesis on \( n \))
10. \( p = ([] : O^{n-1}(a_1)) ++ [] \) (9, Rule-PathVia)
11. \( O^{n-1}(a_1) = a_r \) (8, 10, Defn ++ , :)
12. \( O(a_r) = a_r \) (1, Rule-OwnFunc)
13. \( O(O^{n-1}(a_1)) = a_r \) (11, 12)
14. \( G \vdash p \text{ via } O^n(a_1) \) (9, 10, 8)

Case complete.

**Case 3:** \( p = (p_1 : a_2) : a_1, n = 0 \) (Inductive Case for \( p \), Base Case for \( n \))

Similar to the proof when \( p = [a], n = 0 \), above.

**Case 4:** \( p = (p_1 : a_2) : a_1, n > 0 \) (Inductive Case for \( p \), Inductive Case for \( n \))

15. \( p = (p_1 : a_2) : a_1 \) (Case assumption)
16. \( G \vdash p \text{ via } O^{n-1}(a_1) \) (15, Inductive Hypothesis for \( n \))
17. \( p = (p_2 : O^{n-1}(a_1)) ++ p_3 \) (16, Rule-PathVia)

If \( p_3 = [] \), then \( O^{n-1}(a_1) = a_1 \), so by Corollary 3, \( a_1 = a_r \). Thus, \( O^n(a_1) = a_r \) and, as all paths in \( \langle H, a_r \rangle \) go via \( a_r \) by Theorem 4, \( G \vdash \text{path } p \text{ via } O^n(a_1) \) and the desired property holds.

Consider instead, therefore, the case where \( p_3 \neq [] \):
Inferring Ownership

18: \[ G \vdash \text{path} \left( p_2 : O^{n-1}(a_1) \right) \] from \( a_r \) (2, 17, Corollary 1)

19: \[ G \vdash \text{path} \left( p_2 : O^{n-1}(a_1) \right) \] from \( a_r \) to \( O^{n-1}(a_1) \) (18, Rule-PathTo)

20: \[ \forall k \left[ G \vdash \text{path} \left( p_2 : O^{n-1}(a_1) \right) \right] \] via \( O^k(0^{n-1}(a_1)) \) (19, Inductive Hypothesis on \( p \))

21: \[ G \vdash \text{path} \left( p_2 : O^{n-1}(a_1) \right) \] via \( O(a_1) \) (20, \( k = 1 \), \( \forall \)-elimination)

22: \[ G \vdash \text{path} \left( p_2 : O^{n-1}(a_1) \right) \] via \( O(a_1) \) (17, 21, Lemma 2)

Case complete.

Thus \( G \vdash O \diamond_{\text{own}} \land G \vdash \text{path} \left( p \right) \) from \( a_r \) to \( a \) \( \Rightarrow \forall n \left[ G \vdash \text{path} \left( p \right) \text{ via } O^n(a) \right] \), which was to be shown.

Explanation of the proof: This proof demonstrates, for ownership function \( O \) and object graph \( G \), that all paths, \( p \), to arbitrary address \( a \) from the root, \( a_r \), go via all indirect owners, \( O^n(a) \) for all \( n \). The proof is based on a double induction on both the length of the path and on the size of \( n \).

The base cases should be clear – if the path is of unit length, then \( a = a_r \), and all such paths go via \( O^n(a_r) \), as \( O^n(a_r) = a_r \) for all \( n \). If \( n = 0 \), then the paths need only go via \( a \), which follows immediately from Rule-Via.

Only the inductive case then remains. By induction, we know that path \( p \) goes via \( O^{n-1}(a) \), thus \( p \) can be decomposed into a path ending at \( O^{n-1}(a) \) and the rest of the path, continuing to \( a \) in the object graph. Again, by induction, we know that this smaller path must go via \( O(O^{n-1}(a)) = O^n(a) \) and so the longer path must also go via \( O^n(a) \). There is the additional case that \( O(a) = a \), which does not shorten the path and precludes the use of induction, but as the graph root is the only fixed point of \( O \), \( a = a_r \) and all paths from \( a_r \) go via \( a_r \).

Notes to the proof: Here, the theorem becomes much simpler with the graph’s root as a fixed point of ownership function \( O \). If this were not the case, then the number of applications, \( n \), of \( O \) to address \( a \) would have to be bounded on how \( a \) is positioned in the ownership tree. For example, if \( O^n(a) = a_r \), then clearly any path from \( a_r \) to \( a \) cannot go via \( O^{n+1}(a) \) as it must be undefined.

The fact that this property holds was not entirely obvious intuitively. Early work on the proof involved tracing paths through both the object graph and ownership tree, and ‘stitching’ them together logically.

Lemma 3 paves the way to Theorem 6, which states that all ownership functions are dominators – the “owners-as-dominators” property from the ownership types literature:

Theorem 6 Ownership functions are also valid dominator functions: \( G \vdash O \diamond_{\text{own}} \Rightarrow G \vdash O \diamond_{\text{dom}} \)

Proof of Theorem 6: By contradiction

Suppose for contradiction that \( G \vdash O \diamond_{\text{own}} \) but \( G \not\vdash O \diamond_{\text{dom}} \).
Chapter 7. Ownership in Object Graphs

1: \( G \vdash O \Diamond_{\text{own}} \) (Assumption)
2: \( G \nvdash O \Diamond_{\text{dom}} \) (Assumption)
3: \( \neg \forall a \in \text{dom}(G) \left[ G \vdash_{\text{og}} O(a) \text{ dom } a \right] \) (2, Rule-DomFunc)
4: \( \neg \forall a \in \text{dom}(G) \left[ \forall p \left[ G \vdash_{\text{path}} p \text{ from } a_r \text{ to } a \Rightarrow G \vdash_{\text{path}} p \text{ via } O(a) \right] \right] \) (3, Rule-Dom)
5: \( \exists a \in \text{dom}(G) \left[ \exists p \left[ G \vdash_{\text{path}} p \text{ from } a_r \text{ to } a \land G \nvdash_{\text{path}} p \text{ via } O(a) \right] \right] \) (4, equivalent)
6: \( G \vdash_{\text{path}} p_1 \text{ from } a_r \text{ to } a_1 \) (5, \( \exists \)-elimination)
7: \( G \nvdash_{\text{path}} p_1 \text{ via } O(a_1) \) (5, \( \exists \)-elimination)
8: \( \forall n \left[ G \vdash_{\text{path}} p_1 \text{ via } O^n(a_1) \right] \) (1, 6, Lemma 3)
9: Contradiction (7, instance of 8)

Thus, \( G \vdash O \Diamond_{\text{own}} \Rightarrow G \vdash O \Diamond_{\text{dom}} \), which was to be shown. \( \square \)

**Notes to the proof:** That ownership functions are dominator functions is a simple consequence of Lemma 3. Intuitively, the lemma specifies that all indirect owners of \( a \) exist on all paths from the graph root to \( a \). Clearly, therefore, these owners must dominate \( a \), so ownership functions must be dominator functions.

Note that Lemma 3 is more powerful than we need in this proof – we need only that paths to \( a \) go via the immediate owner. However, this extra power is required as part of the inductive hypothesis in the proof of the lemma itself.

### 7.4 Immediate Dominators

When we observed that dominators do not work in general as owners (referring back to Figure 7.1), the problem appears to be that the potential owner was an indirect dominator of the owned object, which offered the opportunity for the ownership invariant to be broken. In this section, we show that the immediate dominator of an object is a suitable owner.

The following two rules take the immediate dominators from an extended object graph:

\[
\text{(Rule-IDomRoot)} \quad \langle H, a_r \rangle \vdash_{\text{og}} a_r \text{ idom } a_r
\]

\[
\text{(Rule-IDom)} \quad \begin{array}{c}
\quad a_1 \neq a_2 \\
\quad G \vdash_{\text{og}} a_1 \text{ dom } a_2 \\
\quad a_3 \neq a_2 \land G \vdash_{\text{og}} a_3 \text{ dom } a_2 \Rightarrow G \vdash_{\text{og}} a_3 \text{ dom } a_1 \\
\end{array} \quad \begin{array}{c}
\quad G \vdash_{\text{og}} a_1 \text{ idom } a_2
\end{array}
\]

**idom** is a relation which takes the minimal cover of the **dom** relation\(^1\) in **Rule-IDom**. **Rule-IDomRoot** adds that the root always immediately dominates itself – it is clear that this must be added otherwise **idom** is immediately discounted from forming a valid ownership function according to the constraints imposed by **Rule-OwnFunc**.

\(^1\)The **minimal cover** will yield the original **dom** relation under the reflexive and transitive closure.
**Inferring Ownership**

If \textit{idom} is to be used as an ownership function, it must be checked that it gives a unique immediate dominator to every address:

**Theorem 7** \(G \vdash_{\text{og}} a_1 \text{idom } a_3 \land G \vdash_{\text{og}} a_2 \text{idom } a_3 \Rightarrow a_1 = a_2\)

**Proof:** By contradiction

Assume the converse, specifically that: \(\exists a_1, a_2, a_3 \in \text{dom}(G)[G \vdash_{\text{og}} a_1 \text{idom } a_3 \land G \vdash_{\text{og}} a_2 \text{idom } a_3 \land a_1 \neq a_2]\). A contradiction must then be shown.

1: \(G \vdash_{\text{og}} a_1 \text{idom } a_3\) (Assumption)
2: \(G \vdash_{\text{og}} a_2 \text{idom } a_3\) (Assumption)
3: \(G \vdash_{\text{og}} a_1 \neq a_2\) (Assumption)
4: \(G \vdash_{\text{og}} a_1 \text{dom } a_3\) (1, Rule-IDom)
5: \(G \vdash_{\text{og}} a_2 \text{dom } a_3\) (2, Rule-IDom)
6: \(\forall a_4[a_4 \neq a_3 \land G \vdash_{\text{og}} a_4 \text{dom } a_3 \Rightarrow G \vdash_{\text{og}} a_4 \text{dom } a_1]\) (1, Rule-IDom)
7: \(\forall a_4[a_4 \neq a_3 \land G \vdash_{\text{og}} a_4 \text{dom } a_3 \Rightarrow G \vdash_{\text{og}} a_4 \text{dom } a_2]\) (2, Rule-IDom)
8: \(G \vdash_{\text{og}} a_2 \text{dom } a_1\) (3, 5, instance of 6)
9: \(G \vdash_{\text{og}} a_1 \text{dom } a_2\) (3, 4, instance of 7)
10: \(a_1 = a_2\) (8, 9, Theorem 3)
11: Contradiction (1, 10)

Therefore, conclude that \(G \vdash_{\text{og}} a_1 \text{idom } a_3 \land G \vdash_{\text{og}} a_2 \text{idom } a_3 \Rightarrow a_1 = a_2\), which was to be shown.

**Explanation of the proof:** The proof is simply a matter of showing that, if \(a_1 \neq a_2\) then the definition of Rule-IDom means that they must dominate one another cyclically. By the anti-symmetry of \text{dom}, we immediately derive that \(a_1 = a_2\).

It is now known that \textit{idom} can form the basis for a function – it must now be checked that a function derived from \textit{idom} will be a total function for all addresses in its object graph. Therefore, we show that every address has an immediate dominator:

**Theorem 8** \(\forall a \in \text{reachable}(\langle H, a_r \rangle)[\exists a'[\langle H, a_r \rangle \vdash a' \text{idom } a]]\)

**Proof sketch:**

Clearly, if \(a = a_r\), then \(a_r \text{idom } a\) by Rule-IDOMRoot.

Otherwise, \(a_r \text{dom } a\) by Theorem 4, so if there are no other dominators for \(a, a_r \text{idom } a\). Where there are other dominators, this may be refined, but this is sufficient to show that \textit{idom} will specify an immediate dominator for every address in a well-formed object graph.

From Theorems 7 and 8, we know that \textit{idom} is suitable for use as a function. We do this formally, here:

---

Final Year Project
Chapter 7. Ownership in Object Graphs

**Definition 10** The immediate dominators function for graph $G$, $\text{dominators}_G$, is defined thus:

$$\text{dominators}_G : \text{reachable}(G) \rightarrow \text{reachable}(G)$$

$$\text{dominators}_G(a) \overset{\text{def}}{=} a' \text{ where } G \vdash_{\text{og}} a' \text{idom } a$$

Our expectation of this function is that it should at least be a dominator function.

**Theorem 9** $\text{dominators}_G$ is a dominator function for extended object graph $G$: $G \vdash \text{dominators}_G \diamond \text{dom}$.

**Proof:**

The proof follows obviously from the definition of idom.

1. $\text{dominators}_G(a) = a'$ (Assumption)
2. $G \vdash_{\text{og}} a' \text{idom } a$ (1, Defn $\text{dominators}_G$)
3. $G \vdash_{\text{og}} a' \text{ dom } a$ (2, Rule-IDom)
4. $G \vdash_{\text{og}} \text{dominators}_G(a) \text{ dom } a$ (1, 3)
5. $\forall a \in \text{reachable}[G \vdash_{\text{og}} \text{dominators}_G(a) \text{ dom } a]$ (4, $a$ arbitrary, $\forall$-introduction)
6. $G \vdash \text{dominators}_G \diamond \text{dom}$ (5, Rule-DomFunc)

Thus, $\text{dominators}_G$ is always a dominator function for $G$, which was to be shown.  

**7.5 Dominators as Owners, Revisited**

If $\text{dominators}_G$, the function based on the immediate dominators of addresses in $G$, is to be used as an ownership function then it must be shown that $\text{dominators}_G$ is an ownership function – particularly that it does not suffer from the difficulties shown in Section 7.1.

Before we embark on this proof, three lemmata are required. The first partially characterises idom in terms of paths in the object graphs and states that, on a path to address $a$, the final dominator of $a$ on that path must immediately dominate $a$:

**Lemma 4** Suppose that $G \vdash a_1 \text{ dom } a_2$, $a_1 \neq a_2$, and that on a path from $a_r$ to $a_2$ that there was no other dominator of $a_2$ after the necessary occurrence of $a_1$ on that path (as it is a dominator). Then, we can conclude that $G \vdash_{\text{og}} a_1 \text{idom } a_2$.

**Proof:**
Inferring Ownership

1: \( a_1 \neq a_2 \)  
(Assumption)
2: \( \Gamma \vdash a_1 \text{dom} a_2 \)  
(Assumption)
3: \( \Gamma \vdash \text{path} \, (p_1 : a_1) \ldots p_2 \, \text{from} \, a_r \, \text{to} \, a_2 \)  
(Assumption)
4: \( \neg \exists a_4 (\Gamma \vdash \text{path} \, p_2 \, \text{via} \, a_4 \land \Gamma \vdash \text{og} \, a_4 \, \text{dom} \, a_2) \)  
(Assumption)
5: \( p_2 \neq [] \)  
(1, 3)

To show that \( \Gamma \vdash \text{og} \, a_1 \, \text{idom} \, a_2 \), it only remains to be shown (according to Rule-IDOM) that \( \Gamma \vdash \text{og} \, a_3 \, \text{dom} \, a_2 \Rightarrow \Gamma \vdash \text{og} \, a_3 \, \text{dom} \, a_1 \). Suppose, then, that there is some \( a_3 \) such that \( \Gamma \vdash \text{og} \, a_3 \, \text{dom} \, a_2 \), but that \( \Gamma \not\vdash \text{og} \, a_3 \, \text{dom} \, a_1 \). To complete the definition of \text{idom}, a contradiction must be shown.

6: \( \Gamma \vdash \text{og} \, a_3 \, \text{dom} \, a_2 \)  
(Assumption)
7: \( \Gamma \not\vdash \text{og} \, a_3 \, \text{dom} \, a_1 \)  
(Assumption)
8: \( \exists p \, (\Gamma \vdash \text{path} \, p \, \text{from} \, a_r \, \text{to} \, a_1 \land \Gamma \not\vdash \text{path} \, p \, \text{via} \, a_3) \)  
(7, Rule-Dom)
9: \( \Gamma \vdash \text{path} \, p_3 \, \text{from} \, a_r \, \text{to} \, a_1 \)  
(8, \exists\text{-elimination)
10: \( \Gamma \not\vdash \text{path} \, p_3 \, \text{via} \, a_3 \)  
(8, Rule-Dom, \exists\text{-elimination)
11: \( \Gamma \vdash \text{path} \, p_3 \ldots p_2 \, \text{from} \, a_r \)  
(9, Corollary 1)
12: \( \Gamma \vdash \text{path} \, p_3 \ldots p_2 \, \text{from} \, a_r \, \text{to} \, a_2 \)  
(3, Rule-PathTo)
13: \( \Gamma \not\vdash \text{path} \, p_2 \, \text{via} \, a_3 \)  
(6, Instance of 3)
14: \( \Gamma \not\vdash \text{path} \, p_3 \ldots p_2 \, \text{via} \, a_3 \)  
(10, 13, Rule-Via)
15: Contradiction  
(6, 12, 14, Rule-Dom)

Thus, we conclude that \( \Gamma \vdash \text{og} \, a_3 \, \text{dom} \, a_1 \) and, therefore, that \( \Gamma \vdash \text{og} \, a_3 \, \text{dom} \, a_2 \Rightarrow \Gamma \vdash \text{og} \, a_3 \, \text{dom} \, a_1 \). This, and the fact that \( a_1 \neq a_2 \) and that \( \Gamma \vdash \text{og} \, a_1 \, \text{dom} \, a_2 \), is sufficient to show that \( \Gamma \vdash \text{og} \, a_1 \, \text{idom} \, a_2 \), which was to be shown. \( \square \)

Explanation of the proof: The idea behind the proof is easier to appreciate when only acyclic paths are considered. On a path to \( a_2 \), if \( a_1 \) is the last dominator on the path then all other dominators of \( a_2 \) must dominate \( a_1 \) – they must all be on the path to \( a_2 \), but cannot be after \( a_1 \) on that path as \( a_1 \) is the last dominator. Therefore, as all other dominators of \( a_2 \) dominate \( a_1 \), \( a_1 \text{idom}a_2 \). The extension to cyclic paths is, from here, relatively easy as all cyclic paths are just acyclic paths with an extra loop added. Cyclic paths have no influence on domination.

Notes to the proof: This proof is only required for the proof of Lemma 5, and does not directly play a role in the proof that \text{dominators}_G \text{is an ownership function.}

Next, a lemma is required stating that all dominators of an address \( a_2 \) are covered by some number of applications of \text{dominators}_G. In other words, the transitive closure of the \text{idom} relation returns the original \text{dom} relation. Note that this is the idea that motivated the definition of \text{idom} in the first instance, so this is as important not only to confirm the soundness of our definition, but also to complete later proofs.

**Lemma 5** To be shown: \( \Gamma \vdash \text{og} \, a_1 \, \text{dom} \, a_2 \Rightarrow \exists k [\text{dominators}_G^k(a_2) = a_1] \)
The proof proceeds by induction on the length of paths to $a_2$:

**Proof:**

Suppose that $G ⊢_\text{og} a_1 \text{ dom } a_2$, then it must be shown that $∃k[\text{dominators}_G^k(a_2) = a_1]$:

1. $G ⊢_\text{og} a_1 \text{ dom } a_2$ \hspace{1cm} (Assumption)
2. $G ⊢_\text{og} p \text{ from } a_r \text{ to } a_2$ \hspace{1cm} ($G ⊢ \Diamond_{\text{og}}$)
3. $G ⊢_\text{og} p \text{ via } a_1$ \hspace{1cm} ($1, 2, \text{ Rule-Dom}$)

**Case 1:** $\text{length}(p) = 1$ (Base Case)

Then $a_1 = a_2 = a_r$, so $k = 0$ and the case is complete.

**Case 2:** $\text{length}(p) > 1$ (Inductive Case)

Again, if $a_1 = a_2$, then $k = 0$ and the result is immediate. Suppose, therefore, that $a_1 \neq a_2$.

4. $p = (p_1 : a_1) ++ (p_2 : a_2)$ \hspace{1cm} ($2, 3, a_1 \neq a_2$)

Now take $a_3 \neq a_2$ to be such that $G ⊢_\text{og} a_3 \text{ dom } a_2$ and $G ⊢_{\text{path}} (p_2 : a_2) \text{ via } a_3$. If no such $a_3$ can be found, then $G ⊢_\text{og} a_1 \text{ idom } a_2$ by Lemma 4, and $\text{dominators}_G(a_2) = a_1$ by definition. Thus $k = 1$ and the desired property holds.

If there is such an $a_3$, choose it so that $(p_2 : a_2) = (p_3 : a_3) ++ p_4$, and that there is no dominator of $a_2$ in $p_4$; $p_4 \neq []$ as $a_3 \neq a_2$. Then, $G ⊢_\text{og} a_3 \text{ idom } a_2$ by Lemma 4 and $\text{dominators}_G(a_2) = a_3$. Notice that $\text{length}((p_1 : a_1) ++ (p_3 : a_3)) < \text{length}(p)$, so by the inductive hypothesis $∃j[\text{dominators}_G^j(a_3) = a_1]$.

Therefore, $\text{dominators}_G^j(\text{dominators}_G(a_2)) = a_1$, so $k = j + 1$ and the property holds. \hfill ∎

**Explanation of the proof:** We wish to show that every dominator of an object is returned by some number of iterations. The proof proceeds by induction on the length of paths to $a_2$. If the path is only of length one, then $a_2$ is equal to the graph root, which is its own single dominator, so the proof is finished. If the path is longer, then it can be split into portions $p_1$ and $p_2$, the first of which ends at $a_1$ (as all paths to $a_2$ go via $a_1$).

If there are no dominators between $a_1$ and $a_2$, then $\text{dominators}_G(a_2) = a_1$ immediately by Lemma 4. Otherwise, we can find a dominator closer to $a_2$ on the path such that $\text{dominators}_G(a_2) = a_3$. All the dominators of $a_2$ are dominators of $a_3$ by definition, so there must be some $j$ such that $\text{dominators}_G^j(a_3) = a_1$. The result then follows immediately by substituting in the two equalities.

**Notes to the proof:** Note that this is similar to the spanning-tree theorem for ownership functions (Theorem 5) – $a_r$ is a particular dominator of every address, and from Lemma 5 we know that some number of applications of $\text{dominators}_G$ to every address will reach that address. However, $\text{dominators}_G$’s direct definition in terms of dominators in the object graph makes the proof much more compact.
A final lemma simply relates the dominators of $a_2$ to those of $a_1$ when $G \vdash_{\text{og}} a_1 \text{ref } a_2$. If $a_1$ references $a_2$ then everything that dominates $a_2$ (except $a_2$) also dominates $a_1$:

**Lemma 6** $G \vdash_{\text{og}} a_1 \text{ref } a_2 \land a_3 \neq a_2 \land G \vdash_{\text{og}} a_3 \text{dom } a_2 \Rightarrow G \vdash_{\text{og}} a_3 \text{dom } a_1$

**Proof:**

Suppose the antecedent: $G \vdash_{\text{og}} a_1 \text{ref } a_2 \land a_3 \neq a_2 \land G \vdash_{\text{og}} a_3 \text{dom } a_2$. Then it remains to show that $G \vdash_{\text{og}} a_3 \text{dom } a_1$.

1: $G \vdash_{\text{og}} a_1 \text{ref } a_2$ (Assumption)
2: $a_2 \neq a_3$ (Assumption)
3: $G \vdash_{\text{og}} a_3 \text{dom } a_2$ (Assumption)

If $a_1 = a_3$ then $G \vdash_{\text{og}} a_3 \text{dom } a_1$ follows immediately. So, assume $a_1 \neq a_3$.

4: $G \vdash_{\text{path}} p \text{ from } a_r \text{ to } a_1$ ($G \vdash_{\text{og}} a_1 \text{ref } a_2$)
5: $G \vdash_{\text{path}} p : a_2 \text{ from } a_r \text{ to } a_2$ (1, Rule-PathInd)
6: $G \vdash_{\text{path}} p : a_2 \text{ via } a_3$ (3, Rule-Dom)
7: $G \vdash_{\text{path}} p \text{ via } a_3$ (6, $a_2 \neq a_3$)
8: $G \vdash_{\text{path}} p \text{ from } a_r \text{ to } a_1 \Rightarrow G \vdash_{\text{path}} p \text{ via } a_3$ (4, 7, $\Rightarrow$-introduction)
9: $G \vdash_{\text{og}} a_3 \text{ dom } a_1$ (8, Rule-Dom)

Proof complete. □

The previous three lemmata support the following theorem, which states that $\text{dominators}_G$ is an ownership function for all graphs $G$:

**Theorem 10** $G \vdash \text{dominators}_G \diamond_{\text{own}}$

**Proof of Theorem 10:**

Clearly, $\text{dominators}_G(a_r) = a_r$ by its definition, so to show $G \vdash \text{dominators}_G \diamond_{\text{own}}$, it remains only to show that $G \vdash_{\text{og}} a_1 \text{ ref } a_2 \Rightarrow \exists k \geq 0[\text{dominators}_G^k(a_1) = \text{dominators}_G(a_2)]$

If $a_1 = a_2$, then $k = 1$. If $a_2 = a_r$, then there is a $k$ such that $\text{dominators}_G^k(a_1) = a_r$ by Lemma 6, because $G \vdash a_r \text{ dom } a_r$ by Theorem 4. So, examine the case where $a_2 \neq a_r$ and $a_1 \neq a_2$:

1: $G \vdash_{\text{og}} a_1 \text{ ref } a_2$ (Assumption)
2: $G \vdash \text{dominators}_G \diamond_{\text{dom}}$ (Theorem 9)
3: $G \vdash \text{dominators}_G(a_2) \text{ idom } a_2$ (2, Defn of $\text{dominators}_G$)
4: $\text{dominators}_G(a_2) \neq a_2$ (3, Rule-IDom, $a_2 \neq a_r$)
5: $G \vdash \text{dominators}_G(a_2) \text{ dom } a_2$ (3, Rule-IDom)
Chapter 7. Ownership in Object Graphs

6: \[G \vdash \text{dominators}_G(a_2) \text{ dom } a_1\] (1, 4, 5, Lemma 6)

7: \[\exists k [\text{dominators}_G^k(a_1) = \text{dominators}_G(a_2)]\] (6, Lemma 5)

Therefore, conclude that \(\text{dominators}_G\) is a valid ownership function for all \(G\). \(\square\)

**Explanation of the proof:** The definition of \(\text{dominators}_G\) implies that \(\text{dominators}_G(a_2)\) dominates \(a_2\). As \(a_1\) references \(a_2\), Lemma 6 states \(a_1\) inherits all of \(a_2\)'s dominators, so that \(\text{dominators}_G(a_2) \text{ dom } a_1\). Lemma 5 then states that all the dominators of \(a_1\) can be accessed by some number, \(k\), of applications of \(\text{dominators}_G\) such that \(\text{dominators}_G^k(a_1) = \text{dominators}_G(a_2)\), which is precisely what is required by Rule-OwnFunc. Therefore, \(\text{dominators}_G\) is an ownership function for all graphs \(G\).

**Notes to the proof:** This proof makes the earlier proof that \(G \vdash \text{dominators}_G \triangleleft \text{dom} \) redundant (Theorem 9), as we know more generally that \(G \vdash f \triangleleft \text{own} \Rightarrow G \vdash f \triangleleft \text{dom} \) (Theorem 6). The use of \(G \vdash \text{dominators}_G \triangleleft \text{dom} \) in the proof above is unnecessary – the same effect can be obtained simply by virtue of the definition of \(\text{idom}\) in \(G \vdash \text{dominators}_G(a)\) \(\text{idom}\) \(a\).

We now have a candidate for deriving an ownership from object graphs. However, there are many possible ownership functions for any graph in addition to \(\text{dominators}_G\). In the next section, we characterise the whole space of ownership functions and determine \(\text{dominators}_G\)'s relative precision.

### 7.6 Precision of Ownership Functions

In general, there are many ownership functions that will be valid on any extended object graph. Therefore, we define the set of ownership functions valid on a graph:

**Definition 11** \(\text{ownerships}_G = \{O \mid G \vdash O \triangleleft \text{own} \}\)

This set is clearly finite – there will be at most around the order of \(|\text{dom}(G)|^2\) functions in \(\text{ownerships}_G\), reflecting all the different ownerships that could be assigned in \(G\).

We have already shown in Theorem 10 that \(\text{dominators}_G \in \text{ownerships}_G\), so we also know that \(\text{ownerships}_G\) is never empty. Additionally, root-owns-all is always a valid ownership.

For completeness, then, we define the root-owns-all, \(\rho_G\), ownership function of an ownership graph \(G\):

**Definition 12** \(\forall a \in \text{reachable}(G) [\rho_{(H,a_r)}(a_r) = a_r]\)

\(\rho_G\) is clearly an ownership function: \(\forall k \geq 1 [\rho_G^k(a_1) = \rho_G(a_2)]\), regardless of whether \(G \vdash_{\text{og}} a_1 \text{ ref } a_2\). Therefore, \(\rho_G \in \text{ownerships}_G\). Of course, in the degenerate case, \(\text{dominators}_G = \rho_G\).
Inferring Ownership

Figure 7.4: Two ownership functions, $O_1$ and $O_2$, for snapshot $G_B$ one of which is dominators$_G$. Notice that $O_2(C) = A = O_1(O_1(C))$. Similarly for all the other addresses, a single application of $O_2$ is equivalent to multiple applications of $O_1$. Therefore, $O_1 \sqsubseteq O_2$.

To characterise this space of ownership functions, an ordering is defined:

\[
\frac{\forall a \in \text{dom}(O_2) [\exists k [O_1^k(a) = O_2(a)]]}{O_1 \sqsubseteq O_2}
\]

$O_1 \sqsubseteq O_2$ is read “$O_1$ is more precise than $O_2$” – this follows the convention that the left-hand side of $\sqsubseteq$ has more information than that on the right. According to Rule-OwnPrec, $O_1 \sqsubseteq O_2$ if and only if several applications of $O_1$ from address $a$ reaches the same address as a single application of $O_2$ to $a$.

Note that $\sqsubseteq$ implies little about whether or not an ownership function is valid on a graph: $G \vdash O_1 \diamondown \land O_1 \sqsubseteq O_2 \neq G \vdash O_2 \diamondown$. It should be clear that, if $O_2 \sqsubseteq O_1$, then it may not be that $G \vdash O_2 \diamondown$: $O_2$ may have been made over-precise.

### 7.6.1 Alternative Orderings of Ownership

For completeness, we examine some other potential orderings for the space of ownership functions.

\[
\frac{\exists k [\forall a [O_1^k(a) = O_2(a)]]}{O_1 \sqsubseteq^+ O_2}
\]
The first rule, `Rule-OwnPrec+`, specifies that $O_1 \sqsubseteq^+ O_2$ if a single application of $O_2$ to $a$ is equivalent to the same number of applications of $O_1$ for all addresses. Clearly, $O_1 \sqsubseteq^+ O_2 \Rightarrow O_1 \sqsubseteq O_2$. However, $\sqsubseteq^+$ is an undesirable ordering as it does not order enough of the ownership functions – we would intuitively expect the graphs of Figure 7.4 to be ordered relative to one another, but this is not the case with $\sqsubseteq^+$.

The following is another possible ordering for ownership functions:

$$(\text{Rule-OwnPrec}^-) \quad \forall a[\exists k_1, k_2[O_1^{k_1}(a) = O_2^{k_2}(a) \land k_1 \geq k_2]]$$

Again, $O_1 \sqsubseteq O_2 \Rightarrow O_1 \sqsubseteq^- O_2$. However, $\sqsubseteq^-$ is not an ordering, as it is not anti-symmetric:

**Corollary 4** $\sqsubseteq^-$ is not an ordering: For every $O_1, O_2$ such that $G \vdash O_1 \diamond \text{own}$ and $G \vdash O_2 \diamond \text{own}$, $O_1 \sqsubseteq^- O_2$ and $O_1 \sqsupseteq^- O_2$.

The corollary follows from Lemma 5 – both $O_1$ and $O_2$ can be applied $k$ and $j$ times respectively so that $O_1^k(a) = a = O_2^j(a)$. Depending on whether $O_1 \sqsubseteq^- O_2$ or $O_1 \sqsupseteq^- O_2$ is desired, $k$ or $j$ can be increased without modifying the fact that $O_1^k(a) = a = O_2^j(a)$ by Corollary 3. Thus, a $k$ and $j$ can be found such that $k \geq j$ or $j \geq k$, satisfying the conditions for $\sqsubseteq^-$. Therefore, the excessive restriction of $\sqsubseteq^+$ and the poor behaviour of $\sqsubseteq^-$ makes $\sqsubseteq$ the appropriate choice of ordering for a graph’s ownership functions.

### 7.6.2 The Properties of $\sqsubseteq$

Despite having rejected $\sqsubseteq^+$ and $\sqsubseteq^-$ it still remains to be checked that $\sqsubseteq$ is an ordering. We now show that $\sqsubseteq$ is a partial order on $\text{ownerships}_G$ for all graphs, $G$ – we take the familiar route of showing that $\sqsubseteq$ is anti-symmetric, reflexive and transitive.

First, the anti-symmetry of $\sqsubseteq$ is shown. Strictly, when we say $O_1 = O_2$, this is an abbreviation for $\forall a[O_1(a) = O_2(a)]$:

**Theorem 11** $\forall O_1, O_2 \in \text{ownerships}_G[O_1 \sqsubseteq O_2 \land O_2 \sqsubseteq O_1 \Rightarrow O_1 = O_2]$

**Proof:**

Suppose, for arbitrary address $a$, that $O_1 \sqsubseteq O_2$ and $O_2 \sqsubseteq O_1$. It is then to be shown that $O_1(a) = O_2(a)$ for all such $a$: 
Inferring Ownership

1: \( O_2^j(a) = O_1(a) \) \( (O_2 \sqsubseteq O_1) \)
2: \( O_1^k(a) = O_2(a) \) \( (O_1 \sqsubseteq O_2) \)
3: \( G \vdash_{og} O_2^j(a) \) dom \( O_2(a) \) \( (j \geq 1, G \vdash O_2 \diamond_{dom}) \)
4: \( G \vdash_{og} O_1^k(a) \) dom \( O_1(a) \) \( (k \geq 1, G \vdash O_1 \diamond_{dom}) \)
5: \( G \vdash_{og} O_1(a) \) dom \( O_2(a) \) \( (1, 3) \)
6: \( G \vdash_{og} O_2(a) \) dom \( O_1(a) \) \( (2, 4) \)
7: \( O_2(a) = O_1(a) \) \( (\text{dom is anti-symmetric}) \)

Conclude that \( \sqsubseteq \) is anti-symmetric.

Explanation of the proof: This proof requires little explanation – we make good use of the fact that both \( O_1 \) and \( O_2 \) must be dominator functions for \( G \) as well as ownership functions. We then appeal to the properties of \( \text{dom} \) as a partial order on addresses to achieve the desired result.

We now check that \( \sqsubseteq \) is transitive and reflexive – these results follow immediately and require no explanation:

**Theorem 12** \( \sqsubseteq \) is transitive on ownership functions: \( O_1 \sqsubseteq O_2 \land O_2 \sqsubseteq O_3 \Rightarrow O_1 \sqsubseteq O_3 \).

**Proof:**

For every address, \( a \), in graph \( G \), there is a \( j \) such that \( O_2^j(a) = O_3(a) \). As \( O_2 \) is a spanning tree for the addresses in \( G \) (Theorem 5), then \( O_2^j(a) \) is in \( G \), so there is a \( k \) such that \( O_1^k(a) = O_2^j(a) = O_3(a) \). This satisfies the conditions for \( O_1 \sqsubseteq O_3 \).

**Theorem 13** \( \sqsubseteq \) is reflexive. \( \forall O [O \sqsubseteq O] \).

**Proof:**

This is immediately clear from the definition of \( \sqsubseteq \), where \( k = 1 \).

From \( \sqsubseteq \)'s anti-symmetry, transitivity and reflexivity, we find the following:

**Corollary 5** \( \sqsubseteq \) is a partial order on the set \( \{ O \mid G \vdash O \diamond_{own} \} \), or the set of ownership functions valid for an object graph \( G \).

The corollary follows immediately from Theorems 11, 12 and 13.

### 7.6.3 Characterising dominators\(_G\)

With the \( \sqsubseteq \) ordering, we are now able to place dominators\(_G\) relative to all the other ownership functions valid on \( G \). Moreover, dominators\(_G\) is the most precise ownership function for \( G \) – a very important result:
Chapter 7. Ownership in Object Graphs

**Theorem 14** $\text{dominators}_G$ is the most $\sqsubseteq$-precise ownership function for object graph $G$: $G \vdash O \diamond \text{own} \Rightarrow \text{dominators}_G \sqsubseteq O$

**Proof:** By contradiction

Take arbitrary address $a$:

1. $G \vdash O \diamond \text{own}$ (Assumption)
2. $G \vdash \text{og} \text{dominators}_G^k(a) \text{ dom } a$ (Defn of $\text{dominators}_G$, Transitivity of $\text{dom}$)
3. $G \vdash O \diamond \text{dom}$ (1, Theorem 6)
4. $G \vdash \text{og} O(a) \text{ dom } a$ (3, Rule-DomFunc)
5. $\exists j[\text{dominators}_G^j(a) = O(a)]$ (4, Lemma 5)
6. $\text{dominators}_G \sqsubseteq O$ (5, arbitrary $a$, Rule-OwnPrec)

Conclude, therefore, that $\text{dominators}_G \sqsubseteq O$, which was to be shown. □

**Explanation of the proof:** All other ownership functions are dominator functions, but $\text{dominators}_G$ is defined using the most precise dominators. Therefore, all other ownership functions are less precise than $\text{dominators}_G$.

**Notes to the proof:** The elegance of this proof, which shows one of the most important results of our study of object ownership, justifies the various iterations and re-writes performed when designing the earlier formalism. Of course, it could be argued that this ordering is designed to give the result but in the next section we describe the other potential orderings of ownership functions, and show that they are unhelpful.

In fact, with Theorem 14 and the fact that $\sqsubseteq$ is a partial order, we obtain the following; a complete characterisation of $\text{dominators}_G$ in the space of ownership functions for $G$, $\text{ownerships}_G$:

**Corollary 6** $\text{dominators}_G$ is the unique most-$\sqsubseteq$-precise ownership function – it is the unique $\sqsubseteq$-upper bound of $\text{ownerships}_G$.

This follows by virtue of Theorem 14, stating that $\text{dominators}_G \sqsubseteq O$ for all ownership functions $O$ of graph $G$. If there were another $O\text{min}$ such that $\forall O[O\text{min} \sqsubseteq O]$, then $O\text{min} \sqsubseteq \text{dominators}_G$ and $\text{dominators}_G \sqsubseteq O\text{min}$, so by Theorem 11, $O\text{min} = \text{dominators}_G$.

To balance the corollary above, we show a similar result for the root-owns-all function, $\rho_G$:

**Theorem 15** $\rho_G$ is the unique, least-$\sqsubseteq$-precise ownership function for object graph $G$ – it is the unique $\sqsubseteq$-lower bound of $\text{ownerships}_G$.

**Proof:**

Alisdair Wren 91
Inferring Ownership

According to the reasoning following Definition 12, we already know that $\rho_G \in \text{ownerships}_G$.

Suppose for some other $O$ that $G \vdash O \diamondown$, then Theorem 5 states that $O$ is a spanning tree, so for every address in $G$, there is a $k$ such that $O^k(a) = a_r$. Thus, for every such ownership function, $\forall a[\exists k[O^k(a) = \rho_G(a)]]$, by definition of $\rho_G$, and so $O \subseteq \rho_G$. Thus $G \vdash O \diamondown \Rightarrow O \subseteq \rho_G$, so $\rho_G$ is an upper bound of the ownership functions of $G$ under the $\sqsubseteq$.

It now remains to show that $\rho_G$ is the only ownership function that is an upper bound of $\text{ownerships}_G$: Suppose there was some other $O_{\text{max}}$ such that $G \vdash O_{\text{max}} \Rightarrow O \sqsubseteq O_{\text{max}}$, then $\rho_G \sqsubseteq O_{\text{max}}$ and $O_{\text{max}} \sqsubseteq \rho_G$, which by $\sqsubseteq$’s anti-symmetry shows that any such $O_{\text{max}} = \rho_G$.

Thus, $\rho_G$ is the unique upper bound of the ownership functions of $G$. 

7.7 Ownership under Graph Combination

It has already been observed that domination can be problematic when used as ownership when graphs are combined (Section 7.1). However, ownership is well behaved under combination and, indeed, under separation: An ownership function for a union of graphs is an ownership function for all of the constituent graphs. Furthermore, if an ownership function is valid on a set of graphs then it is also valid on their union:

**Theorem 16** \( G_1 \vdash O \diamondown \land G_2 \vdash O \diamondown \Leftrightarrow (G_1 \cup_{\text{xog}} G_2) \vdash O \diamondown \)

The proof relies on the fact that the result of graph union has exactly those references found in the constituent graphs:

**Proof:**

If $G \vdash O \diamondown$ for any $G$, then $O(a_r) = a_r$, so we need only show that $G \vdash_{\text{og}} a_1 \ref a_2 \Rightarrow \exists k[O^k(a_1) = O(a_2)]$ in both directions of implication.

(\(\Leftarrow\)): If $G_1 \vdash_{\text{og}} a_1 \ref a_2$ then $(G_1 \cup_{\text{xog}} G_2) \vdash_{\text{og}} a_1 \ref a_2$, as references are preserved through graph combination. As $(G_1 \cup_{\text{xog}} G_2) \vdash O \diamondown$, then $\exists k[O^k(a_1) = O(a_2)]$, which was to be shown.

(\(\Rightarrow\)): If $(G_1 \cup_{\text{xog}} G_2) \vdash a_1 \ref a_2$, then either $G_1 \vdash_{\text{og}} a_1 \ref a_2$ or $G_2 \vdash_{\text{og}} a_1 \ref a_2$. As $G_1 \vdash O \diamondown$ and $G_2 \vdash O \diamondown$, then $\exists k[O^k(a_1) = O(a_2)]$, which was to be shown. 

Because $\bigcup_{\text{xog}}$ is associative, this result generalises to the combination of sets of graphs, \( \{G_1, G_2, \ldots, G_n\} \), $\bigcup_{\text{xog}}\{G_i\}$.

This property is particularly important, as it allows us to focus on ownership functions for single extended object graphs, safe in the knowledge that our efforts will extend automatically to the program heaps from which they are constructed. It also suggests a means
by which the problem of finding ownership can be decomposed into smaller problems, and provides a suggestion for how to update ownerships when a new reference is added to an object graph – the graph union is, after all, simply the addition of groups of references and new objects.

We complete this section of the formalism with the most important result when inferring ownership invariants from object graphs:

**Corollary 7** dominators_{G_1 \cup_{xog} G_2} for the union of graphs G_1 and G_2 is the most precise ownership function that is an ownership function for both G_1 and G_2.

Corollary 7 holds because dominators_{G} is the most precise ownership function for G_1 \cup_{xog} G_2, so by Theorem 16, it is the most precise ownership function that is such for both G_1 and G_2.

Thus, we conclude that we can find no better ownership function for a set of n object graphs – each representing an object heap – than dominators_{G} for graph G = \bigcup_{xog} \{G_1, G_2, \ldots, G_n\}.

### 7.7.1 Monotonicity of Ownership under Graph Combination

We earlier discussed how domination does not behave monotonically under combination of extended object graphs. However, the result of Theorem 16 is that every ownership valid on the union of two graphs, G_1 \cup_{xog} G_2, is also valid on G_1 and on G_2. Thus, the set of ownership functions valid on a graph, ownerships_{G}, always shrinks under graph combination. Of course, as objects are added to the object graph, the domains and ranges of the functions in ownerships_{G} must also increase.

### 7.8 Summary

In this chapter, we have examined the possibilities for calculating the most precise ownership possible for a set of program heaps. We showed how it is both sufficient and necessary to find an ownership for a combination (with \cup_{xog}) of program heaps in order to find a valid ownership that is invariant across those individual heaps. An ordering on all the potential ownerships for an extended object graph was given and compared with other potential orderings. The dominators_{G} function was defined based on the ‘immediate’ dominators in the object graph and we showed that it was the unique, most precise ownership function for an extended object graph. Combined with earlier results, dominators_{G} for a union of graphs, G, is the unique most precise ownership that is invariant across all of the individual graphs combined to produce G.

This chapter concludes the formal system constructed in detail – these portions provide a sound basis on which to consider the applications of object graph ownership inference.
Inferring Ownership

In the following two chapters, we examine the possibilities for the inference of ownership types.
Chapter 8

Adapting Ownerships

Ownerships that may be valid on object heaps may not be valid according to the ownership type system. We need, therefore, to modify the ownership we obtain so it can be consistent with an ownership typed program. In this chapter, we introduce such a mechanism, and show how it can be used to satisfy several necessary constraints on type-consistent ownership types.

8.1 Why Adapt Ownerships?

The ownerships inferred from dynamic snapshots of the program heap during execution will never be a complete representation of the behaviour we might expect from an ownership-typed program. Figure 8.1 shows an example of a constraint imposed by the static system upon the validity of an inferred dynamic ownership. We find that that we cannot type field \( f \) of \( X \)'s class, as it can hold object references pointing to both its own and to an external context. In order to bring the dynamic ownership into consistency with the type system, \( Y \) must be moved outside of \( X \)'s context. While it might seem that \( Z \) could be brought inside, there may be other references to \( Z \) that would cause its movement to break the ownership invariant. This exemplifies the pressures imposed by our need for ownership precision conflicting with those imposed by compliance with the ownership type system.

Therefore, given a set of ownerships, such as might be inferred from the system given in the previous chapters, and a set of constraints, such as might be imposed by the circumstances in Figure 8.1, we must have a means for adjusting the ownerships so that they satisfy those constraints.

Alisdair Wren
Figure 8.1: If field \( f \) of object \( X \) references an object inside its context, then that field must be given the \textit{this} ownership parameter. However, \( f \) also references an object outside its own context, so that field must be typed with an ownership parameter. This contradiction can be alleviated by relaxing the ownership of \( Y \) so that it is moved outside of \( X \)’s context.

8.2 Introducing \( \text{relax}_G \)

We therefore define a “relaxation” function, so called because it weakens the ownership constraints in order to satisfy constraints imposed by the type system:

The result of \( \text{relax}_G(O, a) \) is intended to be a valid ownership function in which \( a \) has been “raised” in the ownership graph. To keep the ownership valid, \( \text{relax}_G \) also moves other addresses upwards in the tree of ownership contexts if \( a \)’s raising would break the ownership invariant. Figure 8.2 shows the different aspects of the \( \text{relax}_G \) definition. \( \text{siblings}_G(O, a) \) gives the set of objects in the same context as the promoted address, \( a \). \( \text{belowpromoted}_G(O, a) \) gives the set of objects in any context inside that of the promoted address. The set \( \text{promoted}_G(O, a) \) gives the set of objects which must be moved upwards if \( a \) is promoted, which may include more than just \( a \). There are two circumstances in which extra objects are added to the list of objects to be promoted: Firstly, if an object is to be promoted and it references a sibling, then that sibling must also be promoted (see Figure 8.3). Otherwise, when the promoted object (C) is moved up into the enclosing context, it no longer has access to the referenced sibling, D, according to the ownership invariant, so the reference becomes illegal.

Secondly, if an object in a context below a promoted address, for example E, references a sibling, D, of the promoted address, C, then the sibling must be promoted so that the reference still passes only upwards through the ownership tree.

The definition of \( \text{promoted}_G \) is recursive – the addition of a new node according the the two criteria above may result in yet more nodes added. Notice, however, that it is only siblings
Figure 8.2: The diagram above shows the different parts of the $\text{relax}_G$ definition. $C$ is the address to be relaxed, so it is automatically in $\text{promoted}_G$. $E$ is in a context below $C$, and is thus in $\text{belowpromoted}_G$. $D$ is in the same context as $C$, and is therefore in $\text{siblings}_G$. In this case, $D$ does not have to be raised, but in the general case any number of siblings may also need to be raised to keep the ownership consistent.

**Definition 13** $\text{relax}_G$ and its auxiliary functions are defined as follows:

$siblings_G : \text{XObjectGraph} \rightarrow \text{OwnershipFunction} \rightarrow \text{Address} \rightarrow \mathcal{P}(\text{Address})$

$siblings_G(O,a) \overset{\text{def}}{=} \{a_1 \mid O(a) = O(a_1)\}$

$\text{belowpromoted}_G : \text{XObjectGraph} \rightarrow \text{OwnershipFunction} \rightarrow \text{Address} \rightarrow \mathcal{P}(\text{Address})$

$\text{belowpromoted}_G(O,a) \overset{\text{def}}{=} \{a_1 \mid \exists k \geq 1 [O^k(a_1) = a_2 \land a_2 \in \text{promoted}_G(O,a)]\}$

$\text{promoted}_G : \text{XObjectGraph} \rightarrow \text{OwnershipFunction} \rightarrow \text{Address} \rightarrow \mathcal{P}(\text{Address})$

$\text{promoted}_G(O,a) \overset{\text{def}}{=} \{a\} \cup \{a_2 \mid G \vdash a_1 \text{ ref } a_2 \land a_1 \in \text{promoted}_G(a) \land a_2 \in \text{siblings}_G(O,a)\}$

$\cup \{a_4 \mid G \vdash a_3 \text{ ref } a_4 \land a_3 \in \text{belowpromoted}_G(a) \land a_4 \in \text{siblings}_G(O,a)\}$

$\text{relax}_G(O,a) : \text{XObjectGraph} \rightarrow \text{OwnershipFunction} \rightarrow \text{Address} \rightarrow \text{OwnershipFunction}$

$\text{relax}_G(O,a)(a') \overset{\text{def}}{=} \begin{cases} O(O(a)) & \text{if } a' \in \text{promoted}_G(O,a) \\ O(a') & \text{otherwise} \end{cases}$
Figure 8.3: If we wish to promote $C$ in the left ownership graph, then with the reference from $C$ (in $\text{promoted}_G$) to $D$ (in $\text{siblings}_G$), the resulting graph on the right breaks the ownership invariant. Therefore, whenever a reference comes from an object in $\text{promoted}_G$ to an object in $\text{siblings}_G$, that sibling must also be promoted.
Figure 8.4: If we wish to promote C in the graph on the left, then with the reference from E (in belowpromoted$_G$) to D (in siblings$_G$), the resulting graph on the right breaks the ownership invariant. Therefore, whenever a reference comes from an object in belowpromoted$_G$ to an object in siblings$_G$, that sibling must also be promoted.
Inferring Ownership

of the promoted address, $a$, which are ever promoted alongside.

8.2.1 Properties of $\text{relax}_G$

We have a number of requirements for $\text{relax}_G$ if it is to be useful. Firstly, its output must be a valid ownership for the object graph $G$. Secondly, its output ownership function must be less precise than its input – if this were not the case, we can hardly expect to satisfy more contraints from the type system. Finally, we would prefer that the relaxation was atomic in some way: Given that $a$ must be promoted, do we make the minimum transformation necessary to achieve that promotion whilst still maintaining the ownership invariant?

The most important of these properties is clearly that the result of $\text{relax}_G(O,a)$ is an ownership function. On the way to this result, however, we make use of two lemmata. Firstly, a lemma which shows that the ownership structure of the tree excluding the $\text{promoted}_G$ and $\text{belowpromoted}_G$ parts is maintained precisely as it was before the application of $\text{relax}_G$:

Lemma 7 $a' \notin \text{promoted}_G(O,a) \land a' \notin \text{belowpromoted}_G(O,a) \Rightarrow \forall k[\text{relax}_G(O,a)^k(a') = O^k(a')]$

Proof: By induction

Case 1: $k = 0$

Result is immediate.

Case 2: $k = n + 1$

By definition of $\text{relax}_G$, $\text{relax}_G(O,a)(a') = O(a')$. By definition of $\text{belowpromoted}_G$, there is no $k$ such that $O^k(a') \in \text{promoted}_G(O,a)$ and, consequently, no $k$ such that $O^k(O(a')) \in \text{promoted}_G(O,a)$. Therefore, $\text{relax}_G(O,a)(a') \notin \text{promoted}_G(O,a)$ and $\text{relax}_G(O,a)(a') \notin \text{belowpromoted}_G(O,a)$.

By induction, therefore, $\text{relax}_G(O,a)^n(\text{relax}_G(O,a)(a')) = O^n(O(a'))$, so $\text{relax}_G(O,a)^{n+1}(a') = O^{n+1}(a')$, which was to be shown

Next, a lemma to show a similar result, but that the ownership is preserved inside the $\text{belowpromoted}_G$ portion. $\text{belowpromoted}_G$ is clearly bounded at the top by elements in $\text{promoted}_G$ where, according to the definition of $\text{relax}_G$, the ownership structure is modified. The proof is slightly more complicated than for Lemma 7 in order to deal with this upper bound.

Lemma 8 $a' \in \text{belowpromoted}_G(O,a) \Rightarrow \forall k[O^k(a') \in \text{belowpromoted}_G(O,a) \Rightarrow \text{relax}_G(O,a)^k(a') = O^k(a')]$

Proof: By induction
Suppose \( a' \in \text{belowpromoted}_G(O, a) \).

**Case 1:** \( k = 0 \) (Base Case)

\[ \text{relax}_G(O, a)^0(a') = O^0(a') \]

holds trivially.

**Case 2:** \( k = n + 1 \) (Inductive Case)

Suppose \( O^{n+1}(a') \in \text{belowpromoted}_G(O, a) \), then it must be shown that \( \text{relax}_G(O, a)^{n+1}(a') = O^k(a') \):

As \( a' \in \text{belowpromoted}_G(O, a) \), then \( a' \notin \text{siblings}_G(O, a) \) and \( a' \notin \text{promoted}_G(O, a) \). Therefore, \( \text{relax}_G(O, a)(a') = O(a') \) by the definition of \( \text{relax}_G \), and, as \( 1 \leq n + 1 \), then \( \text{relax}_G(O, a)(a') \in \text{belowpromoted}_G(O, a) \). By induction:

\[
\begin{align*}
\text{relax}_G(O, a)^n(a') &= O^n(\text{relax}_G(O, a)(a')) \\
&= \text{relax}_G(O, a)^n(\text{relax}_G(O, a)(a'))
\end{align*}
\]

so \( \text{relax}_G(O, a)^{n+1}(a') = O(a') \).

Proof complete.

We now show the main part of the theorem in the following lemma, that the ownership invariant holds in the result of \( \text{relax}_G(O, a) \) when \( O \) is a valid ownership function on \( G \):

**Lemma 9** \( G \vdash O \bowtie_{\text{own}} \land G \vdash a_1 \text{ ref } a_2 \Rightarrow \exists j[\text{relax}_G(O, a)^j(a_1) = \text{relax}_G(O, a)(a_2)] \)

**Proof:**

Suppose that \( G \vdash a_1 \text{ ref } a_2 \). Then it must be shown that \( \exists j[\text{relax}_G(O, a)^j(a_1) = \text{relax}_G(O, a)(a_2)] \).

As \( G \vdash O \bowtie_{\text{own}} \), there must be some \( k \) such that \( O^k(a_1) = O(a_2) \).

Notice that the definition of \( \text{relax}_G \) partitions the object graph into three disjoint (except for root address \( a_r \) portions; \( \text{promoted}_G(O, a) \), \( \text{belowpromoted}_G(O, a) \) and all other addresses not in \( \text{promoted}_G(O, a) \cup \text{belowpromoted}_G(O, a) \).

**Case 1:** \( a_1, a_2 \in \text{promoted}_G(O, a) \)

By definition of \( \text{promoted}_G \), \( a_1, a_2 \in \text{siblings}_G \) so \( O(a_1) = O(a_2) = O(a) \). Therefore, \( O(O(a_1)) = O(O(a_2)) \) and, applying the definition of \( \text{relax}_G \), \( \text{relax}_G(O, a)(a_1) = \text{relax}_G(O, a)(a_2) \) as required.

**Case 2:** \( a_1 \in \text{promoted}_G(O, a), a_2 \in \text{belowpromoted}_G(O, a) \)

By definition of \( \text{belowpromoted}_G \) for \( a_2 \), \( O^m(a_2) \in \text{promoted}_G(O, a) \), so \( O^m(a_2) \in \text{siblings}_G(O, a) \). By definition of \( \text{promoted}_G \) for \( a_1, a_1 \in Siblings(O, a) \), so the definition of \( \text{siblings}_G \) implies that \( O^m + 1(a_2) = O(a_1) \).

Recalling that \( O^k(a_1) = O(a_2) \) for some \( k \):
Inferring Ownership

- If \( a_1 = a_r \) then \( \text{relax}_G(O, a)(a_2) = O(a_2) \) by definition of \( \text{relax}_G \). As \( G \vdash O \leadsto_{\text{own}} \), \( O^k(a_1) = a_r \). Therefore, \( a_1 = \text{relax}_G(O, a)^0(a_1) = \text{relax}_G(O, a)(a_2) \).

- If \( a_2 = a_r \) then \( O(a_1) = O^{m+1}(a_2) = a_r \). As \( a_r \) is a fixed-point of \( O \), \( a_r = O(a_1) = O(a_2) \). By definition of \( \text{relax}_G \) for both \( a_1 \) and \( a_2 \), \( \text{relax}_G(O, a)(a_1) = \text{relax}_G(O, a)(a_2) \) as required.

- If \( a_1 \neq a_r \) and \( a_2 \neq a_r \), then \( a_1 \) and \( a_2 \) are not fixed points of \( O \). Notice that \( O(a_2) = a_1 \in \text{promoted}_G \), so \( k = 0 \). By definition of \( \text{relax}_G \), \( \text{relax}_G(O, a)(a_2) = O(a_2) \), so \( \text{relax}_G(O, a)^0(a_1) = \text{relax}_G(O, a)(a_2) \) as required.

**Case 3:** \( a_1 \in \text{promoted}_G(O, a), a_2 \notin \text{promoted}_G(O, a) \cup \text{belowpromoted}_G(O, a) \)

Recall that \( O^k(a_1) = O(a_2) \). There are three sub-cases: \( k = 0, k = 1, k \geq 2 \):

- If \( k = 0 \) then \( a_1 = O(a_2) = \text{relax}_G(O, a)(a_2) \), so \( \text{relax}_G(O, a)^0(a_1) = \text{relax}_G(O, a)(a_2) \) as required.

- If \( k = 1 \) then, by definition of \( \text{promoted}_G \), \( O(a) = O(a_1) = O(a_2) \), so \( a_2 \in \text{siblings}_G(O, a) \). As \( G \vdash_{\text{bog}} a_1 \ref a_2 \), then \( a_2 \in \text{promoted}_G(O, a) \), which contradicts the case assumption for \( a_2 \), so this condition does not arise.

- If \( k \geq 2 \), then \( O^{k-2}(O(O(a_1))) = O(a_2) = \text{relax}_G(O, a)(a_2) \). As \( a_1 \notin \text{belowpromoted}_G(O, a) \) then \( O(O(a_1)) \notin \text{belowpromoted}_G(O, a) \cup \text{promoted}_G(O, a) \). By Lemma 7, \( \text{relax}_G(O, a)^{k-2}(O(O(a_1))) = \text{relax}_G(O, a)(a_2) \). By definition of \( \text{relax}_G \) for \( a_1 \), \( \text{relax}_G(O, a)^{k-1}(a_1) = \text{relax}_G(O, a)(a_2) \) as required.

**Case 4:** \( a_1 \in \text{belowpromoted}_G(O, a) \) and \( a_2 \in \text{promoted}_G(O, a) \)

By definition of \( \text{belowpromoted}_G \), there is some \( m \) such that \( O^m(a_1) \in \text{promoted}_G(O, a) \). By definition of \( \text{relax}_G \) for \( a_1 \), \( \text{relax}_G(O, a)(O^m(a_1)) = O(O^m(a_1)) \). Similarly for \( a_2 \), \( \text{relax}_G(O, a)(a_2) = O(O(a_2)) \).

As \( O^m(a_1) \) and \( a_2 \) are both in \( \text{promoted}_G(O, a) \), then \( O^m(a_1), a_2 \in \text{siblings}_G(O, a) \). By definition of \( \text{siblings}_G \), \( O(O^m(a_1)) = O(a_2) \). Therefore \( \text{relax}_G(O, a)(a_2) = O(O^m(a_1)) = \text{relax}_G(O, a)(O^m(a_1)) \).

As \( a_1 \in \text{belowpromoted}_G \) and \( O^i(a_1) \in \text{belowpromoted}_G \), then \( \text{relax}_G(O, a)^m(a_1) = O^m(a_1) \) by Lemma 8 and the definition of \( \text{relax}_G \) for \( O^{m-1}(a_1) \).

Therefore, \( \text{relax}_G(O, a)^{m+1}(a_1) = \text{relax}_G(O, a)(a_2) \), and this case is complete.

**Case 5:** \( a_1 \in \text{belowpromoted}_G(O, a) \) and \( a_2 \in \text{belowpromoted}_G(O, a) \)

Then \( \text{relax}_G(O, a)(a_2) = O(a_2) \). Recall that \( O^k(a_1) = O(a_2) \).
Case 7: \( O(a_2) \in \text{promoted}_G(O, a) \) then \( O(k(a_1)) \in \text{promoted}_G(O, a) \). As \( O \) is a function, then \( O(k(a_1)) = O(O(a_2)) \) and, by definition of \( \text{relax}_G \) for \( O(k(a_1)) \) and \( a_2 \), \( \text{relax}_G(O(a_1))(O(k(a_1))) = \text{relax}_G(O, a)(a_2) \). By Lemma 8, and the fact that \( O^m(a_1) \in \text{belowpromoted}_G(O, a) \), \( O(k(a_1)) = \text{relax}_G(O, a)(k(a_1)) \). Consequently, \( \text{relax}_G(O, a)(k(a_1)) = \text{relax}_G(O, a)(a_2) \) as required.

Case 8: \( O(a_2) \notin \text{promoted}_G(O, a) \) then \( O(a_2) \in \text{belowpromoted}_G(O, a) \). Therefore \( O(k(a_1)) \in \text{belowpromoted}_G(O, a) \). By Lemma 8, \( O(k(a_1)) = \text{relax}_G(O(a_1)) \). By definition of \( \text{relax}_G \) for \( a_2 \), \( \text{relax}_G(O(a_1))(a_2) = O(a_2) \). Therefore \( \text{relax}_G(O, a)(k(a_1)) = \text{relax}_G(O, a)(a_2) \) as required.

**Case 6:** \( a_1 \in \text{belowpromoted}_G(O, a) \) and \( a_2 \notin \text{promoted}_G(O, a) \cup \text{belowpromoted}_G(O, a) \)

Recall that \( O(k(a_1)) = O(a_2) \) for some \( k \). Clearly \( O(k(a_1)) \notin \text{promoted}_G(O, a) \cup \text{belowpromoted}_G(O, a) \). As \( a_1 \in \text{belowpromoted}_G \), there must be some \( m \) such that \( O^m(a_1) \) and \( O^m(a_1) \in \text{promoted}_G(O, a) \). As a consequence of Lemma 8, \( O^m(a_1) = \text{relax}_G(O, a)(m(a_1)) \).

- Clearly, \( k - m \neq 0 \) as that would imply \( O(k(a_1)) \in \text{promoted}_G(O, a) \), contradicting the case assumption.

- If \( k - m = 1 \), then \( O(O^m(a_1)) = O(a_2) \), so, as \( O^m(a_1) \in \text{promoted}_G(O, a) \), \( O^m(a_1), a_2 \in \text{siblings}_G(O, a) \). As \( G \vdash a_1 \ref a_2 \), by the definition of \( \text{promoted}_G \), \( a_2 \notin \text{promoted}_G \). This contradicts the case assumption, so consideration of this case is unnecessary.

- Otherwise, \( k - m \geq 2 \). As \( O^m(a_1) \in \text{promoted}_G(O, a) \), the definition of \( \text{relax}_G \) is applied so that \( O^{k-m-2}(O(O^m(a_1))) = O^{k-m-2}(\text{relax}_G(O, a)(O^m(a_1))) \). Substituting for \( O^m(a_1) \), \( O^{k-m-2}(\text{relax}_G(O, a)(O^m(a_1))) = O^{k-m-2}(\text{relax}_G(O, a)(m(a_1))) \). \( O^{k-m-2}(\text{relax}_G(O, a)(m(a_1))) = \text{relax}_G(O, a)(k(a_1)) = \text{relax}_G(O, a)(a_2) \) as required.

**Case 7:** \( a_1 \notin \text{promoted}_G(O, a) \) and \( a_2 \in \text{promoted}_G(O, a) \)

\( O(k(a_1)) = O(a_2) \), so \( O(k+1(a_1)) = O(O(a_2)) = \text{relax}_G(O(a_2))(a_2) \). By Lemma 8, \( O(k+1(a_1)) = \text{relax}_G(O, a)(k+1(a_1)) \), so \( \text{relax}_G(O, a)(k+1(a_1)) = \text{relax}_G(O, a)(a_2) \) as required.

**Case 8:** \( a_1 \notin \text{promoted}_G(O, a) \) and \( a_2 \in \text{belowpromoted}_G(O, a) \)

\( O(k(a_1)) = O(a_2) \), but \( O(a_2) \notin \text{belowpromoted}_G(O, a) \) or \( O(a_2) \in \text{promoted}_G(O, a) \) by definition of \( \text{belowpromoted}_G \). There is no \( k \) such that \( O(k(a_1)) \in \text{promoted}_G(O, a) \). So this condition cannot arise.

**Case 9:** \( a_1 \notin \text{promoted}_G(O, a) \) and \( a_2 \notin \text{promoted}_G(O, a) \) and \( a_2 \notin \text{belowpromoted}_G(O, a) \)

\( O(k(a_1)) = O(a_2) \) and, by Lemma 7, \( O(k(a_1)) = \text{relax}_G(O, a)(k(a_1)) \). Similarly, \( O(a_2) = \text{relax}_G(O, a)(a_2) \). Thus, \( \text{relax}_G(O, a)(k(a_1)) = \text{relax}_G(O, a)(a_2) \) follows immediately.

Proof complete.

Alisdair Wren

103
Inferring Ownership

Explanation of the proof: We identify three distinct portions of the object graph given a specific relaxation, \( \text{relax}_G(O, a) \): addresses in \( \text{promoted}_G \), addresses in \( \text{belowpromoted}_G \) and all other addresses not found in either. It is then a matter of checking for two addresses, \( a_1 \) and \( a_2 \) such that \( G \vdash \text{ref}_{a_1} a_2 \), that we can find some number of applications of \( \text{relax}_G(O, a) \) to \( a_1 \) such that we get to an address equal to \( \text{relax}_G(O, a)(a_2) \). Thus, the proof has nine cases, though it is impractical to explain each one in full.

Notes to the proof: We tried a number of different well-known proof methods to show this property, including induction over the depth of \( a_1 \) in the ownership tree and the length of paths in \( G \) to \( a_1 \) – a method suggested by earlier proofs of valid ownership. However, the proof above turned out not to be inductive, but was assisted with the inductive power of Lemmata 7 and 8.

We now tie up the details of the proof that \( \text{relax}_G(O, a) \) is a well-formed ownership for \( G \):

**Theorem 17** \( \forall a \in \text{dom}(G)[G \vdash O \Diamond _{\text{own}} \Rightarrow G \vdash \text{relax}_G(O, a) \Diamond _{\text{own}}] \)

**Proof:**

It must firstly be shown that \( \text{relax}_G(O, a)(a_r) = a_r \):

As \( G \vdash O \Diamond _{\text{own}} \), then \( O(a_r) = a_r \) and \( O(O(a_r)) = a_r \). As \( \text{relax}_G(O, a)(a_r) \) is equal either to \( O(a_r) \) or \( O(O(a_r)) \), it is clear that \( \text{relax}_G(O, a)(a_r) = a_r \) as required.

It remains to be shown that \( G \vdash a_1 \text{ ref } a_2 \Rightarrow \exists k[\text{relax}_G(O, a)^k(a_1) = \text{relax}_G(O, a)(a_2)] \). This result follows immediately from Lemma 9.

Conclude that \( G \vdash O \Diamond _{\text{own}} \Rightarrow G \vdash \text{relax}_G(O, a) \Diamond _{\text{own}} \).

That \( \text{relax}_G(O, a) \) is a valid ownership is vital if we are to use \( \text{relax}_G \) to adapt ownership functions, but we have discussed two other desirable properties. Firstly, that the relaxation of an ownership function is less precise than its input ownership function:

**Theorem 18** \( O \sqsubseteq \text{relax}_G(O, a) \)

**Proof:**

The result follows immediately by definition of \( \text{relax}_G \). Take an arbitrary address \( a' \). If \( a' \in \text{promoted}_G(O, a) \), then \( O(O(a')) = \text{relax}_G(O, a)(a') \), so \( \exists k[O^k(a') = \text{relax}_G(O, a)(a')] \).

Otherwise, \( O(a') = \text{relax}_G(O, a)(a') \), so again \( \exists k[O^k(a') = \text{relax}_G(O, a)(a')] \).

As \( a' \) was arbitrary, conclude that \( \forall a' \exists k[O^k(a') = \text{relax}_G(O, a)(a')] \), and thus that \( O \sqsubseteq \text{relax}_G(O, a) \).

We finally conjecture that \( \text{relax}_G(O, a) \) is an ‘atomic’ relaxation – that it relaxes the graph by the minimum amount required to maintain a valid ownership.
Conjecture 1 \( O_1 \subseteq O_2 \land O_1 \neq O_2 \Rightarrow \exists a_0[\text{relax}_G(O, a_0) \subseteq O_2] \)

It does not seem unreasonable that this should hold – we know that all the modifications to the ownership function made by \( \text{relax}_G \) are necessary, and we make no additional modifications.

8.2.2 Using \( \text{relax}_G \) as an Ownership Inference Algorithm

An obvious stimulus for relaxation of the ownership function might be the addition of a new reference to the object graph. The addition of a reference to address \( a \) may not break the ownership invariant, but if it does, then \( a \) will have to be moved up the ownership tree, as illustrated in Figure 8.5.

Where such a situation occurs, then Theorem 17 indicates that a new valid ownership can be produced with one or more applications of \( \text{relax}_G(O, a) \), where \( a \) is the object at the target of the new reference. If Conjecture 1 holds, then we can quickly modify the ownership as each reference is added and obtain the most precise ownership. Thus, this approach may be an ideal algorithm for calculating ownerships on-the-fly in object heaps. Such an algorithm would be important for dynamic inference of ownership in, for example, a garbage collection system as discussed in Section 4.1.
8.3 Ownership Type-Consistent Functions

We have already seen how the type system’s imprecision enforces constraints on the ownerships derived from run-time object heaps. We say that an ownership is *Ownership Type Consistent* or *OT-Consistent* if the ownership satisfies all the constraints imposed by the type system. In other words, the ownership is an ownership that could have been the result of an ownership-typed program run.

In this section, we examine three *necessary* properties of an OT-Consistent ownership; properties that an ownership must satisfy if it is to be used to suggest ownership types. From this point on, our discussion is informal, and intended to show how one might go on from reliable ownership inference to ownership types on untyped program text.

Now that the type system is under consideration, we must also consider specific fields. Therefore, we replace our general $\text{ref}$ relation with the statement $a' \in a.f_1$, which specifies that $a = (c, F, p_n, a_c) \wedge a' \in F(f)$. We extend this notation so that $a_n \in a_0.f_1.f_2 \ldots .f_n$ when $a_1 \in a.f_1 \wedge a_2 \in a_1.f_2 \wedge \ldots \wedge a_n \in a_{n-1}.f_n$. This notation is the analogue of the $\text{obj.field}$ notation from object-oriented languages, but the result is a set representing the various possible values that are present in the extended object graph. We omit the specification of the graph, $G$, from the notation – it is an implicit parameter to the notation.

8.3.1 Field-Ownership Uniformity

While motivating the storage of object creation information in the object graphs (Section 4.4.6), we acknowledged that objects created at a single point of code by the same object must have precisely the same ownership characteristics, as ownership parameters are invariant at run-time.

The same is true of addresses stored in the same field of the same object. Therefore, all objects stored in any single object field must have the same ownership: $a_1, a_2 \in a.f \Rightarrow O(a_1) = O(a_2)$. Moreover, as objects at $a_1$ and $a_2$ must have the same ownership characteristics – the ownership parameters associated with the field $f$ cannot change – then the values in their fields must have the same characteristics. We therefore extend our requirement on the object graph.

**Definition 14** An ownership, $O$, for extended object graph $G$ is *Field-Ownership Uniform* if and only if:

$$a_1, a_2 \in G \models_{\text{og}} a.f_1.f_2 \ldots .f_n \Rightarrow O(a_1) = O(a_2)$$

Our extended object graph permits ownership which do not satisfy the condition, so any ownership under consideration must be modified with $\text{relax}_G$ to comply with Definition 14, or must be rejected as a potential ownership.
8.3.2 Class-Ownership Uniformity

As discussed in Section 3.6.2, all the ownership parameters passed into a class must be outside (≻) the owner. Therefore, whenever an object references some object inside (≺) its own context, it must have a field annotated with the this parameter, which cannot change for different instances of that class. Thus, if any class’s instance references another object inside its own context, then all such instances must do so:

**Definition 15** An ownership, $O$, for extended object graph $G$ is Class-Ownership Uniform if and only if:

$$G(a_1) = (c, F_1, p_{n1}, a_{c1}) \land G(a_2) = (c, F_2, p_{n2}, a_{c2})$$
$$\land a_3 \in a_1.f \land a_4 \in a_2.f \land O(a_3) = a \Rightarrow O(a_4) = a_2$$

This constraint cannot, of course, be satisfied by pushing objects into the contexts of the objects that reference them, lest it invalidate the ownership constraints. Instead, where a field of several class instances variously references inside and outside the instances’ contexts, relax$_G$ should be used to move ‘inside’ objects outside of the class instance’s context.

8.3.3 Allocation-Ownership Uniformity

Objects allocated by the same creator object by the same new statement must have the same ownership characteristics – that is, their ownerships and parameter lists must be identical. Additionally, therefore, all field references from those objects must also be identically owned. We therefore impose the following constraint:

**Definition 16** An ownership, $O$, for an extended object graph, $G$, is Allocation-Ownership Uniform if and only if, for every $n \geq 0$:

$$G(a_1) = (c, F_1, p_n, a_c) \land G(a_2) = (c, F_2, p_n, a_c)$$
$$\land a_3 \in a_1.f_1.f_2\ldots f_n \land a_4 \in a_2.f_1.f_2\ldots f_n \Rightarrow O(a_3) = O(a_4)$$

Again, satisfaction of this constraint may require that we move some objects upwards in the object graph such that they share a common ownership, for which we may use relax$_G$.

8.4 Summary

In this chapter we discussed methods for adapting ownerships so that they are consistent with other constraints, external (or additional) to the extended object graphs. We gave the definition of relax$_G$, a function for adapting ownerships which does so without violating
the ownership invariant and, we conjecture, with the minimum loss of precision. We also showed how this might be used for performing the calculation of ownership in object heaps as the heaps are constructed.

The idea of Ownership Type (OT-) Consistency was introduced: OT-Consistent ownerships are ownerships that might potentially arise from an ownership typed program. We informally argued how three properties – Field-Ownership Uniformity, Class-Ownership Uniformity and Allocation-Ownership Uniformity – are necessary for OT-Consistency. Thus, we provide further restrictions on the space of potentially inferred ownership typings that must be checked during a dynamic inference.
Chapter 9

Constraining Object Ownership for a Static System

This chapter discusses how we might apply all the previous work on selecting ownerships to finally infer an ownership type for a program. We discuss type inference by showing how the run-time and static substitutions, which govern the flow of ownership in a program, can be inferred from the ownerships deduced by previous work. A scheme for annotating class fields with ownership types is then discussed, and we make some suggestions for distributing these types to produce a fully ownership-typed program. The chapter concludes with a note on the use of the constraint system that results from this work.

9.1 Introduction

The work from previous chapters now allows us to consider the application of the most-precise, valid, ownership type-consistent ownership function to the original program text.

In Section 4.3, we saw how the inference of ownership parameters is the main difficulty facing ownership type inference. Section 3.5 also discussed how the ownership parameters are directly determined by the substitutions necessary to propagate ownership contexts down through the ownership hierarchy. We therefore discuss the potential for inferring these substitutions.

9.2 Inferring Dynamic Substitutions

Recall the dynamic substitutions described in Section 3.5.1, where we saw how every context parameter was instantiated with an object context (represented by an address). For details, refer to the semantics and type-system of $\text{Joe}_0$ in Appendix A.
Inferring Ownership

class X<owner> {
    Y<this, owner> fxy;

    void mX() {
        fxy = new Y<this, owner>;
        fxy.mY();
    }
}

class Y<owner, p1> {
    X<this> fyx;
    Z<p1> fyz;

    void mY() {
        fyx = new X<this>;
        fyz = new Z<p1>;
    }
}

class Z<owner> { ... }

Figure 9.1: This figure was used to represent how contexts were passed through the ownership tree so that deeper objects could refer to contexts outside their own. This chapter now discusses how to infer such a substitution, and this figure reprises its role as an example.

We propose a number of necessary constraints upon these run-time substitutions, in order to further narrow the choice of ownership types that can be inferred.

First, however, we define the function $O_S$ – named consistently with the semantics in Appendix A and not to be confused with ownership functions $O$ – which returns for each address $a$ the substitution for instantiating $a$’s context parameters.

$$O_S : \text{Address} \rightarrow (\mathbb{N} \rightarrow \text{Address})$$

As we are inferring parameters which do not exist in the program text, we simply number the parameters from left to right; hence the use of $\mathbb{N}$ in the type definition above. In the code of Figure 9.1, $Y$’s parameters would be named 1 and 2 as opposed to owner and p1 respectively. There is also an implicit parameter, which represents this, which we shall number 0.

Rather than attempting to infer the $O_S$ function for each address directly, there are a number of constraints we can impose on it to reduce the possible number of such functions, given an object graph ownership, $O$. 
Chapter 9. Constraining Object Ownership for a Static System

9.2.1 Instantiating this and owner

The first constraint is that the owner parameter should be instantiated with the owner according to the valid ownership under consideration, and that this should be instantiated with the context for which the substitution applies: Thus, we immediately obtain the portion of the substitution for parameters 0 and 1 (this and owner respectively) of all classes:

\[
\forall a [ Os(a)(0) = a ] \nonumber \\
\forall a [ Os(a)(1) = O(a) ]
\]

The remainder of the constraints only need be applied to any parameters \( p \) such that \( p \geq 2 \).

According to this constraint, in our example of Figure 9.1 it must be that \( Os(y1)(0) = y1 \) and \( Os(y1)(1) = x1 \).

9.2.2 Parameter Counts

The second constraint represents the observation that all a specific class’s objects must have the same number of ownership parameters.

\[
\forall c [ \exists n [ G(a) = (c,F,p_n,a_c) \Rightarrow \text{dom}(Os(a)) = \{1..n\} ] ]
\]

If this \( n \) can be identified, then we immediately have names for all the parameters in class \( c \). We refer to these parameters as \( \text{params}(c) \), which is equal to \( \text{dom}(Os(a)) \) for some instance of class \( c \) when the above constraint is satisfied.

Referring again to our example of Figure 9.1, both instances of \( X \) must have the same number of substituted parameters, so we can say that \( |\text{dom}(Os(x1))| = |\text{dom}(Os(x2))| \).

9.2.3 Imported Contexts

We have already discussed how the role of ownership parameters is to import contexts into other objects’ contexts such that they may refer to objects higher up the ownership tree. It is useful, then, to classify which contexts are used by each object for typing each reference:

\[
\text{uses} : \text{OwnershipFunction} \rightarrow \text{Address} \rightarrow \mathcal{P}(\text{Address})
\]

\[
\text{uses}(O,a) = \{ a_1 | O \vdash a_2 \prec a \land a_2.f = a_1 \land O \vdash a \prec a_1 \}
\]

We use the notation \( O \vdash a \prec a’ \) as a more concise representation of \( \exists k [ O^k(a) = O(a’) ] \).

\( \text{uses}(O,a) \) contains, by its definition, all those addresses outside \( a \) which are referenced by addresses inside \( a \). It is, therefore, the set of contexts which must be made available to addresses inside \( a \) in some way – either by some parameter passing, or by the fact that the required context is the enclosing context for such addresses. In Figure 9.1, we observe
that \( y_1 \) references its own context and that of the owner of \( x_1 \). Therefore, \( \text{uses}(O, y_1) = \{ y_1, \text{owner}(x_1) \} \).

We can now impose the constraint that all those addresses so required must be passed into the context by the substitutions made by \( O_s \):

\[
\text{range}(O_s(a)) \supseteq \text{uses}(O, a)
\]

Of course, from the earlier constraint that \( O_s(a)(0) = a \) and \( O_s(a)(1) = O(a) \), we know that \( O(a) \) and \( a \) are also in \( \text{range}(O_s(a)) \). The context of \( a \)'s owner, however, must be given to \( a \) explicitly through the parameter system, so \( O(a) \) is always in the range of \( a \)'s run-time parameter substitution. The context for \texttt{this}, on the other hand, is implicitly given to every \( a \). We can therefore refine this constraint slightly:

\[
\forall a \in \text{Address}, n \geq 2 \{ O_s(a)(n) \in \text{uses}(O, a) - \{ a \} \}
\]

Thus, we have that every context used by \( a \) (or some object inside \( a \)) must occur in some parameter excluding 0 and 1, which we have reserved for \texttt{this} and \texttt{owner} respectively.

For \( y_1 \) in the example of Figure 9.1, we therefore know that the context of \texttt{owner}(x_1) must be in some ownership parameter.

\texttt{uses} also gives us a lower bound on the number of ownership parameters that will be required – clearly if an address requires access to, say, four ownership contexts other than its own context, then four ownership parameters will be required. So, we know from the diagram of Figure 9.1 that \( Y \) has at least two ownership parameters – one for its owner, and one for its other context in \texttt{uses}. As we can see, this lower bound is in fact correct, as \( Y \) does indeed have one extra parameter – \( p_1 \). With the constraint above, we also know that the substitution for this parameter, known as 2 during inference on untyped program text, will be \( 1 \mapsto \text{owner}(x_1) \). Thus, according to our scheme so far, the inference for \( Y \) is already complete.

### 9.3 Inferring Static Substitutions

We saw in Section 3.5.2 that dynamic and static substitutions were intertwined, and proposed that this may allow us to infer static substitutions directly from the dynamic substitutions discussed above. The constraints from the previous section should dramatically restrict the number of potential dynamic substitutions that may be appropriate.

We now see how static substitutions may be inferred: For a field, \( f \), in class \( c \) we can determine the class type of \( f \) using the field look-up function, \( F \), associated with the program that is being typed (again, see Appendix A for details). Thus, we have that:

\[
F(P, c, f) = d
\]
Chapter 9. Constraining Object Ownership for a Static System

Where $P$ is the program to be typed, $c$ is the class holding $f$, and $d$ is $f$’s class type in $c$. Then, for all such classes $c$ and $d$, and fields $f$ such that $\mathcal{F}(P, c, f) = d$, there must be some static substitution that substitutes $d$’s parameters from those available in $c$. We will call this static substitution $\sigma_{c,f}$, which we immediately know must map the parameters of $d$ to parameters of $c$:

$$\sigma_{c,f} : \text{params}(d) \rightarrow \text{params}(c)$$  \hfill (9.1)

We now impose constraints on this substitution based on what we know of the run-time substitutions. Suppose, for some addresses $a$, that $G(a) = (c, F, p_n, a_c)$ according to the object graph used for the original inference of ownership (or, indeed, any object graph resulting from the execution of program $P$). Then, suppose that some $a' \in a.f$, for the field $f$ that is under consideration. If $\sigma_{c,f}(n) = n'$ — a substitution from $d$’s parameter, $n$, to $c$’s parameter, $n'$, then that must be position-wise identical to the substitution occurring at run-time when the parameters of $a'$ are instantiated with the contexts used in $a$: $Os(a')(n) = Os(a)(n')$.

When we put it all together, then, we formulate the following constraint on the run-time substitutions, $\sigma$, for a program $P$:

$$\forall c, d, f \mathcal{F}(P, c, f) \Rightarrow \exists \sigma_{c,f} : \text{params}(d) \rightarrow \text{params}(c)$$

such that:

$$\forall n \in \text{params}(d)[\sigma_{c,f}(n) = n' \Rightarrow \forall a, a'[G(a) = (c, F, p_n, a_c) \land a' \in a.f \Rightarrow Os(a')(n) = Os(a)(n')]]$$

This constraint specifies that the run-time substitutions are a valid instantiation of the static substitutions, $\sigma$.

We are thus able, yet again, to restrict the set of possible inference decisions that must be checked. In the following section, we shall see how these substitutions can be used to apply types to the program text.

**Example:** We look again at the program text (supposing it was untyped). As established in the previous discussion of dynamic substitutions, we have established $Os(O, y1)$:

- $Os(O, y1)(0) = y1$
- $Os(O, y1)(1) = x1$
- $Os(O, y1)(2) = \text{owner}(x1)$

Suppose we also have $Os(O, x1)$, which is easily calculated in the same way as for $y1$:

- $Os(O, z2)(0) = z1$
- $Os(O, z2)(1) = \text{owner}(x1)$

Alisdair Wren 113
Inferring Ownership

```java
class Y<1, 2> {
    X< ? > fyx;
    Z< ? > fyz;
    void mY() {
        fyx = new X< ? >;
        fyz = new Z< ? >;
    }
}

class Z<1> { ... }
```

Figure 9.2: The code on the left is untyped save for the number of parameters required by each class. The code on the right has been annotated with the types suggested by a set of static substitutions, $\sigma_{Y,f}$.

We then need to infer a static substitution, say, for the $fyx$ field of $Y$. From the constraint above, it must be that $\sigma_{Y,fyx}(1)$ selects the same choice of 0, 1 or 2 as does $O_s(z1)(1)$. Notice that $O_s(O,z1)(1) = O_s(O,y1)(2)$, and therefore conclude that the necessary substitution is $\sigma_{Y,fyz}(1) = 2$ – in the actual program we find this is correct, as $Z$'s ownership parameter (1 during inference) is substituted with $p1$ (2 during inference).

### 9.4 Inferring Types

We now assume that we have suggestions, for every class $c$ and field $f$ of program $P$, for static context substitutions, $\sigma_{c,f}$. These may now be used to infer types on the program text.

Suppose we have the following substitutions for class $Y$, such as might be inferred from the run-time substitution already discussed:

\[
\begin{align*}
\sigma_{Y,fyx}(1) &= 0 \\
\sigma_{Y,fyz}(1) &= 2
\end{align*}
\]

We now take the program text from Figure 9.1, this time without types. From the domain of all the substitutions $\sigma_{Y,f}$, we can find the number of parameters that $Y$ takes. Such information for all the other classes can be similarly obtained, so we can annotate the definitions of $Y$ and $Z$ with parameter names and placeholders seen on the left of Figure 9.2.

As $\sigma_{Y,fyx}(1) = 0$, we can immediately annotate $fyx$ with the owner $this$ – 0’s analogue in the ownership type system. $\sigma_{Y,fyz}(1) = 2$ then tells us that we must use the second
Chapter 9. Constraining Object Ownership for a Static System

```java
class Y<1, 2> {
    X<#this#> fyx;
    Z<2>     fyz;

    void mY() {
        fyx = new X<#this#>;
        fyz = new Z<2>;
    }
}

class Z<1> { ... }
```

Figure 9.3: The final typed code, in which the types of the fields have been propagated to the `new` statements that assign to them. Similarly, if `mY` had some argument which was assigned, say, to `fyx`, then the parameter’s type would have been similarly derived.

parameter to `Y` as the first parameter of `fyz`’s type, so it can be typed `Z<2> fyz`. We then obtain the partially typed program on the right of Figure 9.2.

9.4.1 Type Propagation

We have thus far established a system for typing all the class fields in an untyped program. It remains now to propagate these types to other typed expressions in the program, which we show by example: The two `new` statements require types, yet these can be immediately inferred from the fields they assign to; the ownership type system specifies that the ownership parameters of the left-hand side of the assignment must match the right. Therefore, we propagate the types of `fyx` and `fyz` to the `new X` and `new Z` statements respectively, as shown in Figure 9.3.

If there had been method parameters, then their type would have been similarly derived from the fields they assign to.

9.5 Conclusions

The constraints on both run-time and static substitutions only restrict the space of possible substitutions which can be inferred from the object graph ownerships, themselves constrained by OT-Consistency and the ownership invariants on which they are based. The constraints are sound with respect to the ownership type system, but are unlikely to be complete – more restraints may be required before we can be sure all possible results from the ‘pipeline’ of successive constraints will yield only type-correct program annotations.

Therefore, the ownership typings that result from the processes outlined in this chapter
must still be type-checked according to the ownership type rules. It is still a matter of further work to prove that these constraints always have at least one type-correct solution, to show how much they constraint the space of possible typings, and whether or not all solutions are always type-correct.

9.6 Summary

In this chapter, we showed how to restrict the space of possible run-time context substitutions, which drive contexts through the ownership system. From these run-time substitutions, we showed how static substitutions can be derived and, directly, how types can be applied to the class fields of an untyped program. Finally, we discussed in overview how one might propagate types through the program to typed expressions other than field names.

Thus, we complete our survey of the proposed inference strategy from Chapter 4.
Chapter 10

Conclusion

In this final chapter, we review the work presented in this report, its contribution and potential for further development. We conclude with some final remarks.

10.1 This Project’s Contribution

In this project we have demonstrated one potential method for inferring ownership types from program text, through run-time observation and static analysis. We show how successive constraint and refinement of run-time information can produce viable suggestions for a static-typing, and have achieved a more complete understanding of the challenges underlying the inference of ownership types. While there are undoubtedly other methods for annotating program text with ownership types, it seems likely that a constraint-based approach will prove fruitful, given the complexity involved and the success of constraint systems for solving other object-oriented type inference problems[20]. However, without an upper bound on the number of variables – in our case, ownership parameters – our system remains undecidable in general.

We defined a formal system for detailed consideration of domination and ownership in object graphs and, as a result were able to prove a number of results. These results imply some ideas for fast, precise algorithms for calculating ownership in object heaps. We have discussed how this information can make significant contributions to improving the effectiveness and speed of garbage collection[4], run-time program optimisation and understanding[14].

To show how these ownerships can be used to infer ownership types, we then showed how ownerships can be adapted as the need arises – either when a new reference needs to be accommodated according to the ownership invariants, or where we wish to achieve conformance with the conservative type system. Again, we formally proved some properties of this transformation, and showed that it always produces valid ownerships. We also showed
Inferring Ownership

how it might be used to generate all the possible ownerships for an object heap, starting with the most precise ownership, followed by the application of successive relaxations. The relaxation, then, can be safely used as the basis of an algorithm to build ownership information as dynamic structures are created.

Finally, we related the results of these investigations to the structures used in the formal description of the ownership type system itself, specifically the substitutions which are responsible for the flow of ownership through an ownership typed program. We gave a number of further constraints, this time applied to these substitutions, and showed how these structures could be used to infer the program annotations which make up an ownership type.

10.2 Further Work

The potential for development of this work is considerable and varied. We outline some possible avenues for further research and refinement of the system:

Development of Run-Time Ownership Inference The results of the run-time ownership inference could most usefully be explored with an implementation, now that it has been placed on a sound theoretical footing. Algorithms may be derived from this work to infer dynamic ownership, which should be compared with existing solutions for their speed and precision. Once developed, ownership inference could be put to some use, for example in some run-time environment’s garbage collector. This could be used to investigate the potential performance benefits, and how approximation of the results – for example, by producing less-precise ownerships – impairs effectiveness.

Completion of OT-Consistency In Chapter 8, we saw how the type system imposes a number of constraints on valid ownerships. While we gave some necessary conditions, we do not expect these to be sufficient. The addition of extra constraints and a formal proof that they are consequences of the type system are both important goals for this part of the work.

Investigation of the Static Constraints There are a number of constraints which we impose on the semantic structures we infer in Chapter 9. Again, there are likely to be other constraints from the type system which can be used to further refine and constrain the potential results of our inference. A formal demonstration of their completeness is also desirable. For both parts of the constraint system, however, it is important to show that the system is not over-constrained: The type system is necessarily conservative, but it is important not to throw away more ownership precision than necessary given its importance to the effectiveness of an ownership typing.

If the constraint system is complete, all ownerships arising as a result should produce type-correct annotations. Then, with a complete constraint system, there are a num-
ber of important properties which must be shown: Firstly, it is important to show, for every potentially type-correct program, that the constraint system will give at least one suggestion for a typing. A subsequent demonstration that the constraints produce a unique typing, and that this typing is the best, most restrictive, possible typing will also be required before implementation becomes practical. (Of course, a partial implementation may help investigations into the constraint system’s behaviour.)

**Language Extensions** The language $\mathcal{J}oe_0$, on which this work is based, is a very restricted subset of a Java-like object-oriented language, extended with ownership types (see Appendix A). Expanding the language will require redefinition of the type system and semantics, as well as a proof of ownership soundness. The inference system’s complexity will also increase: Extension to, for example, multiple-assignment variables will complicate the data flow analysis required during the type propagation phase of annotation inference (Section 9.4.1). However, a more fully-featured language is vital if the benefits of inference are to be studied on ‘real world’ problems.

**Implementation** Once the investigations of the constraint system have been completed, and it has been developed to the point where all the desirable properties hold, then a number of practical aspects must be considered. The complexity of the system is of immediate importance: If the constraint system can reliably infer ownership types, then it is still important that this inference is tractable.

There are also some practical implications for the collection of the original object-graph data. This will certainly require the extension of a virtual runtime environment with the ability to collect and record data about object allocations.

**Other Strategies and Static Inference** The ultimate goal must remain a tractable means by which to infer ownership types directly from program text, though there is a significant trend in run-time observation of programs to establish program properties. There is also potential for a variety of other inference techniques instead of the combination of run-time observation and constraint systems that has been proposed in this work. These also merit thorough investigation and comparison.

### 10.3 Remarks

Throughout this project, a detailed formal treatment of ownership has been immensely helpful to determine new avenues of investigation, and to highlight errors our intuitions. Such errors have occasionally been subtle, and may have been incarnated as equally subtle software bugs. Thus, with detailed formal proofs of correctness, we can be confident in this work’s application to problems where correctness is vital.

The inference of ownership types has a complexity which clearly exceeds that of traditional type inference[10, 20]. It is the hope that this work provides some insight into the issues underlying type inference and the ideas of ownership in general, and that an efficient static inference may be forthcoming.
Bibliography


Appendix A

A Minimal Object-Oriented Language with Ownership

This appendix presents \( J_0 \) from a paper under construction by Matthew Smith and Sophia Drossopoulou. The remainder of this appendix is taken directly from that paper. Many thanks to both the authors for permission to reproduce work-in-progress here.

We give a complete, formal definition of \( J_0 \). \( J_0 \) is a subset of the OOPSLA-02 Joe. In \( J_0 \) we have omitted let-expressions, and effects. Also, we have replaced overloaded definitions/judgments by two or more definitions or judgments. In other words, \( J_0 \) is a little Joe, expressed "pour les nulles".

A.1 Syntax

\[
\begin{align*}
\text{classDecl} & ::= \text{class} \mid \alpha^* \& \text{where} \rho^* \\
& \quad \text{extc}^\langle \sigma^\rangle \\
& \quad \{ \text{fldDecl}^* \& \text{methDecl}^* \}
\end{align*}
\]

\[
\alpha \in \text{OwnParam} = \text{Identifier} \cup \{ \text{owner} \} \quad \text{ownership parameters}
\]

\[
\beta \in \text{OwnContxt} ::= \alpha \mid \this \mid \text{world} \quad \text{ownership contexts}
\]

\[
\rho \in \text{OwnRestrs} ::= \beta \prec \beta' \quad \text{restrictions}
\]

\[
\text{fldDecl} ::= t \ \text{f}
\]

\[
\text{methDecl} ::= t \ \text{m} \ ( t \ x ) \ \{ \ e \}
\]

\[
\text{type} ::= c^\langle \sigma \rangle
\]

\[
\sigma \in \text{OwnSubsts} ::= ( \alpha \mapsto \beta )^* \quad \text{static substitutions}
\]

\[
\text{exp} ::= \this \mid x \mid e.f \mid e.f := e \\
& | e.m ( e )
\]

\[
\]
In the above, the identifiers $c, \alpha, f, m, x \in \textit{Identifier}$, stand for a class identifier, an ownership parameter identifier, a field identifier, a method identifier, and a method parameter identifier.

Substitutions

Static substitutions, $\sigma$, map the formal ownership parameters of a class $c$, onto the ownership contexts available in the current program position (Which ones are available is expressed through an environment, c.f. next section).

We will see later that we shall need to compose substitutions in order to obtain new ones. The composition of substitutions is defined in terms of function composition, i.e., ,

$$(\sigma \circ \sigma')(\alpha) = \sigma(\sigma'(\alpha)),$$

with the addition that $(\sigma \circ \sigma')(\alpha) = \textbf{world}$, if $\sigma'(\alpha) = \textbf{world}$. Note also, that if $\textbf{this} \in \textbf{rng}(\sigma)$, then $(\sigma \circ \sigma')$ is undefined. This means, that a type which ”attempts” to access ”inside” and ownership box from the outside is undefined.

Also, we define $0^\sigma$ to be the empty substitution, and $1^\sigma$ to be the substitution which maps any identifier to itself. Therefore, for all $\sigma$, we have $\sigma \circ 0^\sigma = 0^\sigma = 0^\sigma \circ \sigma$, while $\sigma \circ 1^\sigma = \sigma = 1^\sigma \circ \sigma$.\(^1\) Thus, for any $\sigma$ with $\textbf{this} \in \textbf{rng}(\sigma)$, the composition $\sigma \circ \sigma'$ is defined only if $\sigma' = 1^\sigma$, which means that we can access ”inside the box” only if we are already inside the box.

Environments, Context inside other contexts, and well-formed types

An environment, $E$, contains the types for the receiver ($\textbf{this}$) and the argument ($x$) as well as the restrictions for all locally available ownership contexts.

$$E \in \textit{Environment} ::= t \textbf{this}, t \ x, \rho^*$$

From an environment we can deduce which ownership contexts are ”inside” other ownership contexts. This is expressed through the judgment $E \vdash \beta' \prec \beta$:

$$E \vdash \beta' \prec \beta'' \quad E \vdash \beta'' \prec \beta$$

$$\frac{E, \beta' \prec \beta', E' \vdash \beta' \prec \beta}{E, E' \vdash \beta' \prec \beta}$$

We can define a lookup function $\textit{Restrs}(P, c)$ which reads the restrictions from the \textit{where}-clause in the class declaration for $c$, and adds the restrictions $\textbf{this} \prec \alpha$ and $\alpha \prec \textbf{world}$ for all the ownership parameters, $\alpha$ of $c$.

\(^1\)That is, $0^\sigma$ and $1^\sigma$ are the 0 and 1 of the semi-group, but I am unsure of the exact terms.
Well-formed types, and Subtypes

We now define well-formedness of a type \(c<\sigma>\), in the judgment \(P,E \vdash c<\sigma>\). This requires the class \(c\) to have been defined in \(P\), and the ownership substitution \(\sigma\) to map the ownership parameters of \(c\) into the contexts in \(E\) so, that all the restrictions from \(c\) are satisfied in \(E\):

\[
\frac{\beta' < \beta \in Restrs(P, c)}{P,E \vdash c<\sigma> \quad E \vdash \sigma(\beta') < \sigma(\beta)}
\]

We now define when a type \(t'\) is a subtype of \(t\), in terms of the judgement \(P \vdash t' \leq t\). Note that for subclasses, i.e. when inferring \(P \vdash c<\sigma> \leq c'<\sigma \circ \sigma'\) subtyping involves the combination of \(\sigma\), the substitution that makes \(c\) a type in the current environment, with \(\sigma'\), the substitution that makes \(c\) a subclass of \(c'\).

\[
\frac{P \vdash t' \leq t'' \quad P \vdash t'' \leq t' \quad P \vdash t' \leq t \quad P \vdash c<\sigma> \leq c'<\sigma \circ \sigma'}{P \vdash c<\sigma> \leq c'<\sigma \circ \sigma'}
\]

Note that it is possible for a type \(t'\) to be a subtype of \(t\), and for \(t'\), or \(t\) not to be well-formed. \(^2\)

Field and Method lookup

We now define field and method look up functions. The field lookup \(\mathcal{F}(P, t, f)\) returns the type of \(f\) as declared (or inherited) in \(t\). The function \(\mathcal{F}s(P, t)\) returns all the fields declared or inherited in \(t\), while the function \(\mathcal{M}(P, t, m)\) returns the argument type, the result type, and the method body of \(m\) as declared or inherited in \(t\):

\[
\begin{align*}
\mathcal{F} & : \text{Prg} \times \text{Type} \times \text{FldId} \rightarrow \text{Type sup}\{c\} \\
\mathcal{F}s & : \text{Prg} \times \text{Type} \times \text{FldId} \rightarrow \text{POWERSET(FldId)} \\
\mathcal{M} & : \text{Prg} \times \text{Type} \times \text{MthId} \rightarrow \text{Type} \times \text{Type} \times \text{Exp}
\end{align*}
\]

Field lookup needs to find the type of of a field \(f\) defined in type \(c<\sigma>\); this means that the substitution \(\sigma'\) from the declared type of \(f\) will be combined with the substitution \(\sigma'\) perspective of the substitution \(\sigma\). This is why, \(\sigma'\), the substitution in the type of \(f\) has to be composed with \(\sigma\):

\[\text{Field lookup needs to find the type of of a field } f \text{ defined in type } c<\sigma>; \text{ this means that the substitution } \sigma' \text{ from the declared type of } f \text{ will be combined with the substitution } \sigma' \text{ perspective of the substitution } \sigma. \text{ This is why, } \sigma', \text{ the substitution in the type of } f \text{ has to be composed with } \sigma:\]

\(^2\)However if \(t'\) is well-formed for an environment \(E\), then \(t\) will also be well-formed in \(E\), but not the opposite - TO PROVE. Even stronger, it is possible to have \(P \vdash t' \leq t\), but there exists no environment \(E\) where \(P, E \vdash t\).
\[ F(P, c<\sigma>, f) = \begin{cases} 
\epsilon & \text{if } c<\sigma> = \text{Object}<\epsilon> \\
\epsilon' & \text{if } c \text{ contains } c'<\sigma'> \text{ and } f \\
F(P, c'<\sigma'), f) & \text{if } f \text{ not defined in } c, \text{ and } P(c) = \text{class}<... > ... \text{ext}c'<\sigma'> ... 
\end{cases} \]

\[ F_s(P, c) = \{ f | F(P, c, f) \neq \epsilon \} \]

Similarly, for method lookup, ...

\[ M(P, c<\sigma>, m) = \begin{cases} 
\epsilon & \text{if } c<\sigma> = \text{Object}<0\sigma> \\
(t, t', e) & \text{if } c \text{ contains } c_1<\sigma_1> m ( c_2<\sigma_2> x ) \{ \text{expr} \} \\
F(P, c'<\sigma'), m) & \text{if } m \text{ not defined in } c, \text{ and } P(c) = \text{class}<... > ... \text{ext}c'<\sigma'> ... 
\end{cases} \]

\[ M(P, c<\sigma>, m) = \begin{cases} 
(t, t', e) & \text{if } c \text{ contains } c_1<\sigma_1> m ( c_2<\sigma_2> x ) \{ \text{expr} \} \\
M(P, c'<\sigma'), m) & \text{if } m \text{ not defined in } c, \text{ and } P(c) = \text{class}<... > ... \text{ext}c'<\sigma'> ... 
\end{cases} \]

A.2 Operational Semantics

\[ \sim : \text{Exp} \times \text{Prg} \times \text{Stck} \times \text{Heap} \rightarrow \mathbb{R} \times \text{Heap} \]

\[ S \in \text{Stck} = ( \{ \text{this} \} \cup \text{Param} ) \rightarrow \mathbb{N} \]

\[ H \in \text{Heap} = \mathbb{N}^+ \rightarrow \text{Obj} \]

\[ o \in \text{Obj} = \text{ClassId} \times \text{FieldMap} \]

\[ fm \in \text{FieldMap} = \text{FldId} \rightarrow \mathbb{N} \]

\[ r \in \mathbb{R} = \mathbb{N} \cup \{ \text{NullPntrExc, StuckExc} \} \]

\[ k \in \mathbb{N} \]

\[ \iota \in \mathbb{N}^+ \]

Results, \( r \), are either addresses, or null (0), or exceptions. Addresses, \( \iota \in \mathbb{N}^+ \), are numbers. Stacks, \( S \), are sequences of stack frames, \( S \), which map this to an address, and the parameters \( x_1, ... x \) to an addresses or null. Heaps, \( H \), map addresses to objects. The notation \( o = ( c, fm ) \), stands for an object of class \( c \), with fields described by the mapping \( fm \). For such an object, the fields lookup \( o(f) \) is a shorthand for \( o \downarrow_2 ( f ) \), and field update \( o[f \mapsto \kappa] \) is a shorthand for \( ( o \downarrow_1, o \downarrow_2 [ f \mapsto \kappa ] ) \).
\[
\begin{align*}
\text{var} \\
\var x, P, S, H \leadsto S(x), H \\
\text{this}, P, S, H \leadsto S(\text{this}), H \\
r, P, S, H \leadsto r, H \\
\end{align*}
\]

\[
\begin{align*}
\text{fld} \\
e, P, S, H \leadsto \iota, H' \\
e, P, S, H \leadsto \iota, H' \\
fldAss \\
e, P, S, H \leadsto \iota, H' \\
e', P, S, H' \leadsto \kappa, H'' \\
H''(\iota)(\overline{f}) = \kappa', \\
H''' = H''[\iota \mapsto H''(\iota)[\overline{f} \mapsto \kappa]] \\
e.f := e', P, H, S \leadsto \kappa, H''' \\
\end{align*}
\]

\[
\begin{align*}
\text{call} \\
e, P, S, H \leadsto \iota, H_1 \\
e_1, P_1, S_1, H_1 \leadsto \kappa_2, H_2 \\
H_2(\iota) = (c, \ldots) \\
M(P, c, m) = (\varnothing, \varnothing, e) \\
S'' = \text{this} \mapsto \iota, x \mapsto \kappa_2 \\
e, P, S'', H_2 \leadsto \kappa, H' \\
e.m (e_1), P, S, H \leadsto \iota, H'' \\
\end{align*}
\]

\[
\begin{align*}
\text{stck} \\
e \text{ evaluates to access of non-existing field, or} \\
e \text{ evaluates to assignment of non-existing field, or} \\
e \text{ evaluates to call of non-existing method} \\
e, P, S, H \leadsto \text{stuckExc}, H \\
\end{align*}
\]

\[
\begin{align*}
\text{new} \\
\mathcal{F}_S(P, c) = \{ f_1, \ldots, f_r \} \\
i \text{ is fresh in } H \\
H' = H[\vert \varnothing \mapsto (c, f_1 \mapsto 0 \ldots, f_r \mapsto 0)] \\
\text{new } c<\sigma>, P, S, H \leadsto \iota, H' \\
\end{align*}
\]

\[
\begin{align*}
\text{prop} \\
e', P, S, H \leadsto \text{stuckExc}, H \\
e' \text{ has subexpression } e' \\
e, P, S, H \leadsto \text{stuckExc}, H \\
\end{align*}
\]

Figure A.1: Execution.
In figure A.1 we give the operational semantics of \( J_{\text{Joe}} \). The rules are standard for the operational semantics of such a small object oriented language, and are similar to previous work. Rule \textit{var} describes the lookup of parameters, the receiver, or a value. Rules \textit{fld} and \textit{fldAss} describe field lookup and field update. Rule \textit{call} describes method call: a new frame \( S \) is pushed onto the stack, which is discarded after execution of the method body \( e' \). Rule \textit{new} describes object creation, where all fields of its class are initialized with 0. Rule \textit{stk} describes stuck execution, due to a missing field or method body.

### A.3 Soundness

#### Runtime owners

We define a run-time notion of the actual contexts of a particular object. For each object at address \( \iota \), its runtime owners \( Os(\iota) \) maps each of the context parameters to its actual, runtime contexts, which are other objects.

\[
\begin{align*}
\tau & \in \text{RunOwnSubst} = \text{OwnParam} \rightarrow \mathbb{N}^+ \\
Os & \in \text{RunOwners} = \mathbb{N}^+ \rightarrow \text{RunOwnSubst}
\end{align*}
\]

The \( Os \) gives a notion of inside, defined by the judgment \( Os \vdash \iota' \prec \iota \). Furthermore, \( Os \) is well-formed, \( i.e., \vdash Os \), if the ”inside” relation it determines is acyclic, and if any object’s owner is inside all it other contexts. Furthermore, \( Os \) is well-formed with respect to a particular heap, \( i.e., H \vdash Os \), if any object \( \iota \) containing a direct reference to another object \( \iota' \), is inside the owner of \( \iota' \).

\[
\begin{align*}
\frac{Os(\iota')(\text{owner}) = \iota}{Os \vdash \iota' \prec \iota} & \quad \frac{Os(\iota) \neq \epsilon}{Os \vdash \iota \prec \iota} & \quad \frac{Os \vdash \iota' \prec \iota}{Os \vdash \iota'' \prec \iota} \\
\frac{Os \vdash \iota' \prec \iota \text{ and } Os \vdash \iota < \iota'}{\vdash \iota = \iota} & \quad \frac{Os(\iota')(\text{owner}) = \iota}{Os \vdash \iota' \prec Os(\iota')(\text{owner})} \\
\vdash Os & \quad \frac{H(\iota')(\text{f}) = \iota}{Os \vdash \iota' < Os(\iota)(\text{owner})} & \quad \frac{H \vdash Os}{\vdash Os}
\end{align*}
\]

#### Conformance

A heap, runtime owners, and stack conform to a program and environment, as defined in the judgment \( P, E \vdash Os, H, S \) in figure A.2, if the receiver and arguments point to objects.
which conform to their type declared in the environment, and all object conform to their
classes. An object conforms to a type $c$ if it is of class $c'$, where $c'$ is a subclass of $c$, and
if the object contains appropriate values for all fields of the class $c$. The values for the
fields are appropriate, if they have the appropriate type, and if their substitution is the
composition of the $\sigma$ (the substitution in the type of the object) with $\sigma'$ (the substitution
in the declared type of the filed).

Subject reduction

Soundness of the system guarantees that execution of a well typed expression in an appro-
priate stack and heap and in context of a well formed program will not get stuck, i.e., will
not attempt to access non-existing fields or methods. It does not guarantee that there will
not be a link related exception or a null pointer exception.

We first need to define what a well-formed program is and when a stack and heap conform
to a program and environment.

A program is well formed, judgment $\vdash P$ in figure A.3, if the class hierarchy forms a tree,
no field identifier is defined in a subclass and superclass, any method defined in a subclass
and a superclass will have the same signature, and all methods bodies have a signature,
and they verify to that signature.

We expect to be able to prove the following Theorem:

**Theorem 1 (Soundness)** If $P, E \vdash H, Os, S$, and $\vdash P$, and
• \(P, E \vdash e : t\), and

• \(e, P, S, H \leadsto r, H'\),

then, there exist Os' extending Os, such that

• \(P, E \vdash H', Os', S\),

• \(r \neq \text{StuckExc}\)

• if \(r = \kappa\) then \(P, H, Os \vdash \kappa \triangleleft t\)

In other words, execution of a well typed expression in the context of conforming heap and stack, and well formed program is guaranteed not to get stuck, and if it produces a value, this value is guaranteed to conform with the type of the original expression.

**Proof:** For the proof we need to construct the Os. We do this by extending the operational semantics, as done in figure A.4, where we define \(e, P, S, H, Os \leadsto_{os} t, H', Os'\). Obviously, \(e, P, S, H, Os \leadsto_{os} t, H', Os'\) implies that \(e, P, S, H \leadsto t, H'\). Also, if \(e, P, S, H \leadsto t, H'\) and if for some Os we have \(P, E \vdash H, Os, S\), then, there exists a Os', such that \(e, P, S, H, Os \leadsto_{os} t, H', Os'\) and that \(P, E \vdash H', Os', S'\)
\[
\begin{align*}
\text{tpThisX} & \quad \text{tpNewNull} \\
\text{P}, E \vdash \text{this} : E(\text{this}) & \quad \text{P}, E \vdash t \\
\text{P}, E \vdash x : E(x) & \quad \text{P}, E \vdash \text{new } t : t \\
\end{align*}
\]

\[
\begin{align*}
\text{tpFldAss} & \quad \text{tpFld} \\
\text{P}, E \vdash e : t & \quad \text{P}, E \vdash e : t \\
\mathcal{F}(P, t, f) = t' & \quad \mathcal{F}(P, t, f) = t' \\
\text{P}, E \vdash e' : t' & \quad \text{P}, E \vdash e' : t' \\
\text{P} \vdash t'' \leq t_x & \quad \text{P} \vdash t'' \leq t_x \\
\text{P}, E \vdash e.f := e' : t' & \quad \text{P}, E \vdash e.f : t'
\end{align*}
\]

\[
\begin{align*}
\text{tpMthCll} \\
\text{P}, E \vdash e : t & \quad \text{P}, E \vdash e' : t' \\
\mathcal{M}(P, t, m) = (t_x, t_r, ..) & \quad \text{P} \vdash t'' \leq t_x \\
\text{P}, E \vdash e.m(e') : t_r \\
\end{align*}
\]

\[
\begin{align*}
\text{WF.Class} \\
P(c) = \text{class} < \alpha_1, ... \alpha_n > \text{where } \rho_1, ... \rho_m \text{extc'} < \sigma' > \{ \text{fldDs } mthDs \} \\
\text{owner } \in \{ \alpha_1, ... \alpha_n \} \\
E = \rho_1, ... \rho_m, \text{this } \prec \text{owner}, \text{owner } \prec \alpha_1, ... \text{owner } \prec \alpha_n, \alpha_1 \prec \text{world }, ... \alpha_n \prec \text{world} \\
P, E \vdash c' < \sigma' > \\
t \ f \in \text{fldDs } \implies \\
\text{P}, E \vdash t & \quad \mathcal{F}(P, c' < \sigma' >, f) = \epsilon \\
t \ m ( \ t' x \ ) \ \{ \epsilon \} \ \in \ mthDs \ \implies \\
\text{P}, E \vdash t & \quad \text{P}, E \vdash t' \\
\text{P}, E \vdash t'' \leq t & \quad \mathcal{M}(P, c' < \sigma' >, m) \in \{ \epsilon, (t', e', t) \} \ \text{for some } e' \\
\text{P} \vdash c
\end{align*}
\]

\[
\begin{align*}
\text{WF.Program} \\
P(c) \neq \epsilon & \implies \text{P} \vdash c \\
\text{P} \vdash t \leq t', \ \text{and } \text{P} \vdash t' \leq t & \implies t = t' \\
\vdash \text{P}
\end{align*}
\]

Figure A.3: Types, and Well-formed Program.
\[
\text{var} \quad x, P, S, H, Os \leadsto_{os} S(x), H, Os
\]
\[
\text{this, } P, S, H, Os \leadsto_{os} S(\text{this}), H, Os
\]
\[
r, P, S, H, Os \leadsto_{os} r, H, Os
\]
\[
\text{fld} \quad e, P, S, H, Os \leadsto_{os} t, P', S'H'O'Os'
\]
\[
e.f, P, S, H, Os \leadsto_{os} H'(t)(f), P', S'H'O'Os'
\]
\[
call \quad e, P, S, H, Os \leadsto_{os} t, H_1, Os''
\]
\[
e_1, P, S_1, H_1, Os'' \leadsto_{os} \kappa_2, H_2, Os''
\]
\[
H_2(t) = (c, \ldots )
\]
\[
\mathcal{M}(P, c, m) = (_, \cdot, e)
\]
\[
S'' = \text{this} \mapsto t, x \mapsto \kappa_2
\]
\[
e, P, S'', H_2, Os'' \leadsto_{os} \kappa, H', Os'
\]
\[
e.m\ (e_1), P, S, H, Os \leadsto_{os} t, H', Os'
\]
\[
\text{stick} \quad e \quad \text{evaluates to access of non-existing field, or}
\]
\[
e \quad \text{evaluates to assignment of non-existing field, or}
\]
\[
e \quad \text{evaluates to call of non-existing method}
\]
\[
e, P, S, H, Os \leadsto_{os} \text{stuckExc}, H, Os
\]
\[
\text{fldAss} \quad e, P, S, H, Os \leadsto_{os} t, H', Os'
\]
\[
e', P', S, H', Os' \leadsto_{os} \kappa, H'', Os''
\]
\[
H''(t)(f) = \kappa', \quad H'' = H''[\nu \mapsto H''(t)[f \mapsto \kappa]]
\]
\[
e.f := e', P, H, S, Os \leadsto_{os} \kappa, H'', Os''
\]
\[
\text{new} \quad \mathcal{F}_s(P, c) = \{ f_1, \ldots f_r \}
\]
\[
i \quad \text{is fresh in } H
\]
\[
H' = H[\nu \mapsto (c, f_1 \mapsto 0 \ldots f_r \mapsto 0)]
\]
\[
Os' = Os[\nu \mapsto \tau]
\]
\[
\tau(\alpha) = (Os(S(\text{this})))\sigma(\alpha) \quad \text{if } \epsilon \neq \sigma(\alpha) \neq \text{this}
\]
\[
\tau(\alpha) = S(\text{this}) \quad \text{if } \epsilon \neq \sigma(\alpha) = \text{this}
\]
\[
\text{prop} \quad e', P, S, H, Os \leadsto_{os} \text{stuckExc}, H, Os'
\]
\[
e \quad \text{has subexpression } e'
\]
\[
e, P, S, H, Os \leadsto_{os} \text{stuckExc}, H, Os'
\]

Figure A.4: Execution with Explicit Runtime Owners.
Appendix B

Program Annotations

In a number of places in this report, we make reference to the point in code at which an object was allocated. We now present a function which, given a program, returns a program whose allocation sites have been annotated with unique reference numbers.

Firstly, an auxiliary function, fresh, gives a different unique reference on each evaluation, while the function $\mathcal{A}_n$ annotates the object allocation sides of a program $P$:

\[
\begin{align*}
\text{fresh} & : \mathbb{N} \\
\text{fresh} & \overset{\text{def}}{=} \text{fresh value on each evaluation} \\
\mathcal{A}_n : \text{Program} \rightarrow \text{AnnProgram} \\
\mathcal{A}_n(\text{class classlist}) & \overset{\text{def}}{=} \mathcal{A}_n(\text{class}) \mathcal{A}_n(\text{classlist}) \\
\mathcal{A}_n(\text{method methodlist}) & \overset{\text{def}}{=} \mathcal{A}_n(\text{method}) \mathcal{A}_n(\text{methodlist}) \\
\mathcal{A}_n(\text{class } c\{\text{fieldlist methodlist}\}) & \overset{\text{def}}{=} \text{class } c\{\text{fieldlist } \mathcal{A}_n(\text{methodlist})\} \\
\mathcal{A}_n(t \ m(t_1 \ x) \ \{\text{decls } s \ \text{return } e\}) & \overset{\text{def}}{=} t \ m(t_1 \ x) \ \{\text{decls } \mathcal{A}_n(s) \ \text{return } \mathcal{A}_n(e)\} \\
\mathcal{A}_n(\text{literal}) & \overset{\text{def}}{=} \text{literal} \\
\mathcal{A}_n(\text{new } t) & \overset{\text{def}}{=} [\text{new } t]^n, \text{where } n = \text{fresh} \\
\mathcal{A}_n(\text{lval}) & \overset{\text{def}}{=} \text{lval} \\
\mathcal{A}_n(e.m(e_1)) & \overset{\text{def}}{=} \mathcal{A}_n(e).m(\mathcal{A}_n(e_1)) \\
\mathcal{A}_n(\text{skip}) & \overset{\text{def}}{=} \text{skip} \\
\mathcal{A}_n(s \ s) & \overset{\text{def}}{=} \mathcal{A}_n(s) \mathcal{A}_n(s) \\
\mathcal{A}_n(\text{lval } e;) & \overset{\text{def}}{=} \mathcal{A}_n(\text{lval }) = \mathcal{A}_n(e) \\
\mathcal{A}_n(\text{if } e \ \text{then } \{s_1\} \ \text{else } \{s_2\}) & \overset{\text{def}}{=} \text{if } \mathcal{A}_n(e) \ \text{then } \{\mathcal{A}_n(s_1)\} \ \text{else } \{\mathcal{A}_n(s_2)\}
\end{align*}
\]