University of London
Imperial College of Science, Technology and Medicine
Department of Computing

A Model of Effects with an Application to Ownership Types

Matthew Jeremy Smith

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, May 2007

Abstract

Effects systems statically calculate the state affected or required by an execution. The effect is given in terms of some abstraction of the state of the program. The effect of an expression is given in two parts - the state which is read and that which is written to. The effect of two expressions can show them to be non-interfering and thus can be used to choose when to parallelise code. Similarly, effects can indicate when it is safe to re-order code whilst preserving the semantics.

We extend the calculation of effects to predicates and show how it is possible to calculate the effect of a predicate (the part of the program state required to calculate satisfaction). We then extend the definition of non-interference for predicates and expressions (where the execution does not affect satisfaction of the predicate). This property can be derived statically using effects and we suggest an application in verification.

We have developed a language independent model of effects. Our model is able to describe complex and abstract properties of such systems without reference to particular language features. The model includes predicates and their effects. Results proved within our model show that any language and effects system which fits the model can judge non-interference properties.

We define ODE, an effects system for Ownership Domains. Ownership Domains are an extension of Ownership Types and provide finer granularity of encapsulation but also a more liberal aliasing policy. We base our effects system on that of `Joe`. The increased precision of Ownership Domains allows us to provide more precise effects than would be possible in `Joe`. We prove that ODE is an instance of our model of effects.

Acknowledgments

Five years on from our first involvement I continue to enjoy and benefit from the supervision of Sophia Drossopoulou. Sophia has taught me a great deal, not least the importance of a good diagram! Many students would be lucky to have such a committed and active supervisor.

Susan Eisenbach is famous the world over as a behind-the-scenes mover and shaker and I am fortunate to enjoy the benefit of her guidance. From academic to pastoral and from careers to home improvement her advice is always of the highest quality.

My family have always supported my academic endeavours and I must applaud their patience and tolerance. Thank you for helping me pursue my ambitions and supporting me even when I was being petulant. I'd like to thank Debora for her support and companionship through many of the hardest parts of my studies.

I'd like to thank Dave Clarke and the SLURP team for many interesting discussions both on topic and off. Thanks also to the referees of my papers and all those who have offered me guidance and suggestions. Thanks must also go to my examiners, Sebastian Hunt and Pasquale Malacaria, for their helpful suggestions.

A great many other people deserve thanks for helping me with my PhD and making the experience enjoyable. My many good friends in and out of college have made the last four years memorable in many different ways. I will be able to look back on these years with many happy memories.

Finally the biggest thanks go to all those clever people who understood that I almost never wanted to be asked my how thesis was going. I dedicate this work to them.

Contents

List of Figures

Chapter 1

Introduction

Most major programming languages feature mutable state that may be shared between units of control or processes. Programmers make liberal use of this feature, from passing references to container structures between blocks of control, to concurrent access of buffers. Because of the ability to create multiple references to the same data it is hard to track what computations may alter data on which other computations may later rely. Programmers must take a great deal of care to avoid such behaviours.

Heap structures can become very complicated and static analysis proves difficult. In general, it is hard to determine when data is shared let alone check what shared data will be used or altered by a computation. Most mainstream languages offer little protection to prevent unwanted sharing or determine whether data is shared.

Effects systems allow an estimation to be made of the state used or altered by a computation. Comparing the *effect* of two expressions it is possible to tell, statically, that they will not affect each other's execution. Systems for encapsulation such as Ownership Domains allow state to be protected from unwanted aliasing. Because such systems provide a partitioning of the state of the program they can sometimes be used as a basis for an effects system.

1.1 Technologies

Good programming style can avoid unnecessary sharing of state and unwanted interactions. This alone is not enough to enable a static analysis to detect such behaviours. There are however technologies for controlling and describing the state that an expression can access. Effects systems were developed in the late eighties for the language FX[GJLS87]. Working similarly to a type system they statically provide a conservative approximation of the parts of the heap that may be used or changed by an expression. Type systems which provide encapsulation enforce boundaries around groups of objects, preventing certain references and thus reducing sharing of state.

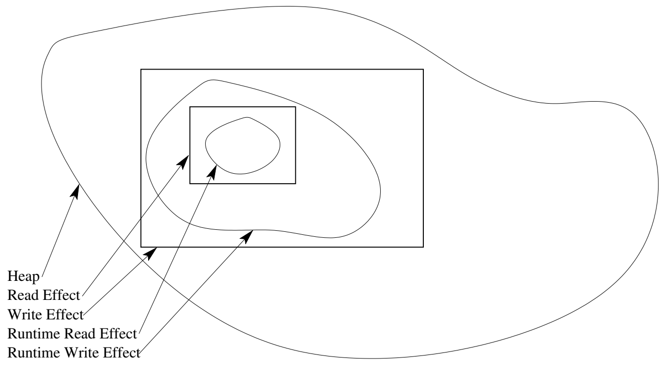


Figure 1.1: Representation of a heap with static and runtime effects

1.1.1 Effects

Whenever an expression is executed there is some part of the heap that it reads e.g. through field lookup and some part that it modifies e.g. field assignment. Effects systems provide an estimation of these sub-heaps and methods by which to compare them. Figure 1.1 represents a heap and shows the areas of the heap that execution of some expression reads and writes. Effects systems, like types, are necessarily conservative - the estimation of the sub-heap used will often be larger than that which is actually used. In Figure 1.1 the square boxes represent the calculated read and write effects - they are larger and coarser grained than the sub-heaps they approximate.

Hereinafter we shall refer to the statically calculated estimations of the heap used by an expression as the *static effect*. The *static read effect* and *static write effect* are respectively the estimation of the parts of the heap read and modified by execution. The *dynamic effect* is the precise sub-heap used by an execution. We use the terms *dynamic read effect* and *dynamic write effect* for the precise sub-heaps read and written to by execution.

Effects as represented in Figure 1.1 are in terms of the precise parts of a real heap they refer to. In practise static effects must be expressed in terms of some abstraction of the heap since in a static context we do not know exactly what the heap's structure will be. These abstractions are usually supported by some sort of encapsulation or grouping scheme which provides statically known names which correspond to sets of objects or values. Static effects can then be described in terms of these sets. If a value is read, the static effect recorded is of reading the name of the set to which the value belongs. To capture both the part of the heap read and written static effects have the form `rd` ϕ `wr` ϕ' where ϕ and ϕ' describe parts of the heap.

The nature of effects, ϕ, vary greatly from system to system. In FX[GJLS87] each variable is assigned a 'region', effects are simply sets of these regions. The effects of `Joe` describe sets of objects based on the ownership structure they inhabit (see Section 1.1.2). The object oriented effects system of Greenhouse and Boyland divides the fields of each object into groups and effects are given in terms of sets of these groups. Thus, these effects relate to portions of objects rather than sets of whole objects (as in `Joe`). We

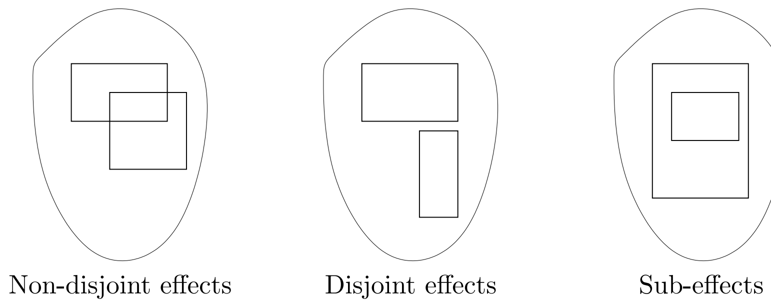


Figure 1.2: Three configurations of effects

note that effects systems are applicable in different language paradigms and also that within a paradigm they may still differ substantially in the way they describe the heap.

Notice in Figure 1.1 that in both the static and runtime cases, the write effect is contained in the read effect. This restriction is part philosophical and part practical. It is natural to consider that it is impossible to write to a value without observing it, thus if you write you must read. In the object oriented context this requirement is also helpful for modularity, see Section 2.2.2.1 for further discussion.

Comparison of effects can reveal when they refer to disjoint parts of the heap (with no values in common). The check is usually part syntactic and part semantic, based on the program and context. For example, in systems where each value belongs to precisely one group if two groups are distinct e.g. they have different names, then the effects they denote are disjoint. It is usually also possible to judge one effect to be contained in another in a relation analogous to subtypes. Figure 1.2 shows three configurations of effects: disjoint, overlapping and nested. Note that, due to the conservative nature of effects, whilst it is possible to determine statically if two effects are disjoint, the converse is not possible. Because of dynamic allocation of memory, it is quite possible for the runtime equivalent of an effect to be empty.

With the ability to judge effects disjoint, it is sometimes possible to determine that two expressions are non-interfering – that is, execution of one does not affect execution of the other. In Figure 1.3 we show static and runtime effects for two expressions (one indicated in solid lines, the other in dashed lines). They are disjoint in their write effects but not in their read effects. Thus, executing the first expression cannot affect the second - it only modifies data inside its write effect which is disjoint from the entire effect of the second. The same holds for execution of the second expression. Disjointness of the read effects is not necessary, since the data in the intersection is unchanged by either execution.

This result - effects showing non-interference of expressions - was first used in FX[GJLS87] to automatically parallelize code. `Joe` [CD02] also has this property and in the work of Greenhouse and Boyland [GB99] it was suggested that effects could be used to show when it was safe to reorder execution in a single threaded context.

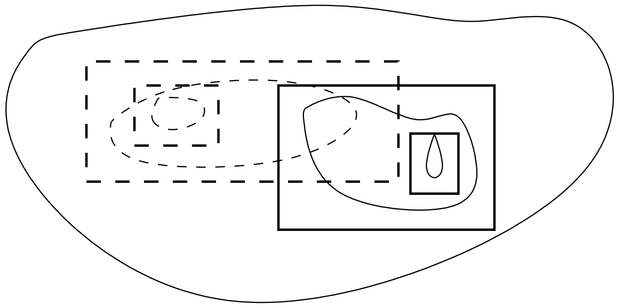


Figure 1.3: Effects of non-interferring expressions

1.1.2 Encapsulation

Fundamental to any effects system is the abstraction of the heap used. In recent years a number of systems for encapsulation have emerged which are not directly motivated by effects. Nonetheless these systems do provide some kind of abstraction of the heap by grouping related objects together. The object oriented community has produced a variety of systems such as Balloons [Alm97], Islands [Hog91], Ownership Types [CPN98] and Universes [MPH01] that provide varying styles and levels of encapsulation. As Hogg put it rather bluntly in [Hog91]:

> The big lie of object-oriented programming is that objects provide encapsulation.

Similar complaints are voiced in nearly every paper on encapsulation. The object oriented style means that the abstract value of an object may depend on the values of many other objects - often described as the object's *rep*. Reasoning about the state of an object requires knowledge of its rep. Unfortunately without some form of protection rep objects can be changed without the knowledge of their dependent. Encapsulation systems attempt to remedy or at least mitigate this situation. They seek to enforce, through static typing, boundaries of encapsulation which correspond to the intent of programmers and aid analysis. Units of encapsulation have been used to aid in program verification (as in e.g. Universes[MPH99] and Boogie [BCD$^+$05]) or exploited at runtime to aid memory control [ZNV04], to name but two areas of application.

In Figure 1.4 we give code for a standard linked list implementation in Java. The linked list is the canonical example for encapsulation in object oriented programs as it features data which should be protected (the list representation) and data which may be shared (the list elements). Following the principles of abstraction and data-hiding, the `List` class hides the implementation (in terms of `Node` objects) and provides an interface to manipulate the underlying representation (add, remove, etc.). The field `head` is private so that no external object can directly access the representation by a field access. So long as the list allocates new nodes itself, we might hope that this guarantees encapsulation, but it does not. Without applying data flow analysis or other techniques we cannot be sure that references to internal `Node` objects are not leaked, allowing arbitrary modification

```
class List<X> {                         class Node<X> {
  private Node<X> head;                   private X data;
  public void add(X x){...}               private Node<X> next;
  public boolean remove(X x){...}         ...
  public int length(){...}              }
  ...
}
```

Figure 1.4: Linked List Implementation in Java

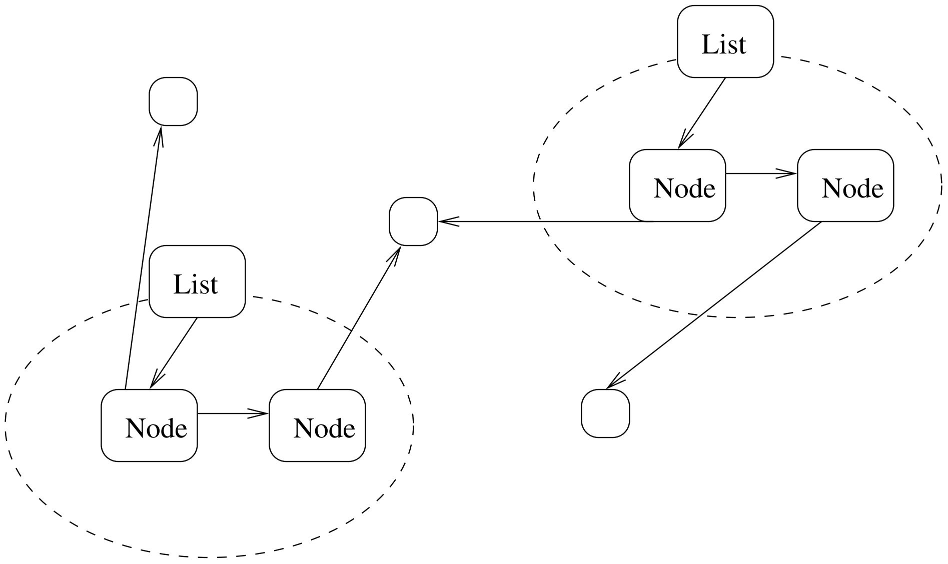
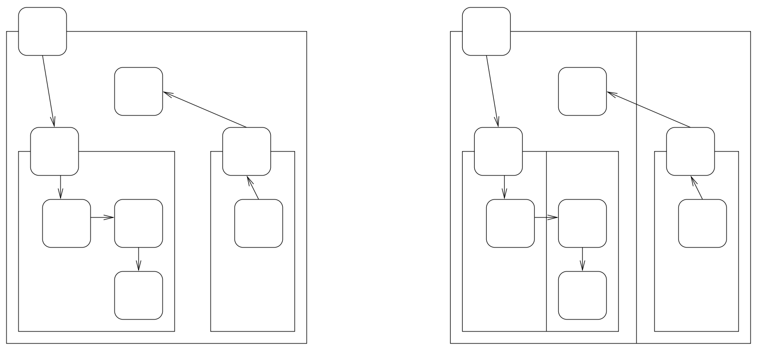


Figure 1.5: Possible encapsulation of `List` and `Node` objects

of the list.

Most object oriented encapsulation systems tackle this problem by allowing the programmer to impose boundaries around groups of objects. Restrictions are imposed on what references, from one object to another via a field, are allowed. In Figure 1.5 we show possible encapsulation boundaries with a diagram showing instances of class `List`. The representation (`Node` objects) of each list is encapsulated within the boundary defined by the corresponding `List` object. Normally such boundaries will forbid inward references which break through boundaries e.g. from one `List` object to the `Node` of another. The `List` object sits on the encapsulation boundary and can communicate with objects inside as well as outside. We usually want to be able to share data between lists so we only encapsulate the representation. In Figure 1.5 the data objects (small squares) represent the list data. They are not protected by an encapsulation boundary and so may be referenced by both lists.

In ownership based systems such as Ownership Types, Universes and Ownership Domains, every object has a unique owner (an object or other entity) which defines a boundary and governs access to the object. Each system has differing policies regarding what references are allowed. In Ownership Types no references may break into a unit of



Ownership Types Ownership Domains

Figure 1.6: A heap in ownership types and ownership domains

encapsulation (though references may break out). In Universes boundaries may be broken by read-only references (the use of which may not lead to modification of objects). Ownership Domains have a flexible, permissions based systems where the programmer may allow some references to break boundaries, and others not. What all three of these systems have in common is that the ownership structure is nested and gives the heap a tree like structure as shown in Figure 1.6. Figure 1.6 shows equivalent heaps in ownership types and ownership domains. Both feature units of encapsulation with access controlled by an owning object. In ownership domains each unit of encapsulation can be further subdivided giving a more fine-grained partition of the heap.

1.2 Contribution

We consider effects to be an under-used and poorly understood technology. We believe they provide a generic and versatile way to explicitly show information which is used implicitly in other techniques. Further we believe effects systems to have a general form and may be developed to complement existing type systems and systems of reasoning. Effects are a natural application for some encapsulation systems which do not currently have an accompanying effects system.

We extend the notion of effect to predicates of program logics. Using a similar technique to that in programs, we calculate an effect which expresses what part of the heap is required to calculate satisfaction of the predicate. By comparing the effects of predicates and expressions we suggest a rule for Hoare Logics which is related to but more general than the Rule of Constancy found in [Rey81].

As described in Section 1.1.1, a variety of effects systems exist which differ in the style of language they support and the way they describe the heap. We find there to be much in common between these apparently disparate systems. We present a model of effects systems which abstracts away from any specific language but captures the basic properties we find in all effects systems. Our model not only describes what constitutes an effects system, but what properties a programming language must have to enable

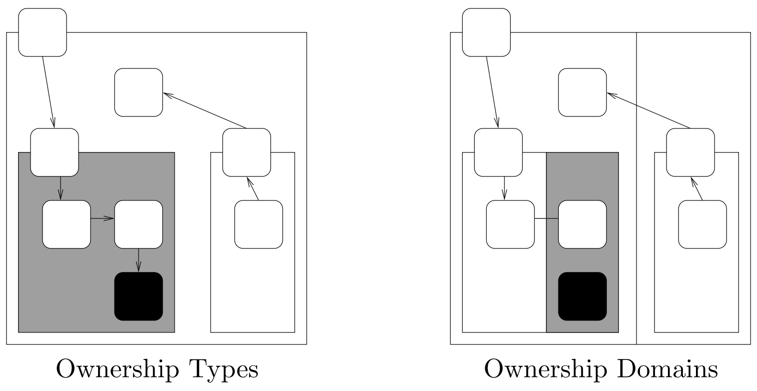


Figure 1.7: Effects in Ownership Types and Ownership Domains

them. Without any knowledge of the precise nature of the programming language our system constrains program behaviour in complex ways. We present results in our model for using effects to determine non-interference of pairs of expressions and of expressions and predicates. Thus, any effects system which can be shown to fulfill the requirements of our model can be used to determine such non-interference properties.

Finally, we present a new instance of our model in the form of an effects system for Ownership Domains. ODE (Ownership Domains with Effects) has an effects system based on that of `Joe`. ODE represents an improvement over `Joe` since ownership domains are more flexible and more precise. In Figure 1.7 we show possible effects in each. The black object represents the runtime effect and the grey area the static effect. Because of the finer-grained encapsulation of ownership domains the static effect can be more precise. We extend both Ownership Domains and the approach of `Joe` to provide an effects system which gives more precise effects and can cope with the more advanced features of Ownership Domains. We prove that ODE is an instance of our model of effects.

1.3 Thesis Structure

Chapter 2 introduces the definitions we shall use for non-interference. It also shows how effects can deduce non-interference of two expressions. Finally we suggest how effects can be calculated for predicates and how they can be used to determine non-interference. Chapter 3 presents our language-independent model of effects and presents results on non-interference using the model. Chapter 4 defines ODE, our version of Ownership Domains extended with an effects system. Chapter 5 features proofs that ODE is an instance of our model of effects. Related work is discussed in Chapter 6. In Chapter 7 we present our conclusions and suggest areas for future work.

Chapter 2

Non-Interference and Effects

In this chapter we define non-interference and show its relationship with effects. As well as giving our definition of non-interference for pairs of expressions we provide a definition of non-interference between an expression and a predicate. We motivate this definition by discussion of a possible use in program verification. Using the `Joe` system [CD02] we give examples of how effects can determine non-interference for expressions. We introduce the idea of effects for predicates with a simple predicate language for which we give `Joe`-style effects. Finally, we apply the effects for expressions and predicates to our suggested application for program verification.

In the next chapter we will use the definitions of non-interference given here in the context of our model of effects.

Parts of this chapter appeared in the paper ‘Cheaper Reasoning with Ownership Types’ [SD03] co-authored with Sophia Drossopoulou.

2.1 Non-interference

A number of definitions are possible for non-interference of expressions. We informally state our definition and discuss alternatives.

2.1.1 Expressions

In this thesis, we restrict ourselves to consideration of single-threaded programs and our definition of non-interference reflects this. We shall define two expressions as being non-interfering when, if they can be executed sequentially in one order, they can be executed in reverse order with the same results.

Whilst relatively simple this definition remains a useful notion. Being able to swap the order of execution in a meaning preserving way is of obvious use when refactoring code. Optimisations such as call aggregation [YK03] could benefit from knowledge of this type of non-interference, allowing calls separated by arbitrary amounts of code to be aggregated and called earlier.

An alternative but similar definition could be that, for a given start state, if each expression can be executed from this state then the other expression can be executed subsequently. We would require that each execution gives the same result for each expression and that the two orders of execution give the same final state. We discuss this definition further, with respect to our model, in Section 3.6.4. The notion of non-interference becomes more complex in a setting that considers concurrency as we would need to consider all possible orders and interleavings of execution.

2.1.2 Predicates

We are by no means the first to apply the idea of non-interference with respect to predicates. In [Rey81] Reynolds presents the 'Rule of Constancy' - a Hoare Logic rule of the form:

$$\frac{\Gamma \vdash \{P\}c\{Q\} \;\; c\#R}{\{P \wedge R\}c\{Q \wedge R\}}$$

where P, Q and R are predicates and c is a command of the programming language (in this case Algol). The statement $c\#R$ means c does not interfere with predicate R. The relation guarantees that execution of c does not modify any data which affects the satisfaction of R. As such we can expect that if R holds before execution of c then it will hold afterwards. Thus it can be conjoined with the pre- and post-conditions of any other Hoare sentence about c.

In Reynolds's version of this rule the relation # is a language-specific syntactic comparison like that introduced for expressions in [Rey78]. We suggest a semantic definition of non-interference for predicates and expressions in the style of that for expressions. We shall say that a predicate and expression are non-interfering if execution of the expression does not reverse satisfaction of the predicate. In order to be as flexible as possible (and so describe a variety of predicate languages) we shall assume satisfaction of predicates to be three-valued with $\perp <$ true and $\perp <$ false. Our non-interference definition will allow for satisfaction to go from $\perp$ to true or false but will not allow satisfaction to change from true or false.

We believe this definition provides a great deal of flexibility. It allows for three-valued definitions of satisfaction or two different two-valued interpretations: where $\perp$ is ignored altogether or where $\perp$ is treated as false giving $\mathsf{false} < \mathsf{true}$.

When we can judge non-interference according to our definition, we believe that the Rule of Constancy should be generally applicable to Hoare Logics. We suggest effects to judge this non-interference and thus an effects enabled Rule of Constancy.

2.1.3 The 'other' non-interference

The term non-interference is also, and possibly more famously, used by research communities interested in concurrent systems and security. Following Goguen and Meseguer in [GM82] the term has been used to describe the flow of information between security levels. To put it abstractly, two non-interfering entities (processes, units of code ...) cannot affect the output visible to each other. This definition of non-interference has been widely used in the study of concurrent systems and remains an important and evolving concept in process algebras.

This is certainly related to our definition though the notion of *visibility* is not important to us only the notion of *use*. Also the distributed setting in which this non-interference is considered means issues of information flow become dominant over issues of program state. In both cases detecting non-interference requires careful consideration of the side-effects of computation.

Since Reynolds predates Goguen and Meseguer in his use of the term by four years we shall continue to use it in our sense and as used in other effects related literature.

2.2 Effects and Non-interference

We now explore the use of effects to determine non-interference as we defined it above. First, we present an example in a Java-like language and then in `Joe` [CD02] which we will use as a basis for our discussion.

Figure 2.1 shows a simple project management example, written in a Java-like language. An `Employee` has a `Timetable` which is a linked-list containing pairs of `Project` and `Duration` objects. We propose two desirable properties for the `Timetable` of an `Employee`, `e`:

`nonO(e)` The entries in `e.t` are non-overlapping (with respect to their start and finish dates)

`durInP(e)` The duration of each entry in `e.t` is contained within the duration of the project for that entry

We expect changes to be made to a `Timetable` and as an example consider a simple mutation whereby all an employee's `Timetable` entries are delayed by some number of days. This is the purpose of the method `delay` in class `Employee`.

Suppose we wish to prove that the properties `nonO` and `durInP`, of an object `e`, are preserved when executing the `delay` method of `Employee` on `e`.

The possible side effects of performing `delay` on `e` make such assertions difficult to prove. `Employee` objects might share `Timetable` objects and thus performing delay on `e` might affect the properties of any number of other `Employee` objects. To ensure our properties

```
class Duration{
    int start;
    int end;

    void shunt(int period){
        start += period;
        end += period;
    }
}

class Employee{
    Timetable t;
    void delay(int period){
        // invokes delay on t
    }
}
```

```
class Timetable{
    Project p;
    Duration d;
    Timetable next;

    void delay(int period){
        // shunts d by period
        // and call on next
    }
}

class Project{
    Duration d;
}
```

Figure 2.1: Project Example in almost-Java

hold through execution of `delay` we would need to directly reason about all `Employee` objects which might alias `Timetable` objects referenced by `e`. For example, assume two `Employee` objects, `alice` and `bob`, which we know not to be aliases. Assume also, that for some program point, `p1` we have established that `nonO(alice.t)`, `nonO(bobt.t)`, `durInP(alice.t)` and `durInP(bob.t)`. Now we execute `alice.delay(23)`. Because of the potential for aliasing all these assertions need to be re-established not just those pertaining to `alice`. However, if we knew that `bob`'s `Timetable` was not accessible through `alice`, we could argue that `alice.delay(23)` cannot affect `bob`'s timetable and expect that the assertions `nonO(bob.t)` and `durInP(bob.t)` should be preserved through execution of `alice.delay(23)`.

Similarly, we might expect `alice.delay(23)` and `bob.delay(17)` to be non-interfering but as above the possibility of aliasing among `Timetable` objects makes this hard to establish.

2.2.1 Joe

`Joe` is a Java-like language with ownership types and effects [CD02]. We give a brief outline here to enable the following discussion. The effects of ODE given in Chapter 4 are based on those of `Joe` so may provide some clarification.

The types of `Joe` are parameterised with ownership *contexts* (p,q etc.). Classes are defined using context variables which are bound to objects or the global context, `world`, when types are instantiated. Contexts define units of encapsulation and by declaring a context to be an object's owner we encapsulate the object within the context's boundaries. Further parameters are used to describe the location of objects in other contexts e.g. in the types of fields.

In Fig. 2.2 we add ownership types (and effects) to the example of Fig. 2.1. Figure 2.3 shows a possible instantiation of classes from Fig. 2.2. The rounded boxes represent

```
class Duration<owner>{
    int start;
    int end;

    void shunt(int period)
            wr this {
        start += period;
        end += period;
    }
}

class Employee<owner, proj>{
    Timetable<this, proj> t;
    void delay(int period)
            wr under(this) {
        // invokes delay on t
    }
}
```

```
class Timetable<owner, proj>{
    Project<proj> p;
    Duration<this> d;
    Timetable<owner,proj> next;

    void delay(int period)
            wr under(this){
        // shunts d by period
        // and call on next
    }
}

class Project<owner>{
    Duration<this> d;
}
```

Figure 2.2: Project Example in with ownership types in `Joe`

objects. For example `alice` : `Employee` represents the object `alice` of class `Employee`. The square boxes in the diagram represent the ownership hierarchy, the box on the border being the owner of those immediately inside. Arrows are references i.e. fields. In ownerships systems of this type, arrows can break out of boxes (into 'higher' contexts) but not into them[1]. All the employees and projects are owned by some higher object, perhaps representing the company or department. Each `Employee` owns their `Timetable` objects, the `Timetable` objects in turn own the underlying `Duration` objects.

`Joe` differs from the original ownership types systems [CPN98] as it sometimes permits access to objects across ownership boundaries i.e. breaking into boxes. `let` statements in `Joe` allow assignment of any reachable statement to a local variable. The variable name can then be used as an ownership context and replaces `this` in types of objects from inside an ownership box. This is safe as the variable is a "short lived dynamic alias"[CD02]. The variable is only on the stack and fields must always preserve the owners as dominators property.

Two contexts are *disjoint* when they refer to different objects or one is `world` and the other isn't. Disjointness can be reasoned from type/ownership information stored in the typing environment. Disjointness of types is based on the class hierarchy and the disjointness of contexts. Types are disjoint when i) one is not a subclass of the other, or ii) any of their ownership parameters are disjoint, or iii) one is a subtype of some type which is disjoint from the other.

Since we know that each `Employee` owns its `Timetable` and each `Timetable` owns its `Duration` objects we also know that no `Timetable` object can be referenced by more than one `Employee`. Thus, changes which only affect `alice`'s `Timetable` leave `bob`'s `Timetable` unaffected. Hence the properties `nonO` and `durInP` would be preserved for

[1]Therefore the fields of `alice` may *not* point to `t3` nor to `d1`.

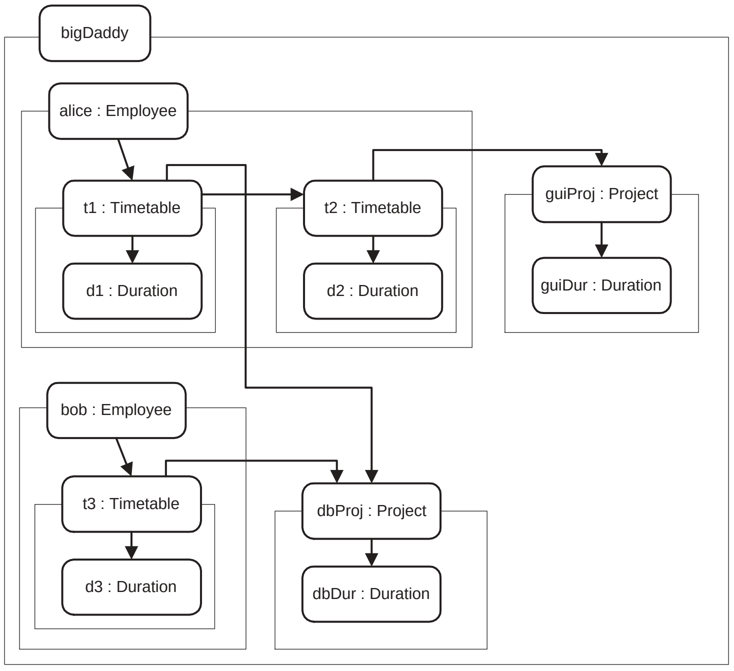


Figure 2.3: Possible instantiation of code of Fig. 2.2

`bob`.

A single judgement calculates the type and effect of an expression in `Joe` e.g. :

$$\mathtt{a}{:}\mathtt{Employee}{<}x,x\ {>}, x \prec^* x\ \vdash \mathtt{a.delay(23)}{:}\mathtt{void}\ !\ \mathtt{rd}\ \mathtt{under}(\mathtt{a})\ \ \mathtt{wr}\ \mathtt{under}(\mathtt{a})$$

where $\mathtt{a} : \mathtt{Employee}{<}x, x{>}, x \prec^* x$ is typing environment, `void` is the type of the expression and $\mathtt{rd}\ \mathtt{under}(\mathtt{alice})\ \ \mathtt{wr}\ \mathtt{under}(\mathtt{alice})$ is the effect of the expression.

Effects in Joe describe sections of the heap in terms of contexts and are based on the ownership hierarchy. An effect could be just the object bound to a context parameter p, say. The effect $\mathtt{under}(p)$ refers to the union of all the objects directly or indirectly owned by p. $\phi \cup \phi'$ is the effect derived from set union of ϕ and ϕ'.

It is possible to reason not only disjointness of types but also of effects based on some type environment E. Read $\mathsf{E} \vdash \phi \ \# \ \phi'$ as ϕ and ϕ' are disjoint. One can deduce some disjointness of context from the type environment and further disjointness of effects follows intuition from set theory and the tree structure of the ownership hierarchy. For example, if two contexts, p, q, are disjoint and have a common owner then the effects $\mathtt{under}(p)$ and $\mathtt{under}(q)$ are also disjoint.

Soundness of the effects of expressions i.e. that they describe at least the part of the state that the execution reads and writes is the fundamental result of Joe.

2.2.2 Effects for Expressions

Using the code in Figure 2.2 we now show how comparison of effects can show non-interference of expressions. Suppose:

$$\begin{aligned} \mathsf{E} \quad = \quad & \texttt{alice} : \texttt{Employee} < \texttt{bigDaddy}, \texttt{bigDaddy} >, \\ & \texttt{bob} : \texttt{Employee} < \texttt{bidDaddy}, \texttt{bigDaddy} > \end{aligned}$$

and also that we know $\mathsf{E} \vdash \texttt{alice}\#\texttt{bob}$ (which we have if we know `alice` and `bob` are not aliases).

By application of type and effect rules of [CD02] we obtain that:

$$\begin{gathered} \mathsf{E} \vdash \texttt{alice.delay}(23) : \texttt{void!rd}\,\texttt{under}(\texttt{alice})\,\texttt{wr}\,\texttt{under}(\texttt{alice}) \\ \mathsf{E} \vdash \texttt{bob.delay}(17) : \texttt{void!rd}\,\texttt{under}(\texttt{bob})\,\texttt{wr}\,\texttt{under}(\texttt{bob}) \end{gathered}$$

Since each `delay` method only alters `Timetable` objects which are owned by (and thus `under`) their respective `Employee` objects. By the effects disjointness rules of `Joe` we find that

$$\mathsf{E} \vdash \texttt{under}(\texttt{alice})\#\texttt{under}(\texttt{bob})$$

since they are i) at the same level in the ownership hierarchy (as they have the same owner) and ii) not aliases (by assumption). `under(alice)` and `under(bob)` each define subtrees below `alice` and `bob` which by the tree structuring of the heap cannot have objects in common.

Since the read effect of each expression is disjoint from the write effect of the other executing one does not change any part of the heap the other will read. Thus the order of execution can be swapped. Suppose we can execute `alice.delay(23)` followed by `bob.delay(17)`; all the parts of the heap used by `bob.delay(17)` are as they were before `alice.delay(23)` so `bob.delay(17)` can execute in the original stack and then none of the parts used by `alice.delay(23)` are changed so it can execute.

2.2.2.1 The Format of Effects

Some effects systems [CD02, GB99] require that the read effect be contained within the write effect. As Greenhouse and Boyland point out, in an object-oriented context this can be helpful. Overriding methods must have effects compatible with the methods they override. Such methods will often extend behaviour and as such record larger effects. Enforcing this requirement allows extending methods to read what is written by the originals as might be desirable in for example a subclass that implements a cache or undo feature.

We will expect this property of effects systems since we find it practically and philosophically desirable. Also, any effect system that does not have this property could easily be modified to conform. This requirement makes the judgement of non-interference more succinct as we need only check disjointness of read effects against write effects. Our model includes this requirement (encoded in property LS4). It would be easy to change although the resulting definition would be a little longer.

$$
\begin{array}{lll}
\mathit{\Pi} & ::= & \underline{p}(\mathit{arg}^*)\phi(A) \\
\mathit{\Pi s} & ::= & \overline{\mathit{\Pi}} \\
\mathit{arg} & ::= & t\ z \\
z & ::= & \texttt{this} \mid x \\
A & ::= & A_0 \wedge A_1 \mid \neg(A) \mid \texttt{ff} \mid a_0 = a_1 \mid p(\overline{z}) \mid \texttt{let}\ x = a\ \texttt{in}\ A \\
a & ::= & z \mid z.f \mid \texttt{null}
\end{array}
$$

Figure 2.4: A toy assertion language for Joe

2.2.3 Effects for Predicates

We now proceed to demonstrate how effects can be calculated for predicates and how these effects may be applied to deducing non-interference. As a proof of concept we give a toy language of assertions for Joe and rules for calculating effects. We then show an application using our effects and the Rule of Constancy.

2.2.3.1 Assertions

Practical assertion languages for object oriented languages [Mül01, LPC$^+$02] tend to be complex. The complexity arises from the need to support reasoning in the presence of aliasing and language features such as inheritance. Due to these complexities we do not wish to consider any particular system in detail so in Figure 2.4 we present a toy assertion language to discuss the application of effects to predicates. We note that as a matter of necessity any assertion language must bear some resemblance to the related programming language - in order to refer to the program data. It is this similarity that will allow us to calculate effects for predicates in the same style as for the language.

We suggest assertions based on first order logic with definable predicates. $\mathit{\Pi}$, a predicate declaration, consists of a name, p, a list of arguments with their types, an effect ϕ, and a predicate body, A. ϕ is the declared effect of a predicate, in terms of the ownership parameters of the formal parameters and the formal parameters themselves. This declared effect is analogous with the declared effect of methods in `Joe` and declared types in declaratively typed languages. `this` refers to the current receiver, x ranges over local program variables and predicate parameters and f ranges over field identifiers. Assertion expressions, a, are a subset of `Joe` expressions without side-effects. $a_0 = a_1$ states that the two expressions are equal i.e. they refer to the same heap location (we can easily allow other operators). Conjunction and negation are as one expects. p ranges over predicate names and $p\ (\overline{z})$ is the application of a predicate to a set of parameters.

We employ the `let` syntax used in `Joe` to allow assertions to examine objects inside ownership boundaries. This is safe not only by virtue of safety of `Joe` but intuitively as our assertions have no computational effect - as no reassignment is occurring no ownership properties can be compromised.

As noted previously our assertions lack the supporting machinery that most object-oriented program logics require. However in terms of expressivity we believe we have

$$\frac{\mathsf{E} \vdash A_0 : \phi_0 \qquad \mathsf{E} \vdash A_1 : \phi_1}{\mathsf{E} \vdash A_0 \land A_1 : \phi_0 \cup \phi_1} \text{ ASS-CONJ} \qquad \frac{\mathsf{E} \vdash A : \phi}{\mathsf{E} \vdash \neg(A) : \phi} \text{ ASS-NEG}$$

$$\frac{\begin{array}{c} p(\overline{t\ x})\phi(A) \in \mathit{\Pi s} \\ \mathsf{E} \vdash \overline{z} : \sigma(\overline{t})\overline{!\mathtt{rd}\,\phi\ \mathtt{wr}\,\phi'} \end{array}}{\mathsf{E} \vdash p(\overline{z}) : \sigma[\overline{x} \mapsto \overline{z}](\phi)} \text{ ASS-PRED} \qquad \frac{\begin{array}{c} \mathsf{E} \vdash a_0 : t_0!\mathtt{rd}\,\phi_0 \\ \mathsf{E} \vdash a_1 : t_1!\mathtt{rd}\,\phi_1 \end{array}}{\mathsf{E} \vdash a_0 = a_1 : \phi_0 \cup \phi_1} \text{ ASS-EQ}$$

$$\frac{\begin{array}{c} \mathsf{E} \vdash a : t\ !\,\mathtt{rd}\,\phi_0\ \ \mathtt{wr}\,\phi_1 \\ \mathsf{E}, x : t \vdash A : \phi' \\ \mathsf{E}, x : t \vdash \phi' \cup \phi_0 \sqsubseteq \phi \quad \mathsf{E} \vdash \phi \end{array}}{\mathsf{E} \vdash \mathtt{let}\ x = a\ \mathtt{in}\ A : \phi} \text{ ASS-LET}$$

Figure 2.5: Inference rules for assertion effects

offered a useful and sensible subset. The basic operations of selection and comparison, along with the standard logical connectives we feel we present a proof of concept for application of effects to meaningful predicates. We discuss treatment of predicates with quantification in Section 3.6.5.

The following predicate definition, declared with respect to the code of Fig. 2.2 captures `nonO` from Sec. 2.2. It exploits recursion to traverse the list of `Timetable`. For convenience we assumed that the entries in `t` are ordered. We omit the details of the effect ϕ for now.

```
nonO(Timetable<o,po>t)ϕ(
    let x = null in ¬(¬(t=null)∧
        ¬(let dur = t.d in
            let next = t.next in
                let ndur = next.dur in
                    let start = ndur.start in
                        let end = dur.end in
                            end≤start  ∧ nonO(next))))
```

2.2.3.2 Effects of Assertions

The judgement $\mathsf{E} \vdash A{:}\phi$ states that satisfaction of the assertion, A, depends on the part of the heap characterised by ϕ, with respect to the type environment E. We take the program `P` and the predicate declarations $\mathit{\Pi s}$ as implicit. Figure 2.5 gives the rules. The rules for calculating the effect of an assertion are similar to those of `Joe`. In rules ASS-EQ, ASS-PRED and ASS-LET we use the types and effects judgement of `Joe` (as described in Sec. 2.2.1) assertion expressions, a, since these are valid `Joe` syntax. When using the Joe rules we pick the read part of the effect since this corresponds to the heap required to determine satisfaction (also by construction the write part will be empty).

The effect of a predicate is declared in terms of the declared contexts of its parameters and the parameters themselves. The effect of a predicate application is the declared effect

$$\frac{\overline{t\ x}, constr(\overline{t\ x}) \vdash B : \phi' \qquad \overline{t\ x}, constr(\overline{t\ x}) \vdash \phi' \sqsubseteq \phi}{\vdash p(\overline{t\ x})\phi(B)}$$

Figure 2.6: Rule for checking validity of predicate declared effect

under the substitution which maps the ownership variables of the predicate declaration to those of the supplied arguments and the formal arguments to the actual parameters.

An assertion, A, and the predicate definitions, Πs, are well-formed with respect to a type environment (and also implicitly the program and the predicate declarations) if all assertion expressions a, are well-formed, and types of arguments to predicates match the declared types. Using the type system of `Joe` to find the effect of a_0 and a_1 has the added benefit that assertions which have an effect are 'well-formed' i.e. expressions of the form $z.f$ are type correct and thus we needn't check elsewhere that field, f exists.

Well-formedness of predicate definitions must be checked separately to ensure the declared effect is valid for the body of the predicate. Figure 2.6 gives the rule for checking predicates. This is a direct analogue of type or effect checking of method bodies against the declarations. A predicate declaration is well-formed if i) the type environment generated from the declared types of its parameters is valid, ii) the effect of its body, in the context of the constructed environment is well-formed with effect ϕ', and iii) the calculated effect of the body is a sub-effect of the declared effect (which also requires that ϕ is well-formed with respect to the environment). The type environment is as in `Joe` and includes the type of each parameter and the ownership constraints generated by the types.

2.2.3.3 Example

The effect `under(o)` would be a valid effect for the predicate `non0` defined previously[2]. In `non0` comparisons of variables and `null` have no effect (as the comparison does not involve the heap). Effect is accumulated by the field accesses e.g. `t.next`, `next.dur` and the recursive appeal to `non0`. All the field accesses are performed on `Timetable` objects which, are all owned by `o`. Thus all field accesses are covered by `under(o)`.

2.2.3.4 Soundness of Effects for Predicates

Soundness of effects for expressions requires that the portion of the heap described by the effects is at least that read or written by execution of the expression. A consequence of this is that execution of the expression in the heap described by effects gives the same result and modifies the heap in the same ways as execution in the whole heap.

For the effects of a predicate we expect something similar, that is that satisfaction of the predicate is the same in the sub-heap described by the effect as it is in the whole heap.

[2]Since `ints` are passed by value in Java we assume they are `under` the object referencing them.

2.2.3.5 Application to Reasoning

Suppose there exists a Hoare Logic for the language with sentences of the form:

$$\mathsf{E} \vdash A_0 \ \{e\ \} A_1$$

where E is a typing environment, binding `this` and local variables to types. The program z, and predicate definitions Πs are taken to be implicit.

We argue that the following rule could be added to such a Hoare Logic.

$$\frac{\mathsf{E} \vdash A_0\ \{\mathtt{e}\ \} A_1 \qquad \mathsf{E} \vdash A : \phi \qquad \mathsf{E}\ \vdash \mathtt{e} : t\ \texttt{!}\,\texttt{rd}\,\phi_0\ \ \texttt{wr}\,\phi_1 \qquad \mathsf{E} \vdash \phi\ \#\ \phi_1}{\mathsf{E} \vdash_O A_0\ \ \wedge A\ \{\mathtt{e}\ \} A_1\ \ \wedge A}$$

The first judgement, $\mathsf{E} \vdash A_0\ \{\mathtt{e}\ \} A_1$, is a derivation of the assumed Hoare logic. The second judgement, $\mathsf{E} \vdash A{:}\phi$, is the calculation of the effect of the assertion, A, as described in Sec. 2.2.3.2. $\mathsf{E}\ \vdash \mathtt{e} : t\ \texttt{!}\,\texttt{rd}\,\phi_0\ \ \texttt{wr}\,\phi_1$ is the types and effects judgement of `Joe` as described in Sec. 2.2.1. $\mathsf{E} \vdash \phi\ \#\ \phi_1$ is the disjointness judgement described in Sec. 2.2.1. Thus, the Hoare logic rule we propose states that any assertion which is true before the execution of an expression and which inhabits a part of the heap which is not written to by the expression, is true after execution of the expression.

2.2.4 Example

We shall now show how one might apply our suggested Hoare rule in the context of the example of Fig. 2.2.

Consider the set of predicate definitions in Figure 2.7 (we abbreviate `let` statements to more Java-like syntax in the knowledge we can express these predicates in our assertion language), intended to capture the properties discussed in Sec. 2.2:

By application of the rules of [CD02] and those of Sec. 2.2.3.2 we obtain

$$\begin{aligned}
&\mathsf{E} \vdash \texttt{nonO}(\texttt{alice}) : \texttt{under}(\texttt{alice})\\
&\mathsf{E} \vdash \texttt{durInP}(\texttt{alice}) : \texttt{under}(\texttt{bigDaddy})\\
&\mathsf{E} \vdash \texttt{nonO}(\texttt{bob}) : \texttt{under}(\texttt{bob})\\
&\mathsf{E} \vdash \texttt{durInP}(\texttt{bob}) : \texttt{under}(\texttt{bigDaddy})
\end{aligned}$$

Therefore, by application of the Hoare logic extension rule, we obtain that:

$$\mathsf{E} \vdash_O \texttt{nonO}(\texttt{bob})\{\ \texttt{alice.delay}(23)\ \}\texttt{nonO}(\texttt{bob})$$

Furthermore, if we were able to derive (the hard way) that:

$$\mathsf{E} \vdash \texttt{nonO}(\texttt{alice})\ \{\ \texttt{alice.delay}(23)\}\ \texttt{nonO}(\texttt{alice})$$

```
Πs   =
            within(Duration<o1> d1, Duration<o2> d2) under(o1) ∪ under(o2)
                ( d1.start ≥ d2.start ∧ d1.end ≤ d2.end )

            nonO(Timetable<o, po> t) under(o)
                ( t.next ≠ null⇒
                    t.d.end ≤ t.next.d.start ∧ nonO(t.next) )

            durInP(Timetable<o,op> t) under(op)
                ( t.next ≠ null⇒ within(t.d,t.p.d) ∧ durInP(t.next) )

            nonO(Employee<o,op> e) under(e)
                ( nonO(e.t) )

            durInP(Employee<o,op> e) under(op)
                ( durInP(e.t) )
```

Figure 2.7: Predicate definitions for example in Section 2.2

then we could also obtain (the cheap way) through the extension rule that:

$$\mathsf{E} \vdash_O \texttt{nonO(bob)} \wedge \texttt{nonO(alice)} \ \{ \ \texttt{alice.delay(23)} \ \} \\ \texttt{nonO(bob)} \wedge \texttt{nonO(alice)}$$

So, we were able to obtain the preservation of the assertion `nonO(bob)` after the execution of `alice.delay(23)` through mechanisms of the type system.

Note, however, that we are not able to deduce that:

$$\mathsf{E} \vdash_O \texttt{durInP(bob)}\{\texttt{alice.delay(23)}\}\texttt{durInP(bob)}$$

even though we know `alice.delay(23)` only shifts the timetable of `alice` and thus does not affect the validity of `durInP(bob)`. This is so because in Joe the effects are too coarse to distinguish this situation. In Chapter 4 we describe a similar but more precise effects system based on Ownership Domains.

Chapter 3

Model

In this chapter we describe our model of effects. Our model captures an abstract set of requirements for effects systems which can judge non-interference of expressions and predicates. Our main definitions are of heap structures, programming languages, non-interference of expressions and predicates and programming languages with effects. We state and prove theorems on non-interference of expressions and predicates and non-interference of two expressions.

Our definitions are such that we identify only the key properties we require and do not describe any structure in detail. In order to apply our system we need only know that a language and effects system satisfy these basic properties. Our approach also allows us to be modular allowing us, for example, to consider multiple effects systems for one language.

Notation For ease of presentation we omit quantification in general, with variables being implicitly universal unless explicitly stated otherwise. For example $\mathsf{h} * \mathsf{h}' = \mathsf{h} \implies \mathsf{h}' = \epsilon$ is a shorthand for $\forall \mathsf{h}, \mathsf{h}' : \mathsf{h} * \mathsf{h}' = \mathsf{h} \implies \mathsf{h}' = \epsilon$

3.1 Heaps

We assume the semantics of a language to be described in terms of an operational semantics using stacks and heaps. We require the following properties of heaps:

Definition 3.1.1 *A tuple*

$$\mathsf{H} = (\mathsf{HC}, \epsilon, *)$$

consisting of a set, an element of the set and a partial operator on the set is a structure of separable heap structure *if the following axioms are obeyed*

SH1 $\mathsf{h} * \mathsf{h}' = \mathsf{h}' * \mathsf{h}$

SH2 $(\mathsf{h} * \mathsf{h}') * \mathsf{h}'' = \mathsf{h} * (\mathsf{h}' * \mathsf{h}'')$

SH3 $\mathsf{h} * \mathsf{h}' = \mathsf{h} \iff \mathsf{h}' = \epsilon$

SH4 $\mathsf{h}_1 \sqsubseteq \mathsf{h}_2 * \mathsf{h}_3 \implies \exists \mathsf{h}', \mathsf{h}''.\ \mathsf{h}' \sqsubseteq \mathsf{h}_2\ ,\ \mathsf{h}'' \sqsubseteq \mathsf{h}_3\ ,\ \mathsf{h}_1 = \mathsf{h}' * \mathsf{h}''$

SH5 $\mathsf{h}\#\mathsf{h}'\ ,\ \mathsf{h}'' \sqsubseteq \mathsf{h}\ ,\ \mathsf{h}'' \sqsubseteq \mathsf{h}' \implies \mathsf{h}'' = \epsilon$

SH6 $\mathsf{h}_1\#\mathsf{h}_2\ ,\ \mathsf{h}_1\#\mathsf{h}_3\ ,\ \mathsf{h}_2\#\mathsf{h}_3 \implies \mathsf{h}_1 * \mathsf{h}_2\#\mathsf{h}_3$

where (for convenience)

$$\begin{array}{lll} \# \subseteq \mathsf{HC} \times \mathsf{HC} & \textit{with} & \mathsf{h}\#\mathsf{h}' \Leftrightarrow \mathsf{h} * \mathsf{h}' \textit{ is defined} \\ \sqsubseteq \subseteq \mathsf{HC} \times \mathsf{HC} & \textit{with} & \mathsf{h} \sqsubseteq \mathsf{h}' \Leftrightarrow \exists \mathsf{h}''\ .\ \mathsf{h}' = \mathsf{h} * \mathsf{h}'' \end{array}$$

and h, h', h_1, $\ldots \in \mathsf{HC}$

In the above definition, $\mathsf{h}\#\mathsf{h}'$ states that h and h' are *disjoint*. $\mathsf{h} * \mathsf{h}'$ is the heap formed by joining the disjoint heaps h and h'. If $\mathsf{h} = \mathsf{h}' * \mathsf{h}''$ we might also say that h can be *split* into h' and h''. h is a *sub-heap* of h' ($\mathsf{h} \sqsubseteq \mathsf{h}'$) if h' can be split into h and some other heap. The axioms, **SH1** – **SH6** are to be interpreted as follows:

- **SH1** states commutativity of $*$.
- **SH2** states associativity of $*$.
- **SH3** gives semantics for ϵ, the empty heap. ϵ is an identity for $*$ and can be joined to (is disjoint from) all heaps.
- **SH4** states that, if h_1 is a sub-heap of $\mathsf{h}_2 * \mathsf{h}_3$ then h_1 can be split into sub-heaps of h_2 and h_3.
- **SH5** states that if two heaps are disjoint then they have no (non-empty) sub-heaps in common.
- **SH6** states that, for three mutually disjoint heaps, a composition of two is disjoint from the third. Alternatively, three mutually disjoint heaps can be composed.

Note that our axioms do not fully specify the nature of disjoint heaps. One could have expected that any two heaps are either disjoint or have a non-empty sub-heap in common, *a la* set disjointness. In fact this is too strong for our purposes. Consider a simple heap which maps integers (addresses) to integers (values). We wish to join ($*$) such heaps with simple union. Heap $(1 \mapsto n)$ should not be disjoint from $(1 \mapsto n')$, since $*$ would not be defined as the domains are not set disjoint. Moreover, unless $n = n'$, they have no common sub heaps other than the empty heap. We do not solve this ambiguity by considering heaps as functions as we intend to allow for a variety of types of heaps and ways of splitting them.

For separable heaps $\mathsf{H} = (\mathsf{HC}, *, \epsilon)$ (where from now on we will write $\mathsf{h} \in \mathsf{H}$ for $\mathsf{h} \in \mathsf{HC}$) we have the following Lemmas and auxiliary definitions:

Lemma 3.1.1 $\mathsf{h}_1 \sqsubseteq \mathsf{h}_2 * \mathsf{h}_3 \quad , \mathsf{h}_1 \# \mathsf{h}_3 \quad \Longrightarrow \mathsf{h}_1 \sqsubseteq \mathsf{h}_2$

Lemma 3.1.2 $\mathsf{h}_1 * \mathsf{h}_2 \# \mathsf{h}_3 \quad \Longrightarrow \mathsf{h}_1 \# \mathsf{h}_3 \quad , \mathsf{h}_2 \# \mathsf{h}_3$

Lemma 3.1.3 $\mathsf{h} = \mathsf{h}_1 * \mathsf{h}_2 = \mathsf{h}_1 * \mathsf{h}_3 \quad \Longrightarrow \quad \mathsf{h}_2 = \mathsf{h}_3$

For notational convenience we define the intersection of two heaps. This is only defined for two heaps with a common super-heap.

Definition 3.1.2 *For* h_1, h_2 *such that there exists* h *with* $\mathsf{h}_1 \sqsubseteq \mathsf{h}$ *and* $\mathsf{h}_2 \sqsubseteq \mathsf{h}$

$$\mathsf{h}' = \mathsf{h}_1 \sqcap \mathsf{h}_2 \iff \forall \mathsf{h}''.\ \mathsf{h}'' \sqsubseteq \mathsf{h}' \ \Leftrightarrow\ \mathsf{h}'' \sqsubseteq \mathsf{h}_1\ ,\ \mathsf{h}'' \sqsubseteq \mathsf{h}_2$$

Lemma 3.1.4 *For all* h_1, h_2, h *such that* $\mathsf{h}_1, \mathsf{h}_2 \sqsubseteq \mathsf{h}$

$$\begin{array}{ll} \textit{(i)} & \mathsf{h}_1 \sqcap \mathsf{h}_2 \ \textit{is defined} \\ \textit{(ii)} & \mathsf{h}_1 \sqcap \mathsf{h}_2 \ \textit{is unique} \\ \textit{(iii)} & \mathsf{h}_1 \sqcap \mathsf{h}_2 = \mathsf{h}_2 \sqcap \mathsf{h}_1 \end{array}$$

Lemma 3.1.5 $\mathsf{h} \sqsubseteq \mathsf{h}_1 * \ldots * \mathsf{h}_\mathsf{n} \quad \Longrightarrow \quad \mathsf{h} = (\mathsf{h} \sqcap \mathsf{h}_1) * \ldots * (\mathsf{h} \sqcap \mathsf{h}_\mathsf{n})$

Lemma 3.1.6 $\mathsf{h} \sqsubseteq \mathsf{h}' \quad \Longrightarrow \quad \mathsf{h} \sqcap \mathsf{h}' = \mathsf{h}$

3.2 Programming Languages

We assume programming languages to be defined by sets of syntactic constructs (programs, expressions, ...), sets of configurations (stacks, heaps, values) and judgements thereon (operational semantics, well-formedness, ...). This is consistent with common language formalisms.

We also assume a language to have associated with it a language of predicates and a function which gives satisfaction of a predicate for a given runtime configuration (heap and stack).

We exploit separability of heaps in order to describe semantics for languages whilst remaining abstract. We will appeal to the reader's intuition to justify these definitions. We emphasise that the justifications we give relate to the motivation behind the axioms. Stricter mathematical interpretations may be possible.

Definition 3.2.1 *For a structure of separable heaps,* $\mathsf{H} = (\mathsf{HC}, \epsilon, *)$*, a language description* L *is a tuple*

$$\mathsf{L} = (\Pi, \mathsf{E}, \mathsf{S}, \mathsf{H}, \mathsf{V}, \mathsf{P}, \Gamma, \rightsquigarrow, \vdash_\gamma, \vdash_\mathsf{e}, \vdash_\mathsf{c}, [\![]\!]_\mathsf{p}, \models)$$

obeying the following axioms

L1 $\pi \vdash \gamma\ ,\ \gamma \vdash \mathsf{e}\ ,\ \gamma \vdash \mathsf{s},\mathsf{h}\ ,\ \mathsf{e},\mathsf{s},\mathsf{h} \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v},\mathsf{h}''\ \Rightarrow\ \gamma \vdash \mathsf{s},\mathsf{h}''$

L2 $\mathsf{e},\mathsf{s},\mathsf{h} \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v},\mathsf{h}''\ ,\ \mathsf{h}''\#\mathsf{h}'''\ \Rightarrow\ \mathsf{h}\#\mathsf{h}'''$

L3 $\mathsf{e},\mathsf{s},\mathsf{h} \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v},\mathsf{h}''\ \Rightarrow\ \exists \mathsf{h}_\mathsf{r},\mathsf{h}'''.\ \mathsf{h} = \mathsf{h}' * \mathsf{h}_\mathsf{r}\ ,\ \mathsf{h}'' = \mathsf{h}''' * \mathsf{h}_\mathsf{r}\ ,\ \mathsf{e},\mathsf{s},\mathsf{h}' \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v},\mathsf{h}'''$

L4
$$\begin{array}{l} \mathsf{s} \models \mathsf{h}\ ,\ \mathsf{e},\mathsf{s},\mathsf{h}_1 * \mathsf{h}_2 \overset{\mathsf{h}}{\rightsquigarrow}_\pi \mathsf{v},\mathsf{h}_1 * \mathsf{h}'_2 * \mathsf{h}_3\ , \\ \mathsf{e}',\mathsf{s},\mathsf{h}_1 * \mathsf{h}'_2 * \mathsf{h}_3 \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v}',\mathsf{h}''\ ,\ \mathsf{h}_3\#\mathsf{h}_2 \\ \Rightarrow\ \mathsf{h}_3 \sqcap \mathsf{h}' \neq \epsilon\ \Rightarrow\ \mathsf{h}'_2 \sqcap \mathsf{h}' \neq \epsilon \end{array}$$

L5 $\gamma \vdash \mathsf{s},\mathsf{h}\ \Longrightarrow\ \mathsf{s} \models \mathsf{h}$

where $\pi \in \Pi$*, programs,* $\mathsf{e} \in \mathsf{E}$*, expressions,* $\mathsf{s} \in \mathsf{S}$*, stacks,* $\mathsf{h} \in \mathsf{H}$*, separable heaps,* $\mathsf{v} \in \mathsf{V}$*, values,* $\mathsf{p} \in \mathsf{P}$*, predicates,* $\gamma \in \Gamma$*, environments and*

$$\begin{gathered} \rightsquigarrow\ \subseteq \mathsf{E} \times \mathsf{S} \times \mathsf{H} \times \mathsf{H} \times \Pi \times \mathsf{V} \times \mathsf{H} \\ \vdash_\gamma\ \subseteq \Pi \times \Gamma \\ \vdash_\mathsf{e}\ \subseteq \Gamma \times \mathsf{E} \\ \vdash_\mathsf{c}\ \subseteq \Gamma \times \mathsf{S} \times \mathsf{H} \\ [\![]\!]_\mathsf{p} : \mathsf{S} \times \mathsf{H} \times \mathsf{P} \to \{\mathsf{true},\mathsf{false},\bot\} \end{gathered}$$

and $(\{\mathsf{true},\mathsf{false},\bot\},\preceq)$ *is a partial order where* $\bot \preceq \mathsf{false}$ *and* $\bot \preceq \mathsf{true}$.

We use the following shorthands

$$\begin{gathered} \mathsf{e},\mathsf{s},\mathsf{h} \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v},\mathsf{h}''\ \textit{for}\ (\mathsf{e},\mathsf{s},\mathsf{h},\mathsf{h}',\pi,\mathsf{v},\mathsf{h}'') \in\ \rightsquigarrow \\ \pi \vdash \gamma\ \textit{for}\ (\pi,\gamma) \in \vdash_\gamma \\ \gamma \vdash \mathsf{e}\ \textit{for}\ (\gamma,\mathsf{e}) \in \vdash_\mathsf{e} \\ \gamma \vdash \mathsf{s},\mathsf{h}\ \textit{for}\ (\gamma,\mathsf{s},\mathsf{h}) \in \vdash_\mathsf{c} \\ [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}}\ \textit{for}\ [\![]\!]_\mathsf{p}(\mathsf{s},\mathsf{h},\mathsf{p}) \end{gathered}$$

For a language description as above:

- $\mathsf{e},\mathsf{s},\mathsf{h} \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v},\mathsf{h}''$ captures the operational semantics of the language i.e. expression e, executed on stack s and heap h, with respect to program π give the result v and modifies the heap to h''. The execution *requires* the heap h', a sub-heap of h. Further explanations are offered in the commentaries on the axioms.
- $\pi \vdash \gamma$ states that the environment, γ, is well-formed for the program, π.
- $\gamma \vdash \mathsf{e}$ states that the expression e, is well-formed in the environment γ
- $\gamma \vdash \mathsf{s},\mathsf{h}$ states that a configuration (a stack and a heap) is well-formed in the environment γ

- $[\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}}$ gives satisfaction of the predicate, p, in h,s. $[\![]\!]_{\mathsf{p}}$ is total but we allow for satisfaction to be undetermined ($\perp$). This definition allows us to consider a larger family of predicate definitions than just mapping to true and false. In concrete instances the predicate satisfaction function will often be found by taking the least fixed point of a monotonically increasing sequence of functions. Were we to use only true and false with the order $\mathsf{false} \prec \mathsf{true}$ then we might have to restrict predicates, to be even[1] [BPP03] for example, in order to get monotonicity. We prefer to be as general as possible. $\perp$ may be interpreted as a synonym for false.

- The judgement $\mathsf{s} \models \mathsf{h}$, to be read as '$\mathsf{h}$ is *complete* in s' captures what might best be described as a 'closure' property of the heap. For any complete heap, adding on some extra heap should not change what executions can occur in that heap. This would hold if, say, h contains no dangling pointers.

L1 gives a rudimentary notion of soundness for the language. If e, s, h are all well-formed in γ (and γ is well-formed in π) and execution of e in this configuration terminates, then the resultant configuration, s, h', is also well-formed in γ.
L2 requires that, for any execution, if a heap is disjoint from the final heap, it is disjoint from the initial (as illustrated in Figure 3.1). Intuitively this states that heaps can only grow (and be modified) under execution, sub-heaps cannot be removed. Suppose some sub-heap of the initial heap were deallocated in the execution. We would expect this to be disjoint from the final heap (provided it has not been reallocated) but it would not be disjoint from the initial heap, by axiom **SH4**.
L3 gives a semantics for the *required* heap, h'. h' should be interpreted as a sub-heap which contains all that is needed for execution to succeed. We do not need to be so strong as to state that h' is minimal, though in most language instances it probably would be (by construction). Consider Figure 3.2 (where we ommit the stack and expression). We require that h' is a sub-heap of the initial heap, h and that h' characterises a 'remainder', h_r which is unused, and hence unmodified, by the execution. We show parts of the heap that may have changed in grey. Notice that only parts of h', or new parts of h'', not in h, can change. Execution on h' succeeds with the same result and has the same required heap. The resulting heap is the same as in the original execution, less the remainder. The required heap is not a standard feature of operational semantics, though similar constructs do exist in the literature[CD02, BPP03]. It should be straightforward to add this to any operational semantics definition, as we shall demonstrate in Chapter 4.
L4 gives the semantics of complete heaps. During execution on a complete heap, any new sub-heap allocated during an execution is reachable only though the portion of the heap that was modified. In **L4** we capture this property by considering consecutive executions. In the first execution the heaps before and after execution are split as follows (see Figure 3.3): h_1 relates to the portion of the heap unmodified by execution (but possibly read) and h_2 refers to the portion of the heap which will be changed (note that h_2 need not be precise, it may be larger than the heap which is strictly modified, furthermore, h may have parts in both h_1 and h_2). In the post heap we still have h_1 (it is unchanged). We

[1]atoms appear under an even number of negations

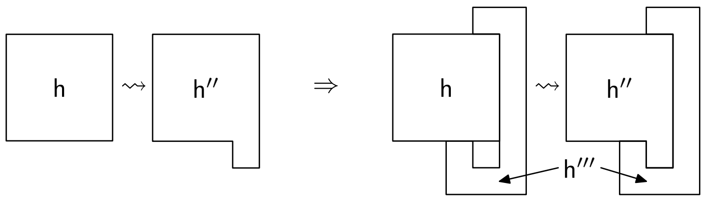


Figure 3.1: Heaps and Executions in L2

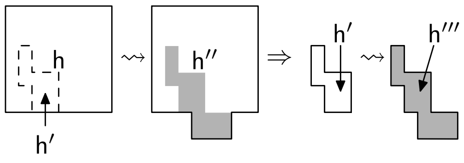


Figure 3.2: Heaps and Executions in L3

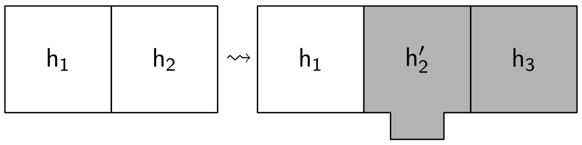


Figure 3.3: Execution of e in L4

have h'_2 referring to the modified h_2, h_3 contains new state and as such we require that it is disjoint from h_2 (it is disjoint from h_1 by construction), thus it cannot be 'hiding' any of the original heap. The reader should note that nothing prevents h'_2 containing new state in addition to the modified h_2 but h_3 contains *only* new state. We require that if h_3 (the new heap) has a sub-heap in common with h' (the required heap of the second execution) then h' must have a sub-heap in common with h'_2 (the modified portion). Thus, if we need some of the new heap, we also need some of the modified heap. In non-complete heaps, allocation could attach dangling pointers. If this were the case, then execution could access h_3 via e.g. h_1. The need for for **L4** will be further discussed in the discussion of Theorem 3.5.2 where the context provided by the proof should be helpful.

We require some additional properties of languages to apply our approach. We describe these properties as *locality*. We describe locality for predicates and for expressions. Locality relates to some properties of invariance or monotonicity with regard to the size of a heap. In a local language, adding more state cannot 'break' anything it can only improve our knowledge.

Definition 3.2.2 (LocalLangDef) *A language*

$$\mathsf{L} = (\Pi, \mathsf{E}, \mathsf{S}, \mathsf{H}, \mathsf{V}, \mathsf{P}, \Gamma, \rightsquigarrow, \vdash_{\mathsf{e}}, \vdash_{\mathsf{c}}, [\![]\!]_{\mathsf{p}})$$

is a local language *if the following axioms are obeyed*

LL1 $\mathsf{h} \sqsubseteq \mathsf{h}' \implies [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}} \preceq [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}'}$

LL2 $\mathsf{e}, \mathsf{s}, \mathsf{h} \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v}, \mathsf{h}''\ ,\ \mathsf{h}''\#\mathsf{h}''' \implies \mathsf{e}, \mathsf{s}, \mathsf{h} * \mathsf{h}''' \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v}, \mathsf{h}'' * \mathsf{h}'''$

LL1 states that predicate satisfaction in a larger heap is greater than or equal to ($\preceq$) that in the smaller state. If you like, adding more state can only improve our knowledge of satisfaction, it cannot reverse it.
LL2 states that if execution in a heap, h, succeeds resulting in a heap h'', it succeeds, with the same result in a larger heap. Additionally the final heap is h'' plus the additional heap. This is like stating that if an execution is successful it can read no more of the larger heap than that in which execution succeeded.

3.3 Non-interference

Now that we have defined programming languages we formalise the notions of non-interference we discussed in Chapter 2.

Definition 3.3.1 *For a language,* L *we define the relations:*

$$\_ \models \_\#\_ \subseteq \Gamma \times \mathsf{P} \times \mathsf{E} \qquad \_ \models \_\#\_ \subseteq \Gamma \times \mathsf{E} \times \mathsf{E}$$

as follows

$$\gamma \models \mathsf{p}\#\mathsf{e} \iff \pi \vdash \gamma\ ,\ \gamma \vdash \mathsf{s}, \mathsf{h}\ ,\ \mathsf{e}, \mathsf{s}, \mathsf{h} \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v}, \mathsf{h}'' \Rightarrow [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}} \preceq [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}''}$$

$$\begin{array}{lll} \gamma \models \mathsf{e}\#\mathsf{e}' \iff & \pi \vdash \gamma\ ,\ \gamma \vdash \mathsf{s}, \mathsf{h}\ , & \mathsf{e}, \mathsf{s}, \mathsf{h}_1 \overset{\mathsf{h}}{\rightsquigarrow}_\pi \mathsf{v}, \mathsf{h}_2\ ,\ \mathsf{e}', \mathsf{s}, \mathsf{h}_2 \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v}', \mathsf{h}_3 \\ \qquad\qquad \Rightarrow & & \mathsf{e}', \mathsf{s}, \mathsf{h}_1 \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v}', \mathsf{h}_4\ ,\ \mathsf{e}, \mathsf{s}, \mathsf{h}_4 \overset{\mathsf{h}}{\rightsquigarrow}_\pi \mathsf{v}, \mathsf{h}_3 \end{array}$$

In the the above we express that

- If $\gamma \models \mathsf{p}\#\mathsf{e}$ then evaluation of e does not change satisfaction of p other than to make it more precise.
- If $\gamma \models \mathsf{e}\#\mathsf{e}'$ then evaluation of e does not affect execution of e'. The order of execution can be changed but the final heap and the results of the executions are the same. Moreover, each execution has the same required heap, regardless of the order (the execution 'works in the same way').

3.4 Languages With Effects

We view an effects system as sitting atop a language, L, and producing a *language with effects*, L_E. Thus we allow multiple effects systems for a single language. Any such effects system will thus be expressing properties already in L rather than introducing new properties. It is feasible that multiple effects systems exist for a single language, either exploiting different properties of the language or giving differing strengths of effect based on the same principal.

Effects are syntactic entities describing regions of heaps. The meaning of an effect can be interpreted with respect to any heap and stack. In degenerate cases the effect may only describe the empty sub-heap, for example, if the effect refers to some feature that the particular heap does not have.

We view an effects system as giving effects for both predicates and expressions. These effects describe the heaps needed for satisfaction and execution respectively. Effects are, in general, conservative and thus will characterise larger heaps than are strictly necessary. Our model allows for effects of any precision from exact to uselessly imprecise.

Definition 3.4.1 *For a structure of separable heaps* $\mathsf{H} = (\mathsf{HC}, \epsilon, *)$ *and a language description* $\mathsf{L} = (\Pi, \mathsf{E}, \mathsf{S}, \mathsf{H}, \mathsf{V}, \mathsf{P}, \Gamma, \rightsquigarrow, \vdash_{\mathsf{e}}, \vdash_{\mathsf{c}}, [\![]\!]_{\mathsf{p}})$ *a language with effects,* $\mathsf{L_E}$*, is a tuple*

$$\mathsf{L_E} = (\mathsf{L}, \Phi, ||_{\mathsf{H}}, \#_\phi, :_{\mathsf{e}}, :_{\mathsf{p}})$$

which obeys the following axioms

LS1 $|\mathsf{h}|_{\mathsf{s},\phi} = \mathsf{h}' \implies \mathsf{h}' \sqsubseteq \mathsf{h}$

LS2 $\gamma \vdash \phi \# \phi'\ ,\ \gamma \vdash \mathsf{s}, \mathsf{h} \implies |\mathsf{h}|_{\mathsf{s},\phi} \# |\mathsf{h}|_{\mathsf{s},\phi'}$

LS3 $\gamma \vdash \mathsf{p} : \phi\ ,\ \gamma \vdash \mathsf{s}, \mathsf{h}\ ,\ |\mathsf{h}|_{\mathsf{s},\phi} = \mathsf{h}' \implies [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}} = [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}'}$

LS4 $\gamma \vdash \mathsf{e} : \phi, \phi'\ ,\ \pi \vdash \gamma\ ,\ \gamma \vdash \mathsf{s}, \mathsf{h}\ ,\ \mathsf{e}, \mathsf{s}, \mathsf{h} \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v}, \mathsf{h}''$

$\Rightarrow \exists \mathsf{h}_1,\ \mathsf{h}_2,\ \mathsf{h}_3,\ \mathsf{h}_4,\ \mathsf{h}_2{}'.$

$|\mathsf{h}|_{\mathsf{s},\phi} = \mathsf{h}_1 * \mathsf{h}_2\ ,\ |\mathsf{h}|_{\mathsf{s},\phi'} = \mathsf{h}_2\ ,\ \mathsf{h} = \mathsf{h}_1 * \mathsf{h}_2 * \mathsf{h}_3\ ,\ \mathsf{h}'' = \mathsf{h}_1 * \mathsf{h}_2{}' * \mathsf{h}_3 * \mathsf{h}_4$

$\mathsf{e}, \mathsf{s}, \mathsf{h}_1 * \mathsf{h}_2 \overset{\mathsf{h}'}{\rightsquigarrow}_\pi \mathsf{v}, \mathsf{h}_1 * \mathsf{h}_2{}' * \mathsf{h}_4\ ,\ \mathsf{h}_2 \# \mathsf{h}_4\ ,\ \mathsf{h}_1 * \mathsf{h}_2{}' \sqsubseteq |\mathsf{h}''|_{\mathsf{s},\phi}$

$\mathsf{h}_2{}' \sqsubseteq |\mathsf{h}''|_{\mathsf{s},\phi'}$

LS5 $\gamma \vdash \mathsf{e} : \phi, \phi' \Rightarrow \gamma \vdash \mathsf{e}$

where Φ *is the set of* effects *(with* $\phi, \phi', \ldots \in \Phi$*) and*

$$||_{\mathsf{H}} : \mathsf{H} \times \mathsf{S} \times \Phi \to \mathsf{H} \qquad \#_\phi \subseteq \Gamma \times \Phi \times \Phi$$
$$:_{\mathsf{e}} \subseteq \Gamma \times \mathsf{E} \times \Phi \times \Phi \qquad :_{\mathsf{p}} \subseteq \Gamma \times \mathsf{P} \times \Phi$$

and we use the following shorthands:

$|\mathsf{h}|_{\mathsf{s},\phi}$ *for* $(\mathsf{h}, \mathsf{s}, \phi) \in ||_{\mathsf{H}}$

$\gamma \vdash \phi \# \phi'$ *for* $(\gamma, \phi, \phi') \in \#_\phi$

$\gamma \vdash \mathsf{e} : \phi, \phi'$ *for* $(\gamma, \mathsf{e}, \phi, \phi') \in :_{\mathsf{e}}$

$\gamma \vdash \mathsf{p} : \phi$ *for* $(\gamma, \mathsf{p}, \phi) \in :_{\mathsf{p}}$

- $|\mathsf{h}|_{\mathsf{s},\phi}$ maps h to the heap described by ϕ with respect to s. The function is total. A stack is, in general, necessary as it is the bridge between the language syntax (i.e. variables) and elements of the heap. It provides our access points into the heap.

- $\gamma \vdash \phi \# \phi'$ states that the two effects are *disjoint*, with respect to the the environment γ.

- $\gamma \vdash \mathsf{e} : \phi, \phi'$ states that the effect of executing e, with respect to γ, is ϕ,ϕ' where ϕ is the *read effect* and ϕ' is the *write effect*.

- $\gamma \vdash \mathsf{p} : \phi$ states that the predicate p, with respect to γ, has effect ϕ. Predicates effects consists of only one effect (unlike expressions) since predicates will only inspect the heap and not modify it.

LS1 enforces that projection of an effect onto a heap must result in a sub-heap. We do not wish effects to 'invent' state!
LS2 gives semantics to disjoint effects. The projection of two disjoint effects are disjoint heaps.
LS3 describes soundness of predicate effects. If a predicate has effect ϕ, satisfaction must be the same in the projected heap as in the original. Thus the effect describes all the state which determines satisfaction of that predicate (otherwise counterexamples should exist).
LS4 describes soundness of effects for expressions. The read effect, ϕ, describes the sub-heap required for execution and includes the sub-heap described by ϕ'. ϕ' is the write effect and includes the parts of the heap that will be modified. Execution succeeds on the heap described by ϕ as it did on the full heap i.e. with the same result and the same required heap, resultant heap is the same as in the larger heap, less the unused part (h_3). The execution on the full heap is illustrated in Figure 3.4. As in **L4** the condition $\mathsf{h}'_2 \# \mathsf{h}_4$ ensures that h_4 contains only new state. Additionally, we require that the $\mathsf{h}_1 * \mathsf{h}'_2$ is a sub-heap of the projection of ϕ in the new heap, and the same for h'_2 and ϕ'. Thus, execution preserves effects, nothing can be removed from an effect it can only be enlarged. h_4, the new state, is disjoint from h_1 and h'_2. Parts of h_4 might be contained in the projection of the effects onto h'', but we do not make an assumption either way. This approach will allow the model to describe effects systems which record allocation of a new object in the write effect or which don't. This does not affect soundness but does make the model more general. **LS5** simply states that effects can only be found for well-formed expressions.

3.5 Effects and Non-Interference

We now present our central results which relate properties of effects to the properties of non-interference.

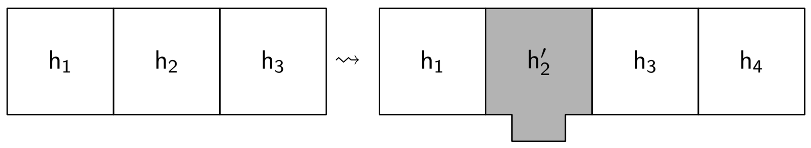


Figure 3.4: Execution on the initial heap in LS4

Theorem 3.5.1
For a local language with effects

$$\mathsf{L_E} = ((\Pi, \mathsf{E}, \mathsf{S}, (\mathsf{HC}, \epsilon, *), \mathsf{V}, \mathsf{P}, \Gamma, \rightsquigarrow, \vdash_{\mathsf{e}}, \vdash_{\mathsf{c}}, [\![]\!]_{\mathsf{p}}), \Phi, ||_{\mathsf{H}}, \#_{\phi}, :_{\mathsf{e}}, :_{\mathsf{p}})$$

If

1: $\gamma \vdash \mathsf{e} : \phi_1, \phi_2$
2: $\gamma \vdash \mathsf{p} : \phi_3$
3: $\gamma \vdash \phi_2 \# \phi_3$

then

4: $\gamma \models \mathsf{e} \# \mathsf{p}$

Proof: Suppose:

5: $\pi \vdash \gamma$
6: $\gamma \vdash \mathsf{s}, \mathsf{h}$
7: $\mathsf{e}, \mathsf{s}, \mathsf{h} \overset{\mathsf{h}'}{\rightsquigarrow}_{\pi} \mathsf{v}, \mathsf{h}''$

Then there exist h_1, h_2, h_3, $\mathsf{h}_2{}'$, h_4 and h_5 such that

8: $|\mathsf{h}|_{\mathsf{s},\phi_1} = \mathsf{h}_1 * \mathsf{h}_2$ (1,6,7,**LS4**)
9: $|\mathsf{h}|_{\mathsf{s},\phi_2} = \mathsf{h}_2$ (1,6,7,**LS4**)
10: $\mathsf{h} = \mathsf{h}_1 * \mathsf{h}_2 * \mathsf{h}_3$ (1,6,7,**LS4**)
11: $\mathsf{h}'' = \mathsf{h}_1 * \mathsf{h}_2{}' * \mathsf{h}_3 * \mathsf{h}_4$ (1,6,7,**LS4**)
12: $|\mathsf{h}|_{\mathsf{s},\phi_3} = \mathsf{h}_5$ ($||_{\mathsf{H}}$ is total)
13: $\mathsf{h}_5 \sqsubseteq \mathsf{h}$ (12, **LS1**)
14: $[\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}} = [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}_5}$ (2,6,12,**LS3**)
15: $\mathsf{h}_5 \# \mathsf{h}_2$ (2, 6, **LS2**, 9, 12)
16: $\mathsf{h}_5 \sqsubseteq \mathsf{h}_1 * \mathsf{h}_3$ (10, 13, 15, Lemma 3.1.1)
17: $\mathsf{h}_5 \sqsubseteq \mathsf{h}''$ (11, 16, transitivity of $\sqsubseteq$)
18: $[\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}_5} \preceq [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}''}$ (17, **LL1**)
19: $[\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}} \preceq [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}''}$ (14, 18)

□

In the above proof, by **LS3**, we know that satisfaction of p is the same in h_5 (the projection of the effect of p) as it is in h. The disjointness of h_2 (the projection of the write effect of e) and h_5 along with Lemma 3.1.1 yields that h_5 is a sub-heap of $\mathsf{h}_1 * \mathsf{h}_3$ which is unchanged by the execution (it is a sub-heap of h''). Thus h_5 is a sub-heap of h' and by **LL1** satisfaction of p in h'' is greater than or equal to satisfaction of p in h. By definition, p and e are non-interfering.

Theorem 3.5.2
For a local language with effects

$$\mathsf{L_E} = ((\Pi, \mathsf{E}, \mathsf{S}, (\mathsf{HC}, \epsilon, *), \mathsf{V}, \mathsf{P}, \Gamma, \rightsquigarrow, \vdash_{\mathsf{e}}, \vdash_{\mathsf{c}}, [\![]\!]_{\mathsf{p}}), \Phi, ||_{\mathsf{H}}, \#_{\phi}, :_{\mathsf{e}}, :_{\mathsf{p}})$$

If

20: $\gamma \vdash \mathsf{e} : \phi_1, \phi_2$
21: $\gamma \vdash \mathsf{e}' : \phi_3, \phi_4$
22: $\gamma \vdash \phi_1 \# \phi_4$
23: $\gamma \vdash \phi_3 \# \phi_2$

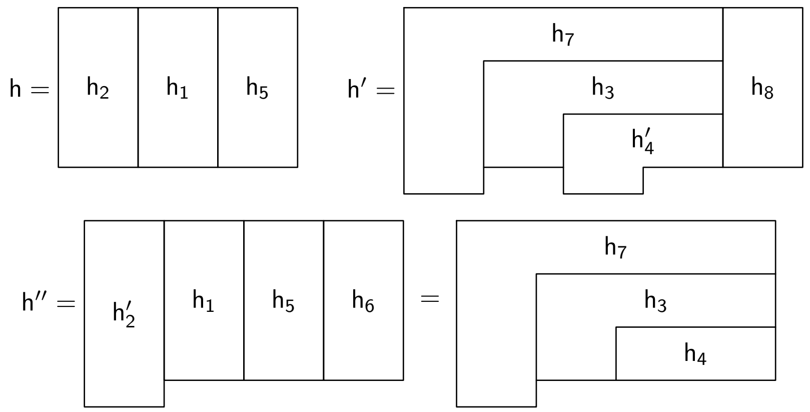
then

24: $\gamma \models \mathsf{e} \# \mathsf{e}'$

Much of the proof is concerned with the manipulation of heaps and checking of disjointness. We offer a graphical summary of the key points to aid reading of the full proof. The full proof follows this discussion.

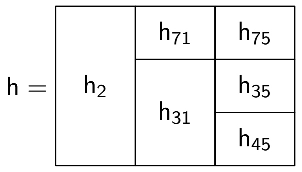
Let e take heap h to h'' and e' take h'' to h'. h_1 and h_2 are the projections of ϕ_1 and ϕ_2 in h. h_3 and h_4 are the projections of ϕ_3 and ϕ_4 in h''. We proceed by deriving a fine-grained splitting of each of the main heaps. We then show that the required heap of each expression is present, and unchanged, when we execute them in reverse order. Finally we show equality between the final heaps in each order of execution.

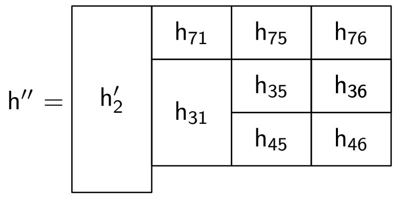
We know, from the effects of e and e' and **LS4** that we can break the main heaps h, h' and h'' as follows:

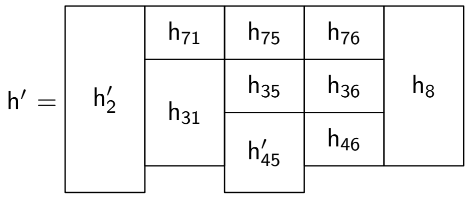


Because of Lemma 3.1.5 we can compare the two splittings of h'' and describe it in terms of the intersections of sub-heaps (h_{ab} refers to the intersection of heaps h_a and h_b). We also exploit the disjointness properties we have from the effects of e and e' to determine that the intersection of certain heaps is empty, h_1 and h_4, for example.

It takes a little extra work (and use of **L4**) to determine that h'_4 compromises h_{46} and h'_{45} i.e. there is no part of h'_4 in h_6. Note a slight abuse of our notation in that h'_{45} is not the intersection of h'_4 and h_5 but rather a modified h_{45}. So, we find that h, h'' and h' are as follows:





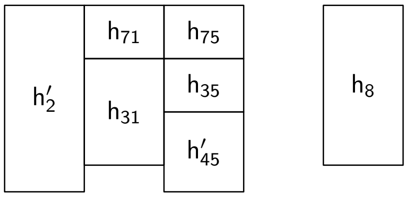


From **LS4** we also know that there is a successful execution of e using only $h_1 * h_2$ and resulting in $h_1 * h'_2 * h_6$. Similarly we know that there is an execution of e' using only $h_3 * h_4$ resulting in $h_3 * h'_4 * h_8$.

Our first aim is to show that there is an execution of e' using h. To do this we show that the required heap (of execution of e' in h''), $h^\ddagger$, is a sub-heap of h and then use **LL2** to add in the additional state. We know that $h^\ddagger$ must be a sub-heap of $h_3 * h_4$ from **L3**. However $h_3 * h_4$ has a sub-heap, $h_{36} * h_{46}$ which is not in h. Through **L4** we determine that h_6 must be disjoint from $h^\ddagger$, since h'_2 is (h'_2 is disjoint from $h_3 * h_4$ and so also from $h^\ddagger$). **L4** is essential here as otherwise we could not know that execution of e' does not require some of h_6 (which is allocated during execution of e and thus is unavailable when executing e' on h). The effects alone are not enough to determine this, as the read effects need not be disjoint. **L4** requires that h_6 can only be accessed via h'_2, which is disjoint from $h^\ddagger$ (by disjointness of effects).

Application of Lemma 3.1.1 gives that $h^\ddagger$ is a sub-heap of $h_{31} * h_{35} * h_{45}$. Thus, by **LL2** we know there is an execution of e' using $h_{31} * h_{35} * h_{45}$ and resulting in $h_{31} * h_{35} * h'_{45} * h_8$ (since there is an execution using only $h^\ddagger$).

$h_{31} * h_{35} * h_{45}$ is a sub-heap of h and, by **LL2**, again, we find that there is an execution of e' taking h to h''' where

$\mathsf{h}''' =$ 

We see that $\mathsf{h}_1 * \mathsf{h}_2$ is a sub-heap of h''' and, by another application of **LL2** that we there is an execution of e resulting in h'.

Proof: First, suppose we can execute e followed by e' on a heap, h (as in Definition 3.3.1) and that the program and heap/stack are well-formed:

25: $\pi \vdash \gamma$
26: $\gamma \vdash \mathsf{s}, \mathsf{h}$
27: $\mathsf{e}, \mathsf{s}, \mathsf{h} \overset{\mathsf{h}^{\dagger}}{\rightsquigarrow}_{\pi} \mathsf{v}'', \mathsf{h}''$
28: $\mathsf{e}', \mathsf{s}, \mathsf{h}'' \overset{\mathsf{h}^{\ddagger}}{\rightsquigarrow}_{\pi} \mathsf{v}', \mathsf{h}'$

We next apply LS4 to the executions of e and e' and find that there exist h_1, h_2, h'_2, h_3, h_4, h_5, h_6, $\mathsf{h}_4{}'$ such that:

29:	$\mathsf{s} \models \mathsf{h}$	(26, **L5**)
30:	$\mathsf{h} = \mathsf{h}_1 * \mathsf{h}_2 * \mathsf{h}_5$	(25, 26, 20, 27, **LS4**)
31:	$\lvert\mathsf{h}\rvert_{\mathsf{s},\phi_1} = \mathsf{h}_1 * \mathsf{h}_2$	(25, 26, 20, 27, **LS4**)
32:	$\lvert\mathsf{h}\rvert_{\mathsf{s},\phi_2} = \mathsf{h}_2$	(25, 26, 20, 27, **LS4**)
33:	$\mathsf{h}'' = \mathsf{h}_1 * \mathsf{h}_2{}' * \mathsf{h}_5 * \mathsf{h}_6$	(25, 26, 20, 27, **LS4**)
34:	$\mathsf{h}_2 \# \mathsf{h}_6$	(25, 26, 20, 27, **LS4**)
35:	$\mathsf{e}, \mathsf{s}, \mathsf{h}_1 * \mathsf{h}_2 \overset{\mathsf{h}^{\dagger}}{\rightsquigarrow}_{\pi} \mathsf{v}'', \mathsf{h}_1 * \mathsf{h}_2{}' * \mathsf{h}_6$	(25, 26, 20, 27, **LS4**)
36:	$\mathsf{h}_1 * \mathsf{h}_2{}' \sqsubseteq \lvert\mathsf{h}''\rvert_{\mathsf{s},\phi_1}$	(25, 26, 20, 27, **LS4**)
37:	$\mathsf{h}_2{}' \sqsubseteq \lvert\mathsf{h}''\rvert_{\mathsf{s},\phi_2}$	(25, 26, 20, 27, **LS4**)
38:	$\gamma \vdash \mathsf{e}$	(20,**LS5**)
39:	$\gamma \vdash \mathsf{s}, \mathsf{h}''$	(38, 25, 26, 27, **L1**)
40:	$\mathsf{h}'' = \mathsf{h}_3 * \mathsf{h}_4 * \mathsf{h}_7$	(25, 37, 21, 28, **LS4**)
41:	$\lvert\mathsf{h}''\rvert_{\mathsf{s},\phi_3} = \mathsf{h}_3 * \mathsf{h}_4$	(25, 37, 21, 28, **LS4**)
42:	$\lvert\mathsf{h}''\rvert_{\mathsf{s},\phi_4} = \mathsf{h}_4$	(25, 37, 21, 28, **LS4**)
43:	$\mathsf{h}' = \mathsf{h}_3 * \mathsf{h}'_4 * \mathsf{h}_7 * \mathsf{h}_8$	(25, 37, 21, 28, **LS4**)
44:	$\mathsf{h}_4 \# \mathsf{h}_8$	(25, 37, 21, 28, **LS4**)
45:	$\mathsf{e}', \mathsf{s}, \mathsf{h}_3 * \mathsf{h}_4 \overset{\mathsf{h}^{\ddagger}}{\rightsquigarrow}_{\pi} \mathsf{v}', \mathsf{h}_3 * \mathsf{h}_4{}' * \mathsf{h}_8$	(25, 37, 21, 28, **LS4**)
46:	$\mathsf{h}_3 * \mathsf{h}_4{}' \sqsubseteq \lvert\mathsf{h}'\rvert_{\mathsf{s},\phi_3}$	(25, 37, 21, 28, **LS4**)
47:	$\mathsf{h}_4{}' \sqsubseteq \lvert\mathsf{h}'\rvert_{\mathsf{s},\phi_4}$	(25, 37, 21, 28, **LS4**)

In order to proceed we must ascertan which heaps are disjoint from, or contained within, each other. We will make use of the properties of disjoint effects and also the properties of heaps. **L4** is also used to show that $\mathsf{h}_{\ddagger}$ is disjoint from h_6.

48:	$\lvert\mathsf{h}''\rvert_{\mathsf{s},\phi_2} \# \mathsf{h}_3 * \mathsf{h}_4$	(23, 37, 39, **LS2**)

49:	$\mathsf{h_2}'\#\mathsf{h_3} * \mathsf{h_4}$	(23, 45, 36, def. of $\sqsubseteq$, Lemma 3.1.2)
50:	$\lvert\mathsf{h}''\rvert_{\mathsf{s},\phi_1}\#\mathsf{h_4}$	(39, 22, 42, **LS2**)
51:	$\mathsf{h_1} * \mathsf{h}_2'\#\mathsf{h_4}$	(50, 36, Lemma 3.1.2)
52:	$\mathsf{h}^\ddagger \sqsubseteq \mathsf{h_3} * \mathsf{h_4}$	(45, **L3**)
53:	$\mathsf{h}^\ddagger\#\mathsf{h}_2'$	(52, 49, Lemma 3.1.2)
54:	$\mathsf{h_7}\#\mathsf{h_3} * \mathsf{h_4}$	(40, def $\sqsubseteq$)
55:	$\mathsf{h_7}\#\mathsf{h}^\ddagger$	(52, 54, def $\sqsubseteq$, Lemma 3.1.2)
56:	$\mathsf{h}^\ddagger\#\mathsf{h_6}$	(29, 27, 30, 33, 28, 34, **L4**, 53)
57:	$\mathsf{h_3} * \mathsf{h_4} * \mathsf{h_7} = \mathsf{h_1} * \mathsf{h}_2' * \mathsf{h_5} * \mathsf{h_6}$	(33, 40)
58:	$\mathsf{h_3} \sqsubseteq \mathsf{h_1} * \mathsf{h_5} * \mathsf{h_6}$	(57, def. $\sqsubseteq$, 49, Lemma 3.1.2,)
59:	$\mathsf{h_4} \sqsubseteq \mathsf{h_5} * \mathsf{h_6}$	(57, def. $\sqsubseteq$, 51, Lemma 3.1.1)
60:	$\mathsf{h_7} \sqsubseteq \mathsf{h_1} * \mathsf{h}_2' * \mathsf{h_5} * \mathsf{h_6}$	(57, def. $\sqsubseteq$)
61:	$\mathsf{h_1} \sqsubseteq \mathsf{h_3} * \mathsf{h_7}$	(57, def. $\sqsubseteq$, 51, Lemma 3.1.2, Lemma 3.1.1)
62:	$\mathsf{h}_2' \sqsubseteq \mathsf{h_7}$	(57, def. $\sqsubseteq$, 49, Lemma 3.1.1)
63:	$\mathsf{h_5} \sqsubseteq \mathsf{h_3} * \mathsf{h_4} * \mathsf{h_7}$	(57, def. $\sqsubseteq$)
64:	$\mathsf{h_6} \sqsubseteq \mathsf{h_3} * \mathsf{h_4} * \mathsf{h_7}$	(57, def. $\sqsubseteq$)

By repeated application of Lemma 3.1.4 and the containment relations inferred above, we define a number of additional heaps. These heaps represent the intersection of the larger heaps above.

These smaller heaps will be necessary later to isolate heaps containing the required heap for execution of each expression. By combining them, we will be able to identify e.g. a sub-heap of h that contains $\mathsf{h}^\ddagger$.

65:	$\mathsf{h_{31}} = \mathsf{h_3} \sqcap \mathsf{h_1} = \mathsf{h_1} \sqcap \mathsf{h_3}$	(57, def $\sqsubseteq$, Lemma 3.1.4)
66:	$\mathsf{h_{35}} = \mathsf{h_3} \sqcap \mathsf{h_5} = \mathsf{h_5} \sqcap \mathsf{h_3}$	(57, def $\sqsubseteq$, Lemma 3.1.4)
67:	$\mathsf{h_{36}} = \mathsf{h_3} \sqcap \mathsf{h_6} = \mathsf{h_6} \sqcap \mathsf{h_3}$	(57, def $\sqsubseteq$, Lemma 3.1.4)
68:	$\mathsf{h_{45}} = \mathsf{h_4} \sqcap \mathsf{h_5} = \mathsf{h_5} \sqcap \mathsf{h_4}$	(57, def $\sqsubseteq$, Lemma 3.1.4)
69:	$\mathsf{h_{46}} = \mathsf{h_4} \sqcap \mathsf{h_6} = \mathsf{h_6} \sqcap \mathsf{h_4}$	(57, def $\sqsubseteq$, Lemma 3.1.4)
70:	$\mathsf{h_{71}} = \mathsf{h_7} \sqcap \mathsf{h_1} = \mathsf{h_1} \sqcap \mathsf{h_7}$	(57, def $\sqsubseteq$, Lemma 3.1.4)
71:	$\mathsf{h}_2' = \mathsf{h_7} \sqcap \mathsf{h}_2' = \mathsf{h}_2' \sqcap \mathsf{h_7}$	(57, def $\sqsubseteq$, Lemma 3.1.4)
72:	$\mathsf{h_{75}} = \mathsf{h_7} \sqcap \mathsf{h_5} = \mathsf{h_5} \sqcap \mathsf{h_7}$	(57, def $\sqsubseteq$, Lemma 3.1.4)
73:	$\mathsf{h_{76}} = \mathsf{h_7} \sqcap \mathsf{h_6} = \mathsf{h_6} \sqcap \mathsf{h_7}$	(57, def $\sqsubseteq$, Lemma 3.1.4)

The intersecting heaps defined above can be recombined, using Lemma 3.1.5, into the larger heaps defined in 30 - 47.

74:	$\mathsf{h_3} = \mathsf{h_{31}} * \mathsf{h_{35}} * \mathsf{h_{56}}$	(58, def $\sqsubseteq$, Lemma 3.1.5, 65, 66, 67)
75:	$\mathsf{h_4} = \mathsf{h_{45}} * \mathsf{h_{46}}$	(59, def $\sqsubseteq$, Lemma 3.1.5, 68, 69)
76:	$\mathsf{h_7} = \mathsf{h_{71}} * \mathsf{h}_2' * \mathsf{h_{75}} * \mathsf{h_{76}}$	(60, def $\sqsubseteq$, Lemma 3.1.5, 70, 71, 72, 73)
77:	$\mathsf{h_1} = \mathsf{h_{31}} * \mathsf{h_{71}}$	(61, def $\sqsubseteq$, Lemma 3.1.5, 65, 70)
78:	$\mathsf{h_5} = \mathsf{h_{35}} * \mathsf{h_{45}} * \mathsf{h_{75}}$	(63, def $\sqsubseteq$, Lemma 3.1.5, 66, 68, 72)
79:	$\mathsf{h_6} = \mathsf{h_{36}} * \mathsf{h_{46}} * \mathsf{h_{76}}$	(64, def $\sqsubseteq$, Lemma 3.1.5, 67, 69, 73)

We now investigate the required heap of the execution of e'. Our aim is to show the required heap is contained in h and to describe the result of executing in h. To simplify matters we consider execution only in $\mathsf{h}_3 * \mathsf{h}_4$ as in 45. Through **L3** and 45 we introduce $\mathsf{h}^{\ddagger\prime}$, to represent the result of execution using only the required heap, and $\mathsf{h}^{\ddagger}_{\mathsf{r}}$ which is the remainder heap when e' is executed using $\mathsf{h}_3 * \mathsf{h}_4$.

80: $\mathsf{e}', \mathsf{s}, \mathsf{h}^{\ddagger} \overset{\mathsf{h}^{\ddagger}}{\rightsquigarrow}_{\pi} \mathsf{v}', \mathsf{h}^{\ddagger\prime}$ (45, **L3**)
81: $\mathsf{h}_3 * \mathsf{h}_4 = \mathsf{h}^{\ddagger} * \mathsf{h}^{\ddagger}_{\mathsf{r}}$ (45, **L3**)
82: $\mathsf{h}_3 * \mathsf{h}'_4 * \mathsf{h}_8 = \mathsf{h}^{\ddagger\prime} * \mathsf{h}^{\ddagger}_{\mathsf{r}}$ (45, **L3**)

We now move towards a precise characterisation of $\mathsf{h}^{\ddagger}_{\mathsf{r}}$, which in turn will increase our knowledge of where $\mathsf{h}^{\ddagger}$ lies. We already know, by 56, that $\mathsf{h}^{\ddagger}$ is disjoint from h_6. h_3 and h_4 have parts in h_6 but these cannot be included in $\mathsf{h}^{\ddagger}$.

83: $\mathsf{h}_{36} * \mathsf{h}_{46} \# \mathsf{h}^{\ddagger}$ (56, 79, Lemma 3.1.2)
84: $\mathsf{h}_{36} * \mathsf{h}_{46} \sqsubseteq \mathsf{h}_3 * \mathsf{h}_4$ (74, 75, def $\sqsubseteq$)
85: $\mathsf{h}_{36} * \mathsf{h}_{46} \sqsubseteq \mathsf{h}^{\ddagger}_{\mathsf{r}}$ (81, 84, 83, Lemma 3.1.1)

Since 85, there must be some remainder in $\mathsf{h}^{\ddagger}_{\mathsf{r}}$ which we will call $\mathsf{h}^{\ddagger}_{\mathsf{rr}}$.

86: $\mathsf{h}^{\ddagger}_{\mathsf{r}} = \mathsf{h}_{36} * \mathsf{h}_{46} * \mathsf{h}^{\ddagger}_{\mathsf{rr}}$ (85, def $\sqsubseteq$)

We substitute for $\mathsf{h}^{\ddagger}_{\mathsf{r}}$ to get the following equalities:

87: $\mathsf{h}_3 * \mathsf{h}_4 = \mathsf{h}^{\ddagger} * \mathsf{h}^{\ddagger}_{\mathsf{rr}} * \mathsf{h}_{36} * \mathsf{h}_{46}$ (81, 86)
88: $\mathsf{h}_3 * \mathsf{h}'_4 * \mathsf{h}_8 = \mathsf{h}^{\ddagger\prime} * \mathsf{h}^{\ddagger}_{\mathsf{rr}} * \mathsf{h}_{36} * \mathsf{h}_{46}$ (82, 86)

We are now in a position to find the constituents of h'_4 as we did in 74 - 79. The equality above shows h_{46} is in $\mathsf{h}_3 * \mathsf{h}'_4 * \mathsf{h}_8$. However, as it is disjoint from $\mathsf{h}_3 * \mathsf{h}_8$ it must be part of h'_4. We introduce h'_{45} to be the remaining part of h'_4. We pick this name for illustrative purposes (we won't need to show that h'_{45} is the modified h_{45}, though it is!).

89: $\mathsf{h}_3 \# \mathsf{h}_4$ (40, def #)
90: $\mathsf{h}_4 \# \mathsf{h}_3 * \mathsf{h}_8$ (89, 44, 43, def #, **SH6**)
91: $\mathsf{h}_{46} \# \mathsf{h}_3 * \mathsf{h}_8$ (75, 90, Lemma 3.1.2)
92: $\mathsf{h}_{46} \sqsubseteq \mathsf{h}'_4$ (91, 88, Lemma 3.1.2)
93: $\mathsf{h}'_4 = \mathsf{h}_{46} * \mathsf{h}'_{45}$ (92, def $\sqsubseteq$)

By substituting for h'_4, among others, we are able to determine the components of $\mathsf{h}^{\ddagger}$, $\mathsf{h}^{\ddagger}_{\mathsf{rr}}$ and $\mathsf{h}^{\ddagger\prime}$.

94: $\mathsf{h}_{31} * \mathsf{h}_{35} * \mathsf{h}_{36} * \mathsf{h}_{45} * \mathsf{h}_{46} = \mathsf{h}^{\ddagger} * \mathsf{h}^{\ddagger}_{\mathsf{rr}} * \mathsf{h}_{36} * \mathsf{h}_{46}$ (81, 74, 75, 86)
95: $\mathsf{h}_{31} * \mathsf{h}_{35} * \mathsf{h}_{36} * \mathsf{h}'_{45} * \mathsf{h}_{46} * \mathsf{h}_8 = \mathsf{h}^{\ddagger\prime} * \mathsf{h}^{\ddagger}_{\mathsf{rr}} * \mathsf{h}_{36} * \mathsf{h}_{46}$ (82, 74, 93, 86)
96: $\mathsf{h}_{31} * \mathsf{h}_{35} * \mathsf{h}_{45} = \mathsf{h}^{\ddagger} * \mathsf{h}^{\ddagger}_{\mathsf{rr}}$ (94, Lemma 3.1.3)
97: $\mathsf{h}_{31} * \mathsf{h}_{35} * \mathsf{h}'_{45} * \mathsf{h}_8 = \mathsf{h}^{\ddagger\prime} * \mathsf{h}^{\ddagger}_{\mathsf{rr}}$ (95, Lemma 3.1.3)

96 shows that $\mathsf{h}^{\ddagger} * \mathsf{h}^{\ddagger}_{\mathsf{rr}}$ is contained in h, since h_{31}, h_{35} and h_{45} are. Using **LL2** we shall show that there exists an execution of e' using h and will be able to characterise the resulting heap. First notice that, by **LL2**, there exists an execution of e' using $\mathsf{h}^{\ddagger} * \mathsf{h}^{\ddagger}_{\mathsf{rr}}$.

98: $\mathsf{h}^{\ddagger}_{\mathsf{rr}} \# \mathsf{h}^{\ddagger\prime}$ (82, 86, def #)
99: $\mathsf{e}', \mathsf{s}, \mathsf{h}^{\ddagger} * \mathsf{h}^{\ddagger}_{\mathsf{rr}} \overset{\mathsf{h}^{\ddagger}}{\rightsquigarrow}_{\pi} \mathsf{v}', \mathsf{h}^{\ddagger\prime} * \mathsf{h}^{\ddagger}_{\mathsf{rr}}$ (80, 98, **LL2**)
100: $\mathsf{e}', \mathsf{s}, \mathsf{h}_{31} * \mathsf{h}_{35} * \mathsf{h}_{45} \overset{\mathsf{h}^{\ddagger}}{\rightsquigarrow}_{\pi} \mathsf{v}', \mathsf{h}_{31} * \mathsf{h}_{35} * \mathsf{h}'_{45} * \mathsf{h}_{8}$ (99, 96, 97)

Now notice, from our previous deconstructions of heaps, that:

101: $\mathsf{h} = \mathsf{h}_{31} * \mathsf{h}_{71} * \mathsf{h}_{2} * \mathsf{h}_{35} * \mathsf{h}_{45} * \mathsf{h}_{75}$ (30, 77, 78)
102: $\mathsf{h}_{71} * \mathsf{h}_{75} * \mathsf{h}_{2}$ is defined (101, **SH2**, **SH1**)
103: $\mathsf{h}_{71} * \mathsf{h}_{75} * \mathsf{h}_{2} \# \mathsf{h}_{31} * \mathsf{h}_{35}$ (101, 102, **SH2**, **SH1**, def #)

We wish to apply **LL2** to 100 using $\mathsf{h}_{71} * \mathsf{h}_{75} * \mathsf{h}_{2}$. In order to do so we must first show that it is disjoint from $\mathsf{h}_{31} * \mathsf{h}_{35} * \mathsf{h}'_{45} * \mathsf{h}_{8}$. We already have 103 so we will show disjointness with h'_{45} and h_{8} separately then combine using Lemma 3.1.2. First we consider h'_{45}:

104: $\mathsf{h}_{71} * \mathsf{h}_{75} \# \mathsf{h}'_{4}$ (76, 43, def #, Lemma 3.1.2)
105: $\mathsf{h}_{71} * \mathsf{h}_{75} \# \mathsf{h}'_{45}$ (104, 93, Lemma 3.1.2)
106: $\mathsf{h}' = \mathsf{h}_{31} * \mathsf{h}_{35} * \mathsf{h}_{36} * \mathsf{h}_{46} * \mathsf{h}'_{45} * \mathsf{h}_{71} * \mathsf{h}'_{2} * \mathsf{h}_{75} * \mathsf{h}_{76} * \mathsf{h}_{8}$ (43, 74, 93, 76)
107: $\mathsf{h}_{36} * \mathsf{h}_{46} * \mathsf{h}_{76} \# \mathsf{h}'_{45}$ (106, **SH2**, **SH1**, def #)
108: $\mathsf{h}_{6} \# \mathsf{h}'_{45}$ (107, 79)
109: $\mathsf{h}_{1} * \mathsf{h}'_{2} * \mathsf{h}_{6} \# \mathsf{h}'_{45}$ (108, 51, 35, def #, **SH6**)
110: $\mathsf{h}_{1} * \mathsf{h}_{2} \# \mathsf{h}'_{45}$ (109, 35, **L2**)
111: $\mathsf{h}_{2} \# \mathsf{h}'_{45}$ (110, Lemma 3.1.2)
112: $\mathsf{h}_{71} * \mathsf{h}_{75} * \mathsf{h}_{2} \# \mathsf{h}'_{45}$ (105, 111, Lemma 3.1.2)

Now we treat h_{8}:

113: $\mathsf{h}_{8} \# \mathsf{h}_{7}$ (43, def #)
114: $\mathsf{h}_{8} \# \mathsf{h}_{71} * \mathsf{h}_{75}$ (113, 76, Lemma 3.1.2)
115: $\mathsf{h}_{8} \# \mathsf{h}_{3} * \mathsf{h}_{7}$ (43, def. #)
116: $\mathsf{h}_{8} \# \mathsf{h}_{3} * \mathsf{h}_{4} * \mathsf{h}_{7}$ (115, 44, **SH6**)
117: $\mathsf{h}_{8} \# \mathsf{h}_{1} * \mathsf{h}_{2} * \mathsf{h}_{5}$ (116, 40, 27, 30, **L2**)
118: $\mathsf{h}_{8} \# \mathsf{h}_{2}$ (117, Lemma 3.1.2)
119: $\mathsf{h}_{8} \# \mathsf{h}_{71} * \mathsf{h}_{75} * \mathsf{h}_{2}$ (118, 114, 102, Lemma 3.1.2)

Using **SH6** we combine our two disjointness results and define h'''.

120: $\mathsf{h}_{31} * \mathsf{h}_{35} * \mathsf{h}'_{45} * \mathsf{h}_{8} \# \mathsf{h}_{71} * \mathsf{h}_{75} * \mathsf{h}_{2}$ (119, 103, 112, **SH6**)
121: $\mathsf{h}''' = \mathsf{h}_{31} * \mathsf{h}_{35} * \mathsf{h}'_{45} * \mathsf{h}_{8} * \mathsf{h}_{71} * \mathsf{h}_{75} * \mathsf{h}_{2}$ (119, **SH2**, **SH1**, 77)

Notice also that:

122: $\mathsf{h}''' = \mathsf{h}_{1} * \mathsf{h}_{2} * \mathsf{h}_{35} * \mathsf{h}'_{45} * \mathsf{h}_{75} * \mathsf{h}_{8}$ (121, 77)

We can now show, using **LL2** that e' can be executed using h.

123: $\mathsf{e}', \mathsf{s}, \mathsf{h} \overset{\mathsf{h}^\ddagger}{\rightsquigarrow}_\pi \mathsf{v}', \mathsf{h}'''$ (100, 120, **LL2**, 121, 106)

Now we need to repeat our previous process and show that we can execute e using h'''. We will use **LL2** again. Since $\mathsf{h}_1 * \mathsf{h}_2$ is in h''' we must only characterise the resulting heap and check disjointness. Notice first that:

124: $\mathsf{h}_1 * \mathsf{h}_2' * \mathsf{h}_6 \# \mathsf{h}_8$ (116, 40, 33, Lemma 3.1.2)
125: $\mathsf{h}_1 * \mathsf{h}_2' * \mathsf{h}_6 \# \mathsf{h}_5$ (33, **SH2**, **SH1**, def #)
126: $\mathsf{h}_1 * \mathsf{h}_2' * \mathsf{h}_6 \# \mathsf{h}_{35} * \mathsf{h}_{75}$ (125, 78, Lemma 3.1.2)
127: $\mathsf{h}_1 * \mathsf{h}_2' * \mathsf{h}_6 \# \mathsf{h}_{45}'$ (106, 77, 79, **SH2**, **SH1**, def #)
128: $\mathsf{h}_1 * \mathsf{h}_2' * \mathsf{h}_6 \# \mathsf{h}_{35} * \mathsf{h}_{45}' * \mathsf{h}_{75} * \mathsf{h}_8$ (124, 126, 127)

and say

129: $\mathsf{h}'''' = \mathsf{h}_1 * \mathsf{h}_2' * \mathsf{h}_6 * \mathsf{h}_{35} * \mathsf{h}_{45}' * \mathsf{h}_{75} * \mathsf{h}_8$ (128, def #)

So, by **LL2**:

130: $\mathsf{e}, \mathsf{s}, \mathsf{h}''' \overset{\mathsf{h}^\dagger}{\rightsquigarrow}_\pi \mathsf{v}'', \mathsf{h}''''$ (35, 128, **LL2**, 129, 122)

It is easily checked that h'''' is the same as h' and so:

131: $\mathsf{h}'''' = \mathsf{h}_{31} * \mathsf{h}_{71} * \mathsf{h}_2' * \mathsf{h}_{36} * \mathsf{h}_{46} * \mathsf{h}_{76} * \mathsf{h}_{35} * \mathsf{h}_{45}' * \mathsf{h}_{75} * \mathsf{h}_8$ (129, 77, 79)
132: $\mathsf{h}'''' = \mathsf{h}'$ (131, 106)
133: $\mathsf{e}, \mathsf{s}, \mathsf{h}''' \overset{\mathsf{h}^\dagger}{\rightsquigarrow}_\pi \mathsf{v}', \mathsf{h}'$ (130, 132)

We have now shown that e' can be executed using h and that e can be executed subsequently with the same results and the same final heap as in the original executions of e and e'. Thus, by Definition 3.3.1, the expressions are non-interfering.

134: $\gamma \models \mathsf{e} \# \mathsf{e}'$ (27, 28, 123, 133, Def. 3.3.1)

□

3.6 Summary and Discussion

The model we have presented suggests sufficient conditions for languages and effects systems which can prove properties of non-interference. Our model is limited to sequential execution and assumes large step semantics but can describe languages in different paradigms. In the model we capture complicated language properties by description of heap structure and the changes made by execution.

We must, however, ask whether our model is sufficient to describe real-world program-

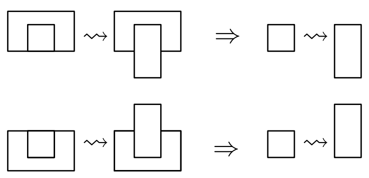


Figure 3.5: Non-determinism of allocation

ming languages. We now discuss a number of apparent limitations of our model but also argue that our results are preserved in the presence of apparently incompatible language features. We will assume that disjointness of heaps is defined by disjointness of the domains of the heaps.

3.6.1 Allocation Semantics

A restriction of our model is that, for a language to satisfy the axioms it must be able to allocate *any* memory not currently allocated i.e. allocation is non-deterministic. This is evidenced in **L3**, where we assume that an execution exists using only the required heap described by an execution in a larger heap. Further, we require that the non-required part of the heap is disjoint from the result of the required-heap execution, and that when put together they form the resultant heap from the original execution. A consequence of this requirement is that, should the expression allocate a new sub-heap, it must be allocated in the same place when execution is carried out with only the required heap.

Consider Figure 3.5. We show execution of the same expression in two different heaps (which are accessed by the same stack), however the required heap is in exactly the same place. Notice that each execution allocates new state in a different location. This must happen as the area allocated in one case is already occupied in the other. **L3** expects that we must, in each case, be able to execute with only the required heap, but get resulting heaps that are subheaps of the original resulting heap. Thus, there must be two executions of the expression on the same required heap, each giving a different result (in terms of the new state allocated). Thus the language must be non-deterministic when allocating. Theorem 3.5.2 takes advantage of the non-determinism of allocation. When e' is executed in h, it allocates the same new heap as it did when executed in h''. Most practical language implementations do not have this property as they allocate any available memory (which in the above case includes the remainder heap) according to their particular policy. It would thus appear that, in the real world, our results cannot hold. However, if we considered heaps not up to equality but up to equivalence we could accommodate deterministic allocation. The address of an object or cell has no inherent meaning, it is the pointer structure and the basic values (ints, floats, bools etc.) which are the defining features. Two heaps can be considered equivalent if a bijection exists between the their address spaces. Then, assuming appropriate stacks, execution of any expression using each heap produces equivalent values and heaps. When new memory is allocated, *where* it is allocated is not meaningful; its meaning is derived from the stack and those other parts of the heap that point to it and that it points to. This relationship will be the same in each of the equivalent heaps. It is easy to show an equivalence result between an operational semantics using deterministic and non-deterministic allocation semantics.

Returning to Theorem 3.5.2, even if two executions of the same expression (starting with the same stack and heap) allocate differently, the structure created by subsequent assignments will be identical. Expressions which are non-interfering (according to Theorem 3.5.2) will still be non-interfering but the resultant heaps and value will be equivalent rather than identical. This can be shown using the afore-mentioned equiavlence result and applying it to the executions in Theorem 3.5.2.

This problem cannot be solved with a clever deterministic allocation strategy since the executions in the required heap have no knowledge of the remainder heap we wish to join post-hoc - the strateggy could never know what locations were ‘safe’ to allocate. Our system relies on the *existence* of an execution that allocates the heap we want, and we choose to use that execution in preference to others. We wish to note that the use of non-deterministic allocation (typically allowing allocation of any ‘free’ address) is common in formal systems although not always exploited as in e.g. [BPP03, DE97], to name but a couple.

3.6.2 Deallocation and Garbage Collection

L2 requires that execution can only increase the size of the heap. Languages with explicit or automatic deallocation of memory cannot satisfy **L2** as any memory deallocated during an execution would be disjoint with the resulting heap but not with the initial. This appears to exclude the majority of real-world programming language implementations from our model. In fact, for ‘well-behaved’ languages e.g. Java, our results on non-interference still hold, in a modified form. As above, the key is to switch consideration to equivalent rather than equal heaps.

3.6.2.1 Explicit and Managed Deallocation

Languages like Java [GJSB05] have no mechanism whereby a programmer can explicitly choose to deallocate heap cells. Instead, a garbage collector [McC60] (running in a background thread) deallocates parts of the heap that are no longer usable i.e. there are no reachable pointers to it. All such deallocations are safe as no subsequent execution can access the garbage collected memory.

This contrasts with languages such as C++ [Str00] where, although garbage collectors exist, the programmer can choose to deallocate memory as they see fit. Undisciplined programming can result in dangling pointers which are unsafe to dereference. Worse still, these pointers can be unexpectedly connected by subsequent allocations.

3.6.2.2 Garbage Collection

Our problems with garbage collection are two-fold. First, the heap may be smaller after garbage collection and thus **L2** would not hold. Second, re-ordered execution of non-interfering expressions might not result in identical heaps. The problem is the same as with deterministic allocation - once memory is deallocated by the garbage collector it can be subsequently reallocated.

Consider expressions $\mathtt{e}$ and $\mathtt{e}'$ with disjoint effects as in Theorem 3.5.2. Consider execution of these expressions in a heap where nothing is eligible for garbage collection and where execution of $\mathtt{e}$ makes some allocation, but does not make any assignments and therefore does not make anything eligible for garbage collection. Suppose also that execution of $\mathtt{e}'$ does make some sub-heap eligible for garbage collection. If $\mathtt{e}'$ is executed first then garbage may be collected before $\mathtt{e}$ is executed. Next $\mathtt{e}$ is executed in the garbage collected heap and different memory is available to be allocated than before. If this occurs then we may end up with non-equal heaps after execution of $\mathtt{e}'$.

Despite all this we still expect our results on non-interference to hold in languages with garbage collection. The solution as in Section 3.6.1 is to consider heaps up to equivalence. Soundness of a garbage collector, as formalised in [HK03], requires that an execution where garbage is collected must give equivalent results to an execution without garbage collection. Applying this to the executions in Theorem 3.5.2 we note that if the executions exist then they still exist in the presence of garbage collection and give equivalent, but not necessarily equal, results.

3.6.3 Advanced Language Features

In many formal models of languages useful features are omitted. In many cases these features are not significant with respect to the other features (e.g. type system) being considered. In other cases such features may be incompatible with aspects of the formal language. We offer a discussion of common features often omitted from formal models and how they relate to effects and our model.

3.6.3.1 Exceptions

Exceptions in the style of Java complicate the issue of static analysis as it becomes harder to track what code will be executed. An effects system for a language with exceptions would have to add to the effect of an expression the effects of all exception handling code that may be run.

Since our model abstracts beyond details of how effects are calculated (and the information contained in the environment γ) our model should be able to accommodate such judgements.

3.6.3.2 Concurrency

Many language formalisms do not model concurrent features for example when modeling type systems where concurrency plays no part. Nonetheless, most programming languages do support concurrency and formal systems do exist. Whilst the individual rules for e.g. assignment, method call appear much as in their sequential counterparts they exist within a much richer framework. Such calculi e.g. [AY05] must maintain a pool of threads including code and context, as well as having some notion of thread scheduling (though this may often be made non-deterministic).

We fully expect effects to give non-interference results in a concurrent setting. After all, FX [GJLS87] was designed in order to support automatic parallelisation of code. For our model to consider concurrent execution we would have to extensively redesign it. It would doubtless become more complex but we believe we could still provide an abstract language independent description. Sequential results should still appear as special cases of the general concurrent result.

3.6.4 Alternative Definition of Non-interference

As mentioned in Section 2.1.1, even in the single threaded context there is an alternative definition of non-interference. Consider the following:

Definition 3.6.1

$$\begin{aligned}\gamma \models \mathsf{e}\#\mathsf{e}' \quad \Longleftrightarrow \quad & \pi \vdash \gamma\ ,\ \gamma \vdash \mathsf{s},\mathsf{h}\ ,\ \mathsf{e},\mathsf{s},\mathsf{h} \overset{\mathsf{h}^{\dagger}}{\leadsto}_{\pi} \mathsf{v},\mathsf{h}'\ ,\ \mathsf{e}',\mathsf{s},\mathsf{h} \overset{\mathsf{h}^{\ddagger}}{\leadsto}_{\pi} \mathsf{v}',\mathsf{h}'' \\ & \Longrightarrow \mathsf{e}',\mathsf{s},\mathsf{h}' \overset{\mathsf{h}^{\ddagger}}{\leadsto}_{\pi} \mathsf{v}',\mathsf{h}'''\ ,\ \mathsf{e},\mathsf{s},\mathsf{h}'' \overset{\mathsf{h}^{\dagger}}{\leadsto}_{\pi} \mathsf{v},\mathsf{h}'''\end{aligned}$$

In the above definition, rather than assuming the existence of sequential executions (e followed by e', as we did in Definition 3.3.1) we assume the existence of executions of each expression starting from the same heap. The expressions are non-interfering if, for each expression, an execution exists starting from the heap found from the previous execution of the other. The final heaps must be equal and, per expression, the two executions must give the same result and have the same required heap.

Although we do not see Definition 3.6.1 as superior, it would further validate our model if we could cater for this definition. Unfortunately Theorem 3.5.2 no longer holds with Definition 3.6.1. Once again this relates to the model of allocation that we assume. In the initial execution of e and e' (starting from h) both executions may allocate new sub-heaps but these sub-heaps might not be disjoint. We could strengthen Definition 3.6.1 to require that the heaps allocated in each of the assumed executions are disjoint and then the Theorem would hold. Obviously, we would much prefer to consider any arbitrary executions.

A model which used heap equivalence rather than equality, would allow us to prove non-interference of executions in the style of Definition 3.6.1 as well as Definition 3.3.1.

Definition 3.6.1 would be restated to say that rather than equal heaps (h''') the second executions of e and e' returned heaps that are equivalent. With the model in this format we would still be able to prove Theorem 3.5.2 with Definition 3.3.1.

3.6.5 Sophisticated Predicates

The example assertion language presented in Section 2.2.3.1 does not include any style of quantification. Quantification (even if implicit) is certainly desirable in a program logic, to describe class invariants, for example [LM04]. The details of the effects system of `Joe` effectively preclude quantifiers in predicates but, with a slight modification, we could allow them.

In `Joe`, object creation produces no effect. This is sound as `Joe` does not include explicit constructors and so object creation cannot alter pre-existing objects. Satisfaction of predicates that quantify over objects in the heap can be changed by the creation of a new object if it does not have the expected property. In a predicate language with typed quantification, the effect of a quantified predicate can be given as `under` the owner of the type, plus any other effect accumulated after the quantification. Now suppose we have such a quantified predicate which is `true` in some heap, and the effect of the predicate is disjoint from the write effect of an expression. Executing the expression could create a new object that invalidates the predicate. Thus non-interference does not hold.

We could easily modify `Joe` to remedy this. Let the effect of an object creation be to read and write under the owner of the new object. Thus all object creations would be recorded in the effect of the expressions. Let us return to our previous predicate and expression. When the expression is now executed it may still create new objects but not any that can change satisfaction of the predicate.

Whilst a solution is forthcoming in `Joe`, the model does not support such quantification. To prove Theorem 3.5.1 we require **LL1**, which allows the parts of the heap that are altered by the execution or not covered by the effect to be added back into the predicate satisfaction function. Unfortunately this property is not compatible with quantification. As we have seen above, adding arbitrary new objects to a heap can break satisfaction of a quantified predicate. The following axiom amalgamates and extends **LL1** and **LS3** and could allow predicates with quantification:

$$\gamma \vdash \mathsf{p} : \phi, \quad \gamma \vdash \mathsf{s}, \mathsf{h} * \mathsf{h}', \quad |\mathsf{h} * \mathsf{h}'|_{\mathsf{s},\phi} = |\mathsf{h}|_{\mathsf{s},\phi} = \mathsf{h}'' \quad \Rightarrow \quad [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}} = [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}''*\mathsf{h}'} = [\![\mathsf{p}]\!]_{\mathsf{s},\mathsf{h}*\mathsf{h}'}$$

Thus we have that any heap which does not contribute to the projection of the effect can be added without changing satisfaction. In order to get the non-interference result we desire we would also have to strengthen **LS4** so that all of the newly allocated heap (h_4) was included in the write effect.

Whilst this formulation would allow us to consider more sophisticated predicates and derive non-interference properties it moves away from the original intent of the model.

We wanted to capture some basic property of predicates and then add an effects system, independently, which exploited this property. Instead the new axiom ties the two together much more tightly. This may be unavoidable, it may be the case that when we consider more powerful predicates their properties become more inter-linked with the effect system describing them. This alternative axiom is a recent idea and requires further consideration.

Chapter 4

Ownership Domains with Effects

In this chapter we describe the Ownership Domains[AC04] system of Aldrich and Chambers and introduce our own version ODE. ODE has most of the original features of Ownership Domains plus an effects system inspired by `Joe`. We describe the advantages of Ownership Domains via example and show how ODE can provide more precise effects than `Joe`. In the second half of the chapter we give a formal description of ODE. In Chapter 5 we will prove that ODE is an instance of our model for languages and effects.

Parts of this chapter have appeared in the publication 'Towards an Effects System for Ownership Domains' [Smi05].

4.1 Ownership Domains

Ownership Domains [AC04, KA05] are an ownership type system which provides a more general and flexible form of ownership than previous systems. Ownership domains have been presented in a generalised form [KA05] but we work in the context of the original Java-like setting [AC04]. The rest of this Section describes Ownership Domains as presented by Aldrich and Chambers[AC04]. As we introduce our effects system, features we add will be described. Ownership Domains as described in this section form a subset of ODE, Section 4.5 gives a precise comparison of ODE with the original Ownership Domains.

Every object is contained within a single *domain*, its owner. Domains are declared in classes and for each instance of a class there are instances of each declared domain, immutably tied to that object. Domains belonging to the same object are disjoint i.e. they do not share any objects. References between domains are constrained by explicit, programmer granted permissions as well as a number of general rules.

In standard ownership type systems the rules governing which references (i.e. fields) are

```
class A<owner, d>
    extends Object<owner>
      assume owner -> d {
  domain a,b;
  link b -> d;
  Object <d> f;
  Object <this.a> g;
  Object <this.b> h;
}
```

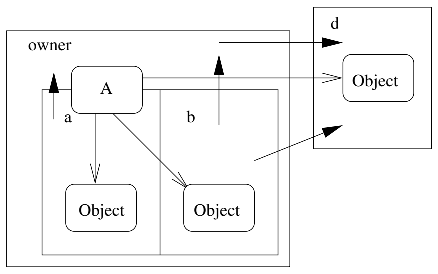


Figure 4.1: A class with ownership domains and a possible instantiation

permitted follow the ‘owners as dominators’ principle. An object may only reference objects it directly owns, objects which directly or transitively own it and objects directly owned by a transitive owner. Ownership domains are similar in that objects and domains form a tree structure but there is no fixed policy restricting references. The programmer may grant reference permissions between domains in a very flexible manner. Objects automatically inherit the permissions granted to their owner and in class definitions these permissions may be further delegated to local domains. A class may also grant permission for an external domain to access a local domain or for permissions between local domains. By this mechanism strong restrictions such as those of ownership types can be encoded but more liberal policies can also be employed. Encapsulation is provided as a class may not grant permissions between two external domains - a local domain must be either the receiever or the target of the granted permsission.

In Figure 4.1 we show a class, `A` and, through a diagram, a possible instance of it. In the diagram open headed arrows represent references (fields) and closed headed arrows permissions between domains. In class `A` the parameters to the class name i.e. `owner` and `d`, are ownership domains (indicated in the diagram with square cornered boxes, objects having round corners). The special parameter `owner` indicates the domain containing an instance of the class. `A` declares two domains, `a` and `b`, shown by the boxes adjoined to the `A` object.

The types of fields may be instantiated with any domains in scope i.e. parameters and locally declared domains (prefixed with `this.`). The domains used to instantiate a type must obey the assumptions declared in the class. Assumptions are of the form d->d' meaning objects in domain d have permission to refer to objects in domain d'. Domains may be explicitly linked together (granting permission for one to access the other) as in `link b -> d`.

The assumed link `owner -> d` allows objects in the `owner` domain to reference objects in the `d` domain, this permits the field `f`. Similarly, `link b -> d` permits objects in `b` to access objects in `d`. The fields `g` and `h` are allowed as an object can always refer to objects in its declared domains. Domain declarations may be marked `public`, in which case they may be referenced by any object which may access the declaring object without the need for an explicit link. The domains `a` and `b` may not refer to each other unless explicitly linked.

```
class Timetable<owner, projects>
    assume owner -> projects {
  domain durs;
  link owner -> durs;
  Duration<durs> d;
  Project<projects> p;
  Timetable<owner, projects> next;
  int delay(int x){
    this.d.delay(x);
    if(this.next!=null){
      this.next.delay(x);
    }
  }
  ...
}

class Duration<owner> {
  int start;
  int dur;
  void delay(int x) {
    this.start = this.start+x;
  }
  ...
}
```

```
class Project<owner> {
  public domain durs;
  Duration<durs> d;
  ...
}

class Employee<owner,projects>
    assumes owner -> projects {
  domain official, unofficial;
  link official -> projects;
  link unofficial -> projects
  final Timetable<official, projects> o;
  final Timetable<unofficial, projects> u;
  void delayo(int x) {
    this.o.delay(x);
  }
  void delayu(int x) {
    this.u.delay(x);
  }
  ...
}
```

Figure 4.2: Project management with ownership domains

4.2 Example

Figure 4.2 gives code, in a Java-like language with ownership domains, for a simple project timetabling programming. It is an expanded version of that in Figures 2.1 and 2.2 and shows some of the advantages of Ownership Domains compared to Ownership Types.

`Employee` objects keep a record of the projects they are working on in two `Timetable` objects which are contained in the local domains `official` and `unofficial`. A `Timetable` is a sequence of pairs of `Project` and `Duration` objects, indicating the project to be worked on and the time period for the work. Each `Timetable` object declares a domain `durs` which contains the relevant `Duration` object. `Project` objects are stored in the parameter domain `project` which allows them to be aliased among timetables.

Figure 4.3 shows a possible instantiation of the classes from Fig. 4.2. Each `Timetable` sequence is completely contained within a domain as are its `Duration` objects. As in Figure 2.2 Projects can be shared between employees but timetables cannot.

We also assume the predicate `nonOverlapping` as in Section 2.2 which takes a `Timetable` as an argument and is satisfied when the durations of no two elements of the `Timetable` overlap. We omit the details of the predicate language although something similar to that in Section 2.2.3.1 would suffice.

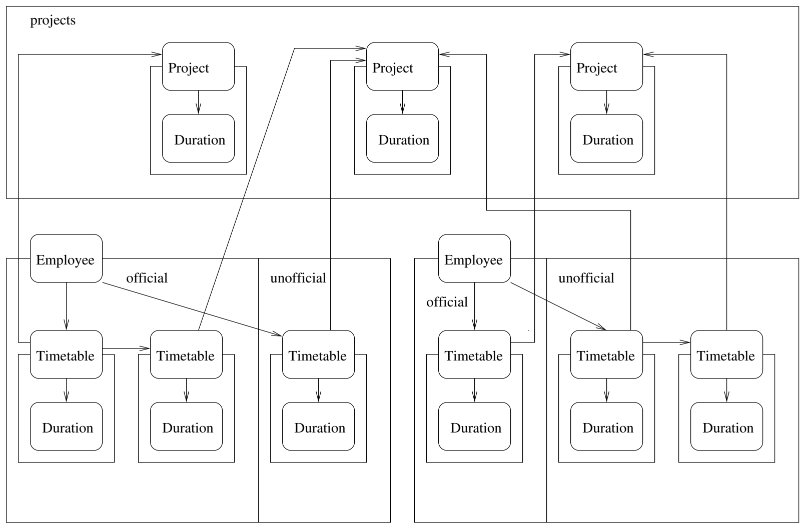


Figure 4.3: Instance of classes from Figure 4.2

Class	Method	Effect
`Duration<owner>`	`delay(int x)`	`rd this wr this`
`Timetable<owner, projects>`	`delay(int x)`	`rd owner.under` `wr owner.under`
`Employee<owner, projects>`	`delayo(int x)`	`rd this+this.official.under` `wr this.official.under`
`Employee<owner, projects>`	`delayu(int x)`	`rd this+this.unofficial.under` `wr this.unofficial.under`

Figure 4.4: Effects for methods

4.2.1 Effects for the Example

In Figure 4.4 we give effects for each of the methods described above. Effects for expressions (and methods) have the form `rd` ϕ `wr` ϕ' where ϕ is the read effect and ϕ' is the write effect. As discussed in Section 2.2.2.1 the read effect contains the write effect.

The `delay` method of `Duration` has effect `rd this wr this` meaning that only the receiver of a call to `delay` is inspected/modified by the call.

The effect of `delay` in the `Timetable` class is more complicated. The method may make a recursive call to the `delay` method of the next `Timetable` in the sequence. The declared effect must describe reads/writes to the current receiver (and the `Duration` object which is delayed) as well as the equivalent effect on the rest of the sequence. The effect `rd owner.under wr owner.under` achieves this. `owner.under` means all the objects in domain `owner` and all those transitively contained i.e. in domains nested inside `owner`. This includes all the `Timetable` objects in the sequence (since they all have the same owner) as well as the `Duration` objects which are nested inside.

The effect declarations for the two delay methods in `Employee` must include the effects of the `Timetable` `delay` call they make and add the effect of reading their own field. For example the effect of `delayo` is `rd this+this.official.under wr this.official.under`;

Predicate	Effect
`nonO(Timetable<owner,projects> t)`	`owner.under`

Figure 4.5: Effects for methods and predicates

`this.official.under` is equivalent to `owner.under` in `Timetable` (as the field `o` has type `Timetable<official, projects>`) and `this` records the read of field `this.o`.

As in Section 2.2.3.2 we calculate an effect for the predicate `nonOverlapping` in a similar way to calculating the effect of an expression. We give this effect in Figure 4.5. As before effects have the form ϕ as they only have a 'read' part. The predicate needs to compare all the `Duration` objects of each `Timetable` object in the sequence beginning at `t`, so must read fields of the `Timetable` objects and the `Duration` objects. The effect `owner.under` includes all the `Timetable` and `Duration` objects in the sequence and so is an appropriate effect for `nonOverlapping`.

4.2.2 Disjointness and Non-interference

We now introduce disjointness of ODE effects and show how they can help deduce non-interference in the example. Throughout we assume that we are working in the context of an environment where `a` and `b` are final variables of type `Employee<owner,projects>`. Thus `a` and `b` are owned by the same domain and their timetables point to projects in the same domain. We assume further that they are known not to be aliases.

4.2.2.1 `a.delayo(5)#b.delayo(23)`

The expressions `a.delayo(5)` and `b.delayo(23)` have effects as follows (based on the types of the variables in our assumed environment and the effects we have given for the methods):

```
 a.delayo(5):rd a+a.official.under wr a.official.under
b.delayo(23):rd b+b.official.under wr b.official.under
```

Recall that two expressions are non-interfering if the read effect of each expression is disjoint from the write effect of the other. Since we know `a` and `b` to not be aliases we know that the domains `a.official` and `b.official` are distinct, they cannot be nested, since `a` and `b` have the same owner. Thus `a.official.under` and `b.official.under` must also be disjoint. Further still `a` cannot be in `b.official.under` since `a` and `b` have the same owner and all objects in `b.official.under` are nested within `owner`. By distributivity since both `a` and `a.official.under` are disjoint from `b.official.under` then `a+a.official.under` is also disjoint from `b.official.under`. Thus `a.delayo(5)` and `b.delayo(23)` are non-interfering. In fact, any combination of delays on `a` and `b` will be disjoint as all the timetables of `a` and `b` are separate.

4.2.2.2 `a.delayo(5)#a.delayu(17)`

The effects are as follows:

```
 a.delayo(5):rd a+a.official.under wr a.official.under
a.delayu(17):rd a+a.unofficial.under wr a.unofficial.under
```

These expressions are also non-interfering. The argument is largely the same as above except that `a.official.under` and `a.unofficial.under` are disjoint because `a.official` and `a.unofficial` are distinct domains (different names) and are not nested (they belong to the same object).

4.2.2.3 `nonOverlapping(a.o)#a.delayu(17)`

The predicate `nonOverlapping(a.o)` and `a.delayu(17)` are non-interfering. This is because the write effect of `a.delayu(17)`, `a.unofficial.under` is disjoint from the effect of `nonOverlapping(a.o)`, `a.official.under`. These effects are disjoint by the same argument as in Sec. 4.2.2.2. The non-interference of `nonOverlapping(a.o) a.delayu(17)` with the rule of constancy and an appropriate Hoare logic would allow the following deduction:

$$\frac{\{P\}\texttt{a.delayu(17)}\{Q\} \quad \texttt{nonOverlapping(a.o)}\#\texttt{a.delayu(17)}}{\{P \wedge \texttt{nonOverlapping(a.o)}\}\texttt{a.delayu(17)}\{Q \wedge \texttt{nonOverlapping(a.o)}\}}$$

4.2.2.4 Comparison with `Joe`

In a `Joe` version of this program we would be able to deduce some, but not all of the above properties of non-interference. In `Joe` we would not be able to separate, with local domains, the official and unofficial timetables. They would be contained per `Employee` object but we would not be able to reason that they are distinct.

For objects `a` and `b` any combination of delays is non-interfering as the unofficial and official timetables are confined per object. Non-interference also holds between calls to `nonOverlapping` and `delay` for timetables not belonging to the target of `delay`.

Because there is no longer any separation between the official and unofficial timetables we can no longer deduce any non-interference of `delayo` and `delayu` when called on the same `Employee`. Similarly we cannot detect non-interference of `nonOverlapping` applied to the official timetable of an employee and a call to to `delayu` on that same employee.

4.3 Effects

In this section we describe in more detail the effects system of ODE. We give the grammar for effects and describe the rest of the system informally (the formal treatment is included

$$
\begin{array}{lcl}
\psi & ::= & \texttt{rd}\ \phi\ \texttt{wr}\ \phi' \\
\phi & ::= & \theta \mid \theta.\texttt{under} \mid \phi{+}\phi' \mid \texttt{emp} \\
\theta & ::= & d \mid \mathit{path} \mid \mathit{path}.\texttt{all} \\
d & ::= & p \mid \mathit{path}.p \mid \texttt{world} \\
\mathit{path} & ::= & z \mid \mathit{path}.f
\end{array}
$$

Figure 4.6: Grammar of ODE effects

in Section 4.6). The effects describe heap structure in terms of the tree structure of domains (where nesting of domains forms a tree). Our approach is based on `Joe` and uses the same technique of projecting 'downward' from a point in the tree. The grammar of effects is given in Figure 4.6; z ranges over program variables, f over field identifiers and p over the names of parameter domains and locally declared domains.

Each effect refers to a set of objects as follows:

d is the set of objects owned by domain d, which could be a parameter to the current receiver, a local domain of an object chosen by a path or the top level domain `world`

path is just the object denoted by the sequence of field accesses path

$\mathit{path}.p$ the set of objects owned by the local domain p of the object pointed to by path

$\mathit{path}.\texttt{all}$ refers to all the objects in all the local domains of the object denoted by path

$\theta.\texttt{under}$ is the set of all the objects below the object(s) denoted by θ in the ownership tree. This includes the object(s) in θ.

`emp` is the empty shape, and represents no objects

$\phi{+}\phi'$ the union of ϕ and ϕ'

For matters of soundness, all paths mentioned in effects must be final i.e. a sequence of accesses to final (immutable) fields rooted at a final variable. The type system will ensure that, after initialisation, final fields cannot be updated. Thus the object denoted by a final path will be the same throughout execution. If this were not the case, the meaning of an effect before and after a computation could be different and the system would not be sound. Similar restrictions are required in both the effects of Joe, where all local variables are final [CD02] and the original domain system [AC04] where final paths are required to make type instantiations sound.

4.3.1 Disjointness of Effects

Judgement of disjointness for our effects is formed from a combination of typing, syntactic comparison and structural properties of trees. We describe a selection of rules from the system.

Arbitrary pairs of domain parameters cannot be detected disjoint as we have no knowledge of their position in the tree, only of the links between them. Simple paths can be shown disjoint when their types are incompatible i.e. neither is a subtype of the other, since they cannot refer to the same object. Further from this, for any disjoint paths, $path$ and $path'$ any local domains of these paths, $path.p$ and $path'.p'$, must be disjoint since domains belong to only one object and these objects are known to be disjoint. Any two $path.p$ effects where the domain names (p) are distinct must be disjoint simply since the domains have different names.

Structural shapes i.e. θ.`under` are more difficult to judge disjoint, as we must be certain that neither shape is nested inside the other. The effects $path.p$.`under` and $path.p'$.`under` are disjoint when $p \neq p'$ since the domains are distinct and at the same level in the ownership tree. The trees undeneath them are parallel but do not overlap.

General rules such as distributivity where ϕ is disjoint from $\phi{+}\phi'$ if it is disjoint from ϕ' and from ϕ'' and rules allowing subsumption of effects complete the system.

4.3.2 Joe

Our effects system is based on that of Joe [CD02]. Both systems exploit the underlying tree structure of ownership types/domains. The grammar for Joe effects is as follows:

$$\begin{array}{lcl} \psi & ::= & \texttt{rd}\ \phi\ \texttt{wr}\ \phi \\ \phi & ::= & \emptyset \ \mid\ p.n \ \mid\ \texttt{under}(p.n) \ \mid\ \phi \cup \phi \end{array}$$

In the effects of Joe the 'root' of each effect (p in p.n) is a local variable or ownership context. The equivalent 'roots' in ODE are richer as we allow final paths, (rather than just variables) and domain expressions (d, $path.d$, $path$.`all`) (rather than just parameter domains). A subtle but important distinction is that, in `Joe`, p whether a variable or a parameter refers to an object (which owns other objects). In ODE, because domains and objects are distinct there is a semantic difference between a $path$ which refers to a single object and a domain expression (d or $path.p$, for example) which refers to a set of objects. In both systems an inclusion relation is used in the effects system to reason when a root is higher in the tree than another. In Joe this is a relation on pairs of objects but in our system it is a relation between pairs of objects (paths), pairs of domains and between domains and objects.

We use the notion of 'under' from Joe (where it is written $under(target)$) to describe all objects which are below the objects described by $target$ in the ownership tree. `Joe` includes a more fine grained effect called bands (p.n), which describes strata within the tree e.g. $z.1$ is the objects owned by z whereas $z.2$ describes the objects owned by those in `z.1`. Such effects could be included for ownership domains but we omit them here for brevity and as they do not add anything to our examples. Treatment of bands could be a little more interesting than in `Joe` if we allowed bands to be selected on both paths and domains. Selection of all local domains e.g. $path$.`all` is equivalent to the Joe effect

z.1.

Naturally some of our judgements of effect disjointness etc. follow from those seen in Joe. We have rules equivalent to each of those in Joe as well as additional rules for dealing with features new to our contribution.

4.4 Further Issues

So far we have discussed ownership domains in general and have presented a simple example. Whilst we have not excluded any part of the system we have not seen the full power of ownership domains. Ownership domains can provide good solutions to programming problems that have proved hard in other ownership type systems. Unfortunately the same features that admit these solutions also make it hard to give accurate effects to methods and expressions in these programs.

In this section we present an example for which we cannot calculate useful effects. We proceed to extend ODE, by augmenting the effects and constraints, which addresses this deficiency.

4.4.1 Iterators

In [AC04] an interesting solution is given to a common problem with ownership type systems. Whilst linked lists are the paradigmatic example of the power of ownership types, iterators over such lists are problematic. Ownership types allow the internal structure i.e. link nodes to be encapsulated by making them owned by the list object. This ensures that i) no two lists can share a representation (though they may contain the same data) ii) the list object controls all access to the list representation. Unfortunately iterators [GHJV95] require access to the internal representation of the list they iterate over. Various solutions have been proposed to this problem with varying degrees of success[1].

Figure 4.7 shows the ownership domains solution to list iterators as presented in [AC04]. In this solution each `List` has a domain for holding the list representation (`list`) and one for iterators (`iters`). The `iters` domain is public so that clients can refer to iterators of the list. The `getIterator` method of `List` creates a specialised `ListIterator` but it is returned with only the generic `Iterator` interface type. The `Iterator` interface takes only two domain parameters, one for the owner and one for the domain where the list data is stored. The specialised `ListIterator` takes a further parameter, `list`, the domain containing the representation of the list it iterates.

[1]Clarke and Drossopoulou [CD02] allow stack variables to break ownership boundaries, allowing an iterator to be created and used within a method body. This has limited usefulness since the reference to the iterator cannot be stored (in a field) by an external client. Boyapati et al. [BLS03] follow a suggestion of Clarke [Cla01] and allow inner classes to refer to owned objects of the enclosing instance but for instances of such inner classes to be passed out to clients.

```
class List<owner, elems>
    assumes owner->elems {
  domain list;
  public domain iters;
  link list->elems, iters->elems,
    iters->list;
  Cons<list, elems> head;
  void add(Object<elems> o){...}
  Iterator<iters, elems> getIter(){
    return new ListIterator<iters, elems,
          list>(head);
  }
}

class Cons<owner, e> assumes owner->e {
  Object<e> obj;
  Cons<owner,e> next;
}

interface Iterator<owner, elems> assumes
      owner->elems {
  Object<elems> next();
  boolean hasNext();
}
```

```
class ListIterator<owner, elems, list>
    implements Iterator<owner, elems>
    assumes owner->elems, owner->list,
         list->elems {
  Cons<list, elems> current;
  boolean hasNext(){...}
  Object<elems> next() {
    Object<elems> obj = current.obj;
    current = current.next;
    return obj;
  }
}

class Client<owner> {
  domain d;
  final List<d,d> l = new List<d,d>();
  void run(){
    Object<d> obj = ...
    l.add(obj);
    Iterator<l.iters, d> i = l.getIters()
         ;
  }
}
```

Figure 4.7: Lists and Iterators in Ownership Domains

Class	Method	Effect
`Iterator<owner,elems>`	`next()`	`rd this+`? `wr this`
`ListIterator<owner,elems,list>`	`next()`	`rd this+list wr this`

Figure 4.8: Partial effects for Iterator methods

4.4.2 Effects or Iterators

In Figure 4.8 we attempt to give effects for the methods in Fig. 4.7. Whilst we can easily calculate an effect for `next` in `ListIterator` we cannot calculate a satisfactory effect for `next` in `Iterator`.

The effect of `next` in `ListIterator` should be obvious. The read effect must include `list` as it reads the `next` field of `current`, which is owned by `list`, also it reads and writes the receiver's field `current` so the effect must include `this`.

The declared effect of an overriding/implementing method must be a sub-effect of the declaration in the superclass (c.f. covariant changes to return types). Thus, the effect of `next` in `Iterator` must include `list`. This poses a problem since in the `Iterator` interface there is no way to mention `list` (since it is not a parameter to the type and there is no structural way to refer to it).

The trivial effect `world.under` would be a sound effect (since it describes the entire heap) but it is also useless as it is only disjoint with the empty effect.

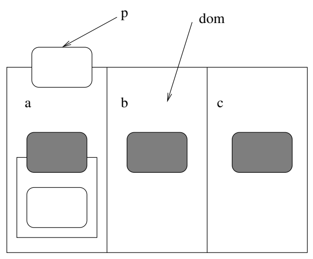


Figure 4.9: The sibling effect

Class	Method	Effect
`Iterator<owner,elems>`	`next()`	`rd this+owner.siblings wr this`
`ListIterator<owner,elems,list>`	`next()`	`rd this+owner.siblings wr this`

Figure 4.10: Full effects for Iterator methods

4.4.3 Constraints and More Expressive Effects

To overcome our problem in calculating effects for iterators we propose additional constructs to our effects and analogous constraints which will be added to the assumes clause of classes.

We propose the addition of a term `.siblings` to our effects, so that:

$$\phi \quad ::= \quad \ldots \mid d.\texttt{siblings}$$

This effect describes all domains belonging to the same object as d. Figure 4.9 shows this graphically where p is a path referring to the indicated object and *dom* is an alias for the indicated domain (if *dom* were a parameter say). The effects $p.a.$`siblings` and *dom*.`siblings` are equal and refer to the three shaded objects, that is, all the objects in all the domains belonging to p. The observant reader may notice that *path*.*d*.`siblings` is equivalent to *path*.`all`. In practise siblings effects will often be converted to *path*.`all` through subsumption.

Now we can give an effect to both `next` methods as shown in Figure 4.10. This would appear to be sufficient but in fact we need a little more. Whilst the effect we have given is appropriate for iterators in the context of Figure 4.7, we have no way of knowing that the domains `list` and `owner` of `ListIterator` are siblings. It is, of course, the programmer's intent that `ListIterator` objects are only ever created by `List` objects, and that `owner` and `list` will only ever be instantiated with sibling domains but that is neither enforced nor statically detectable.

To cope with this we suggest the introduction of further constraints between domains in the `assumes` clause of a class. A constraint of the form p `sibling` p', meaning that parameter domain p is a sibling domain of parameter domain p', will be checked when the type is instantiated (just like the link assumptions). Adding `owner sibling list` to the assumes clause of `ListIterator` guarantees that `owner` and `list` are always sibling domains and that the effect of `next` is sound.

4.4.3.1 Statically Checking Siblings

Implementing the sibling constraint does not require any fundamental changes to the machinery of ownership domains, only extension. We add rules for checking the constraint in the assumes clause and rules for calculating subshapes involving `siblings`.

Consider class `List` in Figure 4.7 and assume the addition of `owner sibling list` to the `assumes` clause of `ListIterator`. In the `getIter` method where `ListIterator` is instantiated, the type system will check that `this.iters` and `this.list` (the instantiations of `owner` and `list`) are siblings. This is straightforward as `iters` and `list` are both declared in `List`. In general we can deduce that for any *path*, *path.d* `sibling` *path.d′*. Sibling constraints can also be propagated through the `assumes` clause, just as `link` is.

Calling method `next` on `i` in the `run` method of `Client` has the effect `rd i+l.iters. siblings wr i`. By subsumption this becomes `rd i+l.all wr i` (which is equivalent) and rules not involving `siblings` can then be used to reason about the effect.

4.4.3.2 Further Constraints

In the interest of assisting the calculation of accurate effects and checking disjointness of effects we further augment the constraints of ODE. Judgment of containment of domains and paths is already necessary to support our effects system (as it is in `Joe`). We add the ability to constrain the containment of parameter domains using the syntax $p <\!+\, p'$ and $p {<}{*} p'$ meaning that p is contained within p' in the first case and that p is contained within or equal to p' in the second case. Boyapati et al. include similar constraints in [BLS03]. In the non-reflexive case these constraints would allow us to judge two parameter domains disjoint, an improvement over the situation discussed in Section 4.3.1.

4.4.4 Modularity

In effects systems for object-oriented languages, some consideration must be given to the modularity of effects. Just as method return types must be covariant with inheritance, effects of methods must be covariant i.e. more precise[2]. If effects were not covariant, the system would be unsound, as a virtually dispatched method might read or write state not mentioned in the effects of its statically known counterpart.

As such, effects of methods must be sufficently liberal to allow programmers to add behaviour in subclasses. Boyland and Greenhouse [GB99] note that inclusion of the write effect in the read effect assists in this aim. ODE also promotes modularity through the `under` and `siblings` effects. `under` effects allow overridden methods to read or write

[2]In the formal system in Section 4.6 for convenience and brevity we require effects annotations to be identical in overridden methods.

deeper in the ownership hierarchy, possibly at levels not existing in the inherited scope. As seen in the previous section the `siblings` effect also facilitates modularity by allowing specification of effects occuring in domains which cannot otherwise be named (since they are not in scope at the higher level). As the iterator example showed, without the `siblings` effect, important idioms of Ownership Domains which employ modularity could only be given trivial effects. However, the `siblings` effect does require a little insight from the programmer. The use of `siblings` in the iterator interface might not make much sense, until it is viewed alongside the implementation. As mentioned previously, we believe the iterator employs a general idiom of ownership domains. Experienced programmers would become familiar with this pattern and understand its use when implementing interfaces.

4.4.4.1 Effect Inference vs. Programmer Annotation

ODE is designed with the intention that programmers provide annotations for the effects of methods. Validity of effects is checked with the rules in Section 4.6.2.9. Unfortunately, declaring effects is a further burden on the programmer. In the context of ODE this burden comes on top of a more complicated type system which also expects programmer annotation.

Effect inference is certainly desirable and has been provided in other systems [TJ92]. In Hu's thesis [Hu05] an algorithm was described to infer effects for the Joe language (for programs with ownership type annotations). The technique could certainly be adapated to the effects of ODE (since they are closely related). However, Hu's thesis does not consider inheritance. In a language with inheritance the inference algorithm would need to check that effects inferred for methods are sufficent to cover the effects of all overriding methods in subclasses. Inference of ownership types has been investigated but remains open [BR01, AKC02, Syk06].

Our opinion is that the additional burden of adding effects annotations is not too great. Moreover it ccompels to programmer to consider where the effect of an expresson will be. This, in turn, encourages the programmer to think about modularity and consider what parts of the heap overridden methods may need to access. We believe that explict consideration of effects the design of the ownership structure of the program, leading to better encapsulation and more useful effects.

4.5 Comparison of ODE and Featherweight Domain Java

In [AC04] Aldrich and Chambers present two versions of Ownership Domains: an informal presentation based on their AliasJava [CD02] language in the first half of the paper and a reduced formal system in the second. The formal system - Featherweight Domain Java (FDJ) - is an imperative calculus in the style of FJ [IPW99].

ODE differs from both the presentations in [AC04]. ODE extends Ownership Domains in the addition of effects and constraints on parameter domains. The syntax of ODE blends that of `Joe` and FDJ. This is motivated by the relationship between `Joe` effects and ODE effects as the expression syntax lends itself easily to the calculation of effects. However, in other areas, the FDJ style simplifies the presentation and we adopt it as appropriate.

The AliasJava version of Ownership Domains features immutable `final` fields as described in Section 4.3. This allows domains to be described by syntax of the form `z.f.g.d` i.e. a sequence of accesses to final fields with a domain name appended. FDJ dispensed with final fields (only allowing domains to be described by parameters, `world` or `this`.p for local domains of the receiver) but we include them in ODE. Inclusion of final fields has the benefit of allowing more programs to be typed and expression of the compelling example of Iterators shown in Section 4.4. Moreover final fields allow more precise effects to be expressed. Use of an object referred to by a final path can be recorded precisely rather than needing to be covered by a larger effect. The fields used must however be final or the meaning of the effect can change during execution and the system would not be sound. Inclusion of final fields has led to a considerably more complex constructor syntax and the need for supporting theory to prove soundness.

In ODE we remove a significant restriction from FDJ. Expression Link Soundness states that the receiver must have permission to access every object that is visible to it. In ODE this would mean that the receiver would need permission to access every object that is the result of a sub-evaluation. `Joe` made the same liberalisation of ownership types, allowing references on the stack to break ownership boundaries but maintaining the ownership structure in the heap. Although this allows the private domains of an object to be infiltrated, because of the effects systems of `Joe` and ODE it is not possible for this to happen invisibly. Any changes made will be noted in the effects. We can easily enforce Expression Link Soundness by adding a check that `this` has permission to access the owner of the result in each typing rule. Since this merely reduces the number of typable expressions ODE would remain sound. As such we prefer to be more liberal, obtaining a more general soundness result for our effects.

FDJ is a small step calculus whilst ODE has large step semantics. We choose large step semantics for compatibility with the model as well as personal preference. We fully separate the runtime and static systems. In small step systems (and some large step systems[CD02]) the same rules are usually used for e.g. typing both static and runtime entities. We find a system where these concerns are separated easier to understand (if somewhat more verbose) as well as closer to the realities of static typing in the compiler.

4.6 Formal System

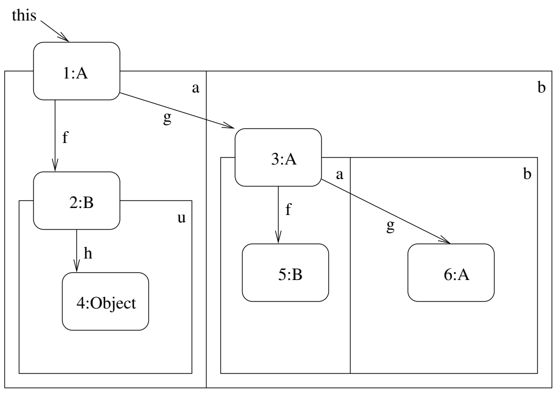
We now present the formal system of ODE.

Throughout, we refer to the example program in Figure 4.11 to clarify and motivate.

```
class A<o,c,d> extends Object<o>
  assumes o->c, o->d, o<c{

      domain a;
      public domain b;
      link this.a -> this.b, this.b
           ->c, this.b->d
    final B<this.a,this.b> f;
      final A<this.b,c,d> g;
}

class B<o,t> extends Object<o>
  assumes o->t, o sibling t{
      domain u;
      link u->o;
      link u->t;
      final Object<this.u> h;
}
```



$\gamma = \texttt{this} \mapsto \texttt{A<o,c,d>}$

Figure 4.11: Sample ODE program

The figure gives the text of a program π, a possible instance of its classes (in a diagram) and a sample type environment, γ to provide a context for discussion.

4.6.1 Syntax

Our syntax is a mixture of the Ownership Domains syntax found in [AC04] and that of `Joe` [CD02]. The syntax of ODE is given in Figure 4.12.

In Figure 4.12 and hereinafter the following variables are used:

k, k', k_1, ...	class names
f, f', f_1, ...	field names
m, m', m_1, ...	method names
p, p', p_1, ...	domain names
x, x', x_1, ...	local variables

ODE programs are made up of a number of class definitions. Class definitions include the name and parameters (to be instantiated with domains) of the class, the superclass (and the instantiation of its parameters), a list of constraints between parameters which must be satisfied in all instances of the class, a list of domain declarations, a list of `link` statements granting permissions between domains, a list of field definitions, a constructor and a list of method definitions. For simplicity we assume the formal domain parameters of the superclass to be a prefix of the parameters of the class being defined. Note also that we differentiate class names (k) from general class names (c) which include `Object` [3]. We will later disallow cycles in the class hierarchy, `Object` is included to provide a root. `Object` is assumed to take one ownership parameter (its owner) and have no methods, fields or domains.

Constraints between domains take five forms. Satisfaction of these constraints is deduced

[3]From here on we shall mostly use c for simplicity even if it is unnecessarily general.

π	$::=$	$\overline{cdef}$
$cdef$	$::=$	$\texttt{class}\ k\texttt{<}\overline{p},\overline{p'}\texttt{>}\ \texttt{extends}\ c'\texttt{<}\overline{p}\texttt{>}\ \texttt{assumes}\ \overline{con}\ \{\overline{dom}\ \overline{lnk}\ \overline{fd}\ constr\ \overline{md}\}$
c	$::=$	$k \mid \texttt{Object}$
con	$::=$	$p\,\$\,p'$
$\$$	$::=$	$\texttt{->} \mid \texttt{<} \mid \texttt{<+} \mid \texttt{<*} \mid \texttt{sibling}$
$constr$	$::=$	$t(\overline{t'\ x})\{\overline{\texttt{this}.f\ =\ cexp}\}$
fd	$::=$	$t\ f$
md	$::=$	$t\ m(t'\ x)\texttt{rd}\ \phi\ \texttt{wr}\ \phi'\{e\}$
lnk	$::=$	$\texttt{link}\ d\ \texttt{->}\ d'$
dom	$::=$	$[\texttt{public}]\ \texttt{domain}\ p$
b	$::=$	$\texttt{null} \mid \texttt{new}\ t(\overline{z}) \mid z.f \mid z.f{=}z' \mid z.m(\overline{z})$
e	$::=$	$\texttt{let}\ x = b\ \texttt{in}\ e \mid z \mid \texttt{null}$
ε	$::=$	$b \mid e$
$cexp$	$::=$	$z \mid \texttt{new}\ t(\overline{x})$
x	$::=$	$\texttt{this} \mid x$
t	$::=$	$[\texttt{final}]\ c\texttt{<}\overline{d}\texttt{>}$
d	$::=$	$p \mid path.p \mid \texttt{world}$
$path$	$::=$	$z \mid path.f$
ψ	$::=$	$\texttt{rd}\ \phi\ \texttt{wr}\ \phi'$
ϕ	$::=$	$\theta \mid \theta.\texttt{under} \mid \phi{+}\phi' \mid \texttt{emp}$
θ	$::=$	$d \mid path \mid path.\texttt{all} \mid d.\texttt{siblings}$

Figure 4.12: Syntax of ODE

in the static and runtime systems as described in Sections 4.6.2.2 and 4.6.3.2. ermissions ($- >$) between two domains require that the domain on the left hand side has permission to access the domain on the right hand side. Containment relations ($<$, $<+$, $<*$) require that domains are contained directly, transitively and reflexively and transitively respectively. The final constraint type is that the two domains are siblings i.e. owned by the same object.

Field declarations consist of a type and a field name. Types may include a `final` tag to declare the field as immutable. In other declarations i.e. method return and parameter types, well formedness will require that the types are not final. Method declarations consist of a return type, argument name and type, an effect declaration (the effect counterpart of the return type) and the method body.

Local domains of the class are declared [`public`] `domain` p where p is the name of the domain being declared. Domains may be optionally declared `public`. A link between two domains, `link` d `->` d' grants permission for the domain on the lhs to access that on the rhs (up to restrictions described in Section 4.6.2.10). Variables d, d' etc. range over domain expressions which occur in three kinds. Parameter domains of the current receiver are referred to by name, p. Local domains of objects are referred to in the form $path.p$ with $path$ being a sequence of field names (rooted at a program variable) referring to the object and p being the declared name of the domain. The universal domain `world` is a valid domain name in any scope.

In similar calculi, constructors are often elided completely as in [DE97] (with fields being automatically set to `null`) or trivialised as in [IPW99]. Because we include final fields, we require a mechanism to initialise their value and so we include a limited form of constructor. Constructors consist of a sequence of special field assignments (which do not conform to the syntax of standard field assignment in ODE). General computation is not allowed in the constructor as we wish to keep `new` statements effect free (see Section 4.6.2.9 for a complete discussion). To achieve this we limit the right hand side of each assignment to include only a local variable (a parameter to the constructor or the receiver), `null` or a `new` statement. As a matter of convenience in the formal system we require constructors to assign to all fields of the class (and superclasses) rather than invoke a *super* call.

Computations in ODE follow the pattern in `Joe` and are broken into two kinds, expressions (e) and blocks (b). Expressions consist of program variables and `let` expressions of the form `let` $x = b$ `in` e where x is a new, immutable, local variable which is assigned the value of b. e is then evaluated with the stack extended with x. A third kind, ε, is the union of blocks and expressions and is introduced purely for use in the signatures of relations and in statement of theorems etc.

Blocks contain the familiar object-oriented operations of object creation, field lookup, field assignment and method call. It should be noted that all these operations use local variables where one might expect to see arbitrary subexpressions. We borrow this approach from `Joe` as it allows us to give types to more expressions and allows calcualtion of more accurate effects. We will explain further in Sections 4.6.2.5 and 4.6.2.9.

Types in ODE are of the form $c{<}\overline{d}{>}$ where c is a class name and $\overline{d}$ are the actual parameters of the type. Types may also be prepended by the `final` modifier. In programs `final` is used in field types, as described above. When typing expressions etc. only non-final fields may be assigned to (outside of the constructor). Path expressions, as appearing in effects and domain expressions, must be immutable and `final` types come into play here (see Section 4.6.2.5).

Full effects (ψ) are expressed as pairs of basic effects (ϕ). The syntax is as described previously.

4.6.2 Static System

We now present the static system of ODE. The main rules in the static system are for typing of expressions, calculating the effect of expressions and well-formedness of programs. Additional helper rules determine subtypes and subeffects as well as disjoint effects (those that refer to distinct parts of the heap) and types. Further rules deduce the relations of permission and containment between pairs of domains or objects. First, we present lookup functions which will be used throughout the system.

We will motivate many of the rules by reference to the examples in Figures 4.1 and 4.11.

4.6.2.1 Lookup Functions

Figure 4.13 gives the definitions for the various lookup and helper functions used in ODE. The functions retrieve information, such as method definitions or the domain assumptions made by a class, from the program for use in the type/effect rules of ODE. The functions are mostly standard or follow from [AC04] but our presentation is a little different in places.

For ease of presentation we make a few assumptions not explicit in the presentation. Most of the functions take as arguments the program and a type. In some contexts functions may be called with types that may be `final` but since these functions are concerned only with lookup this should not affect the result. For brevity we assume all type arguments are not `final` and make the necessary adjustments implicit.

Where the type argument is of the form $c{<}\overline{d, d'}{>}$ we assume that class c is defined in π with definition as shown in Figure 4.13. Moreover we assume that $\overline{d, d'}$, the parameters to the type, are appropriate for c i.e. there are the same number of domains in $\overline{d}$ and $\overline{p}$ and in $\overline{d'}$ and $\overline{p'}$. If these conditions are not met then the function application is not defined.

In many of the definitions, new variables `that` and $x^\dagger$ are used in substitutions to avoid variable capture. These are reserved variables and do not form part of the program syntax. These are necessary to distinguish `this` in a class definition from `this` occuring as part of a type parameter, which are not, in general, the same. For example, the declared type of a field might involve `this`, meaning the object with the field but `this` might occur in the actual type of the object (where it means the receiver in the context where the field is being used). To counter this we replace `this` in the declared type with `that` and then perform further substitutions on the results of the function. The same difficulty could occur with method parameters so we introudce $x^\dagger$ similarly. We will explain further with an example in Section 4.6.2.2.

The function $fields$ takes a program π and a type and returns a list of the field/method definitions in the given class and superclasses (with a recursive call). A substitution is applied to the definitions to convert formal parameters to actuals. At the point of application a further substitution will be applied to adjust references to the receiver in types. The *methods* function performs a similar function for methods but returns the definition of a particular method (due to issues of overriding). Similarly the *constr* function returns the constructor definition from the class with a substitution on type parameters (since there is no receiver to substitute).

The remaining functions are concerned with domains and linking. The *public* function takes as arguments a program π, a type $c{<}\overline{d, d'}{>}$ and a domain name p. The function returns *true* if c (or a super-class) declares a local domain called p which is marked `public`. It returns $false$ otherwise. The *owner* function simply returns the owner of the argument type, that is, the first parameter domain.

The function *assumptions* returns the assumptions (of links between domains and also

$$
\begin{array}{lcl}
fields(\pi, c\texttt{<}\overline{d, d'}\texttt{>}) & = & \overline{fd}\ [\texttt{that}/\texttt{this}][\overline{d}, \overline{d'}/\overline{p}, \overline{p'}],\ fields(\pi, c'\texttt{<}\overline{d}\texttt{>}) \\
fields(\pi, \texttt{Object}\texttt{<}d_1\texttt{>}) & = & \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
methods(\pi, c\texttt{<}\overline{d, d'}\texttt{>}, m) & = & \begin{cases} t_r\sigma\ m(\overline{t}\sigma\ \overline{x})\ \psi\sigma\ \{\ e\ \} \\ \quad \text{where}\ \ t_r\ m(\overline{t\ \overline{x})\ \psi\ \{\ e\ \}}\ \in\ \overline{md} \\ \quad \text{and}\ \sigma = [\texttt{that}, \overline{x^{\dagger}}/\texttt{this}, \overline{x}][\overline{d}, \overline{d'}/\overline{p}, \overline{p'}] \\ methods(\pi, c'\texttt{<}\overline{d}\texttt{>}, m)\ \text{otherwise} \end{cases} \\
methods(\pi, \texttt{Object}\texttt{<}d_1\texttt{>}, m) & = & \perp
\end{array}
$$

$$
cons(\pi, c\texttt{<}\overline{d, d'}\texttt{>}) \quad = \quad constr[\texttt{that}/\texttt{this}][\overline{d}, \overline{d'}/\overline{p}, \overline{p'}]
$$

$$
\begin{array}{lcl}
public(\pi, c\texttt{<}\overline{d, d'}\texttt{>}, p) & = & \begin{cases} true & \text{if}\ \texttt{public domain}\ p\ \in\ \overline{dom} \\ public(\pi, c'\texttt{<}\overline{d}\texttt{>}, p) & otherwise \end{cases} \\
public(\pi, \texttt{Object}\texttt{<}d_1\texttt{>}, p) & = & false
\end{array}
$$

$$
owner(c\texttt{<}\overline{d}\texttt{>}) \quad = \quad d_1
$$

$$
\begin{array}{lcl}
assumptions(\pi, c\texttt{<}\overline{d, d'}\texttt{>}) & = & \overline{con}[\overline{d}, \overline{d'}/\overline{p}, \overline{p'}], assumptions(\pi, c'\texttt{<}\overline{d}\texttt{>}) \\
assumptions(\pi, \texttt{Object}\texttt{<}d_1\texttt{>}) & = & \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
\underline{links(\pi, c\texttt{<}\overline{d, d'}\texttt{>})} & = & \overline{d' \to d''}[\texttt{that}/\texttt{this}][\overline{d}, \overline{d'}/\overline{p}, \overline{p'}], links(\pi, c'\texttt{<}\overline{d}\texttt{>}) \\
\quad \text{where}\ \overline{lnk} = \overline{\texttt{link}\ d'\ \texttt{->}\ d''} & & \\
links(\pi, \texttt{Object}\texttt{<}d_1\texttt{>}) & = & \emptyset
\end{array}
$$

$$
\begin{array}{lcl}
domains(\pi, c\texttt{<}\overline{d, d'}\texttt{>}) & = & \overline{\texttt{this}.p}, \underline{domains(\pi, c'\texttt{<}\overline{d}\texttt{>})} \\
& & \quad \text{where}\ \overline{dom} = \overline{[\texttt{public}]\ \texttt{domain}\ p}
\end{array}
$$

$$
\begin{array}{l}
\pi(c) = \texttt{class}\ c\texttt{<}\overline{p}, \overline{p'}\texttt{>}\ \texttt{extends}\ c'\texttt{<}\overline{p}\texttt{>}\ \texttt{assumes}\ \overline{con}\ \{\overline{dom}\ \overline{lnk}\ \overline{fd}\ constr\ \overline{md}\} \\
\qquad where\ \pi =\ \ \ldots \texttt{class}\ c\texttt{<}\overline{p}, \overline{p'}\texttt{>}\ \texttt{extends}\ c'\texttt{<}\overline{p}\texttt{>}\ \texttt{assumes}\ \overline{con}\ \{etc.\} \ldots
\end{array}
$$

Figure 4.13: ODE Lookup and Helper Functions

containment constraints) made by the type argument. The function collects the declarations from the class definition and from superclasses and applies the substitution from formal to actual parameter domains. Note that the `that`,`this` substitution is not needed as constraints can only be between parameter domains. Similarly the *links* function collects the explicit link statements made in the class and its super-classes. The function converts the declaration from the `link` d `->` d' form to $d \rightarrow d'$ for use in rules deducing domain permissions (see Section 4.6.2.2).

A list of all domains locally declared by a type is given by the function *domains*. Domain declarations are extracted from the class definition and adjusted from the form [`public`] `domain` p to `this`.p, the general form for non-parameter domains. No substitutions are required as the local domain names are not changed.

Note that in the runtime system (Section 4.6.3 onwards) we will reuse some of these functions but with runtime types for arguments rather than static types. Since the function definitions will look exactly the same we have chosen not to clutter the system with re-definition. In some cases (such as in the operational semantics rule B-CALL) this will result in types which are syntactically neither static nor runtime. This is not of concern since in these cases we are not concerned with types for example in B-CALL *methods* is used only to find the method body to execute.

4.6.2.2 Permission and Containment Rules

In Figure 4.14 we give rules for deducing permissions between objects/domains. The relation has two forms. The first form $\pi, \gamma \vdash d \rightarrow d'$ meaning that domain d has permission to access domain d'. The second form $\pi, \gamma \vdash path \rightarrow d$ states that the object referred to by *path* has permission to access d. This judgement is used in type rules to check that object accesses made by expressions are legal and also to check that type instantiations (in `new` expressions) are valid.

The rules generate the relation based on the assumptions and explicit permissions granted in the program, as well as general permission rules. We extend the rules from [AC04] to cope with the admission of final paths as roots for domain expressions.

Rule LINK-ENV exploits explicit permissions granted in any object reachable by a path. If a final path is typeable, then any explicit linking of domains ($d \rightarrow d'$) in that object (found by using the *links* lookup function on the type of the path) gives a valid permission. Considering Figure 4.11 we see that $\pi, \gamma \vdash$ `this`.f.`u` $\rightarrow$ `this.a` since `this`.f has type `final A<this.a, this.b>` and from *links* applied to this type we get `that.u` $\rightarrow$ `that.a` which, with the subsitution of `that` for `this`.f gives `this`.f.`u` $\rightarrow$ `this.a`. This example highlights the importance of the `that`,`this` subsitution. In the result of the call to *links* `this` refers to the current receiver as given in γ. `that` refers to the object being inspected i.e. `this`.f. Without the `that`,`this` substitution performed in *links* the result of the function call would include `this.u` $\rightarrow$ `this.a`. Any attempt then to replace `this` in the LHS with `this`.f would result in `this`.f.`u` $\rightarrow$ `this`.f.`a` which is clearly false as objects of class `B` do not declare a domain `a`.

$$\frac{t = \gamma(\texttt{this}) \qquad d \mathbin{\$} d' \in \mathit{assumptions}(\pi, t)}{\gamma, \pi \vdash d \mathbin{\$} d'}\ \textsc{dd-assume}$$

$$\frac{\pi, \gamma \vdash \mathit{path} : \texttt{final}\ t \qquad \pi, \gamma \vdash d \rightarrow \mathit{owner}(t) \qquad \mathit{public}(\pi, t, p)}{\pi, \gamma \vdash d \rightarrow \mathit{path}.p}\ \textsc{link-dpub}$$

$$\frac{\pi, \gamma \vdash \mathit{path} : t \qquad \texttt{this}.p \in \mathit{domains}(\pi, t)}{\pi, \gamma \vdash \mathit{path} \rightarrow \mathit{path}.p}\ \textsc{link-local}$$

$$\frac{\pi, \gamma \vdash \mathit{path}' : \texttt{final}\ t \qquad \pi, \gamma \vdash \mathit{path} \rightarrow \mathit{owner}(t) \qquad \mathit{public}(\pi, t, p)}{\pi, \gamma \vdash \mathit{path} \rightarrow \mathit{path}'.p}\ \textsc{link-ppub}$$

$$\frac{\pi, \gamma \vdash \mathit{path} : \texttt{final}\ t \qquad d \rightarrow d' \in \mathit{links}(\pi, t)[\mathit{path}/\texttt{that}]}{\pi, \gamma \vdash d \rightarrow d'}\ \textsc{link-env}$$

$$\frac{\pi, \gamma \vdash \mathit{path} : t \qquad \pi, \gamma \vdash \mathit{owner}(t) \rightarrow d}{\pi, \gamma \vdash \mathit{path} \rightarrow d}\ \textsc{link-owner}$$

Figure 4.14: Rules for static judgement of object/domain permissions

In LINK-DPUB a domain (d) is allowed to access any public domain (p) of an object it has permission to access. The object must be referred to via a final path (*path*) and permission to access *path* is checked by finding permission to access the owner of *path* (found using the *owner* lookup function on the type of *path*). This rule captures the nature of `public` domains, that is, that they are available to anyone who can access the owner, and to access the owner (*path*) we need only have permission for the owner of *path*. Figure 4.11 gives $\pi, \gamma \vdash \texttt{this}.f.\texttt{u} \rightarrow \texttt{this.gb}$. since$\pi, \gamma \vdash \texttt{this}.f.\texttt{u} \rightarrow \texttt{this.b}$ (by rule LINK-ENV) and `this.b` is the owner of `this.g`. LINK-LOCAL simply states that any object (given by the final path *path*) may access any domain it declares (since it owns it!). Existence of the domain is checked in the program with the *domains* function.

The rule LINK-OWNER captures delegation of permissions granted to the owner of an object to the object itself. This is the fundemental rule for checking an object's permission to access a domain (since links are granted between domains rather than individual objects and domains). In the example of Figure 4.11 we find, for example that, $\texttt{this}.f \rightarrow \texttt{this.b}$ since `a` (the owner of `f`) is explicitly granted permission to access `b`. Similarly we have $\texttt{this.g} \rightarrow \texttt{c}$ and $\texttt{this.g} \rightarrow \texttt{d}$.

LINK-PPUB is similar to LINK-DPUP but with a path on the left hand side rather than a domain. If *path* has permission to access the owner of path' then it has permission to access any public domain of path'. Note that the inductive appeal to the relation will ensure that *path* is final. In Figure 4.11 we get $\pi, \gamma \vdash \texttt{this.f.h} \rightarrow \texttt{this.g.b}$ as `this.f.h` has permission to access `this.b`.

The final rule DD-ASSUME belongs not only with the LINK rules but also in the containment rules that follow. DD-ASSUME uses the *assumptions* lookup function to find constraints between domains that are declared in the `assumes` clause of the class of the receiver. These may include $<$, $<+$, $<*$ and `sibling` constraints as well as link constraints, hence we have \$ rather than `link`. The rule could read the assumptions of any type in γ but as will be seen in Section 4.6.2.4 all the assumptions of these types must be deducable by other rules. In the context of Figure 4.1 $\mathit{assumptions}(\pi, \texttt{A<d1,d2>}) = \texttt{d1} \rightarrow \texttt{d2}$ and

$$\frac{\gamma,\pi \vdash \mathit{path} : \texttt{final}\ t \quad \mathit{owner}(t) = d}{\gamma,\pi \vdash \mathit{path} < d}\ \text{PD-DIRECT} \qquad \frac{\pi,\gamma \vdash \mathit{path} : \texttt{final}\ t \quad \texttt{this}.p \in \mathit{domains}(\pi, t)}{\gamma,\pi \vdash \mathit{path}.p < \mathit{path}}\ \text{DP-DIRECT}$$

$$\frac{\pi,\gamma \vdash \mathit{path} : \texttt{final}\ t \quad \texttt{this}.p, \texttt{this}.p' \in \mathit{domains}(\pi, t)}{\gamma,\pi \vdash \mathit{path}.p\ \texttt{sibling}\ \mathit{path}.p'}\ \text{DD-SIB} \qquad \frac{\pi,\gamma \vdash d}{\pi,\gamma \vdash d <\!*\ \texttt{world}}\ \text{DD-WORLD}$$

$$\frac{\gamma,\pi \vdash ?\ \$\ ?'' \quad \gamma,\pi \vdash ?''\ \$'\ ?'}{\gamma,\pi \vdash ?\ \$ \sqcap \$'\ ?'}\ \text{COMPOSE} \qquad \frac{\pi,\gamma \vdash d}{\pi,\gamma \vdash d\ \texttt{sibling}\ d \quad \pi,\gamma \vdash d <\!*\ d}\ \text{DD-REF}$$

$\sqcap$	$<$	$<+$	$<*$	`sibling`
$<$	$<+$	$<+$	$<+$	$-$
$<+$	$<+$	$<+$	$<+$	$-$
$<*$	$<+$	$<+$	$<*$	$-$
`sibling`	$<$	$<+$	$-$	`sibling`

Figure 4.15: Static rules for object/domain containment

so if $\gamma =$[`this` $\mapsto$ `A<d1,d2>`$,\ldots$] then DD-ASSUME would give $\pi,\gamma \vdash d1 \rightarrow d2$.

Figure 4.15 gives rules for statically deducing structural relationships between objects/domains. These rules are used to check the validity of type instantiations (in `new` expressions) and in the sub-effect relation (Section 4.6.2.8). Structural relationships come in two flavours, containment and siblings.

We define containment such that we find that an object is below the domain that owns it, which is in turn below the object that owns it. The first object is thus (transitively) below the second object. The containment relation exists between pairs of paths (objects), pairs of domains, or a path and a domain.

The sibling relationship is only between pairs of domains. It captures that two domains have the same owner (an object).

Our rules follow in part from those of `Joe` but are adapted to deal with domains as well as objects. Rules regarding the the sibling relationship are novel. The relation has the form $\pi,\gamma \vdash a\ \$\ a'$ where a may be either a path or domain expression and \$ is a constraint as in the language syntax. Some rules appeal to the typing judgement for paths which can be found in Figure 4.20.

The first rule, PD-DIRECT states that an object (*path*) is contained with in its owner domain (d) e.g. $\pi,\gamma \vdash$ `this` $<$ `o` or $\pi,\gamma \vdash$ `this.`f $<$ `this.a` in Figure 4.11. As in the link rules, *path* must be final for soundness. The second rule, DP-DIRECT, states the converse, that a domain (*path.p*) is contained within its owning object (*path*). We check existence of the domain p with a call to the *domains* function. In Figure 4.11 we find (among others) the following $\pi,\gamma \vdash$ `this` $<$ `o`, $\pi,\gamma \vdash$ `this.g` $<$ `this.b` and $\pi,\gamma \vdash$ `this.`f`.h` $<$ `this.`f`.u`.

DD-WORLD simply captures that any domain is under (reflexively and transitively) the top-level domain `world`.

The rules DD-SIB and DD-REF handle the `sibling` constraint and reflexivity of $<*$. In DD-SIB we deduce that any two domains with the same parent (the object denoted by the final *path*) are siblings. Thus, in Figure 4.11, as we expect should expect we get $\pi, \gamma \vdash$ `this.g.a sibling this.g.b`. DD-REF gives reflexivity for both `sibling` and $<*$ as one expects.

The rule DD-ASSUME (as shown in Figure 4.14) populates the containment and sibling relations with constraints given in the assumptions of the current context. The final rule, COMPOSE, combines two containment relations giving the (reflexive) transitive closure of the containment relation (with the $\sqcap$ operator defined in the accompanying table). As well as combining direct, transitive and reflexive containment we can also combine with `sibling`. `sibling` is transitive but also, since two siblings have the same owner combines (to the right) with direct and transitive containment. It does not combine with reflexive transitive containment as the resulting relation could be `sibling` or containment. `sibling` does not combine to the left except with itself. We also have subsumption of the relations such that $<*$ subsumes $<+$ and $<$ and $<+$ subsumes $<$, we omit rules for these.

4.6.2.3 Well-formed Domains and Disjoint Domains

In Figure 4.16 we give rules for checking that domains are well-formed i.e. valid in the given context. We also give rules for deducing disjointness (innequality) of domains, which will be used in deduction of disjoint types in the following section.

Domains are valid ($\pi, t \vdash d$) when they are parameter domains of `this` (D-THIS), declared domains of any object described by a typable final path (D-PATH) or `world`. Domains can be deduced disjoint ($\pi, \gamma \vdash d \# d'$) in two ways. D-DISCONT says that domains are disjoint when one is strictly contained in the other. D-DISSIB says that for two non-equal sibling domains, any domain strictly contained in one is disjoint from the other. Notice that, since we have DD-REF, this also covers disjointness of non-equal siblings.

4.6.2.4 Well-formed Types and Type Environments, Subtypes and Disjoint Types

In Figure 4.17 we give rules for judging well-formedness of types and the subtype relation.

The rules for subtyping are standard and follow those from FDJ. Subtyping is reflexive (SUBT-REF) and transitive (SUBT-TRANS) with a basic rule (TYPE-DEC) populating the relation. In TYPE-DEC each instance type of a class is a subtype of the superclass instantiated with appropriate parameter domains. For convenience we will reuse the definitions of subtyping in the runtime section where static domains will be replaced with runtime domains.

$$\frac{\gamma(\texttt{this}) = c\texttt{<}\overline{p}\texttt{>}}{\pi, \gamma \vdash p_i}\ \text{D-THIS} \qquad \frac{\begin{array}{c}\pi, \gamma \vdash path : \texttt{final}\ t \\ p \in domains(\pi, t)\end{array}}{\pi, \gamma \vdash path.p}\ \text{D-PATH}$$

$$\frac{}{\pi, \gamma \vdash \texttt{world}}\ \text{D-WORLD}$$

$$\frac{\pi, \gamma \vdash d \texttt{ <+ } d'}{\pi, \gamma \vdash d \# d'}\ \text{D-DISCONT} \qquad \frac{\begin{array}{c}\pi, \gamma \vdash d\ \texttt{sibling}\ d'' \\ \pi, \gamma \vdash d \# d'' \\ \pi, \gamma \vdash d' \texttt{ <+ } d''\end{array}}{\pi, \gamma \vdash d \# d'}\ \text{D-DISSIB}$$

Figure 4.16: Well-formed and Disjoint Domains

$$\frac{}{\pi \vdash t \leq t}\ \text{TYPE-REF} \qquad \frac{\begin{array}{c}\pi \vdash t \leq t'' \\ \pi \vdash t'' \leq t'\end{array}}{\pi \vdash t \leq t'}\ \text{TYPE-TRANS}$$

$$\frac{\begin{array}{c}\pi(c) = \texttt{class}\ c\texttt{<}\overline{p}, \overline{p'}\texttt{>}\ \texttt{extends}\ c'\texttt{<}\overline{p}\texttt{>}\ \ldots \\ |\overline{p}| = |\overline{d}| \quad |\overline{p'}| = |\overline{d'}|\end{array}}{\pi \vdash c\texttt{<}\overline{d}, \overline{d'}\texttt{>} \leq c'\texttt{<}\overline{d}\texttt{>}}\ \text{TYPE-DEC}$$

$$\frac{\begin{array}{c}\pi(c) = \texttt{class}\ c\texttt{<}\overline{p}\texttt{>}\ \ldots \\ \mid \{\overline{p}\} \mid = \mid \{\overline{d}\} \mid \\ \pi, \gamma \vdash assumptions(\pi, c\texttt{<}\overline{d}\texttt{>})\end{array}}{\pi, \gamma \vdash c\texttt{<}\overline{d}\texttt{>}}\ \text{WF-TYPE} \qquad \frac{\pi, \gamma \vdash d}{\pi, \gamma \vdash \texttt{Object<}d\texttt{>}}\ \text{WF-TYPEOBJ}$$

Figure 4.17: Rules for well-formed types and subtypes

$$\frac{\pi(c) = \texttt{class}\ c\texttt{<}\overline{p}\texttt{>}\ \ldots}{\pi \vdash \texttt{this} \mapsto c\texttt{<}\overline{p}\texttt{>}}\ \text{TENV-THIS} \qquad \frac{\pi \vdash \gamma \quad \pi, \gamma \vdash t \quad x \notin \mathit{domain}(\gamma)}{\pi \vdash \gamma[x \mapsto t]}\ \text{TENV-VAR}$$

Figure 4.18: Well Formed Type Environments

$$\frac{\nexists\ \overline{p}, \overline{p'}.\ \pi \vdash c\texttt{<}\overline{p}, \overline{p'}\texttt{>} \leq c'\texttt{<}\overline{p}\texttt{>}\ \vee\ \pi \vdash c'\texttt{<}\overline{p}, \overline{p'}\texttt{>} \leq c\texttt{<}\overline{p}\texttt{>}}{\pi, \gamma \vdash c\texttt{<}\overline{d}\texttt{>}\#c'\texttt{<}\overline{d'}\texttt{>}}\ \text{TYPE-DISCLASS}$$

$$\frac{\pi, \gamma \vdash d_i \# d'_i}{\pi, \gamma \vdash c\texttt{<}\overline{d}\texttt{>}\#c'\texttt{<}\overline{d'}\texttt{>}}\ \text{TYPE-DISPARAM}$$

Figure 4.19: Disjoint Types

Well-formed types, as defined in WF-TYPE, must be of a class defined in the program, have the correct number of parameter domains and all the constraints on the parameter domains of the class must be satisfied in γ. Thus in Figure 4.11 `A<this.a, this.b>` is valid but `A<this.a, c>` is not, as `this.a` and `c` cannot be shown to be siblings[4]. We also allow `Object` as a type, taking one parameter which must be a valid domain.

A type environment is only valid for a given program ($\pi \vdash \gamma$) if the program is well-formed (see Section 4.6.2.10), the types it contains are valid in the program and if the type evironment is internally consistent. The two rules in Figure 4.18 capture these requirements. TENV-THIS validates the basic type environment which only gives a type for the receiver, `this`. All type environments must include `this`, as it provides the basic context for typing. The type of `this` must be a class which exists in the program with parameters as given in the program. The parameters must be as declared since there is no additional context from which to obtain domain names, they form part of our basic assumptions. The second rule, TENV-VAR covers the addition of further variable bindings to the environment. For a type environment $\gamma\ [x \mapsto t]$ the sub-environment γ must be valid, the variable x may not be in the domain of γ (as local variables are all final) and the type of x must be valid in γ. The type of x may thus refer to other variables already in the type evironment (with domain parameters such as `this.`p, for example) but, through our restrictions, we avoid cyclical references, which would not be sound.

Figure 4.19 gives rules for judging disjoint types where $\pi, \gamma \vdash t\#t'$ means t and t' are disjoint. When two types are disjoint they cannot possibly refer to the same value. This will be used when we judge disjoint effects in Section 4.6.2.7. These rules follow directly from those given in `Joe`.

Types are disjoint when there is no subtype relationship between them, but we must be cautious. We cannot simply rely on the subtype judgement as described above as it is conservative. The parameters of the domains may be aliases without our knowledge and thus simple name matching (as in the subtype judgement) will not suffice. Instead, we need to know either that the two types are not in the subclass relationship (of which the

[4] Nor can they be, since `c` is constrained to be above `o`.

program gives total knowledge) or that two domains that should be equal are not.

Rule TYPE-DISCLASS judges two types disjoint if neither is a superclass of the other. We can make use of the subclass judgement here as we are not trying to compare the actual type instances. By picking arbitrary domain parameters we can check whether it is at all possible for the types to be subtypes. If it is not, they must be disjoint.

Rule TYPE-DISPARAM makes use of the naming convention of ODE that superclasses are instantiated with the first n parameters of the subclass. Thus, if either type can be seen to disagree in one of the first n parameters they cannot possibly be subtypes. The rules of the form $\pi, \gamma \vdash d \# d'$ capture that domains are disjoint and are a subset of the rules for disjoint effects in Section 4.6.2.7 namely DIS-NEQ and DIS-CONT.

4.6.2.5 Typing Rules

Typing for ODE is a mixture of that seen in `Joe` and FDJ and the general structure will be familiar to those who have encountered other Java-like systems. Typing rules are found in Figure 4.20. As a matter of personal taste we omit a specific subsumption rule and instead include sub-type checks in B-ASS, B-CALL and B-NEW.

Typing of variables is simply by lookup in the type environment as in TYPE-VAR. Field lookup is typed, in rule TYPE-FLD, using the field lookup function $fields$ on the type of the target object. The type corresponding to the desired field is selected from the list of candidates. Note another manifestation of the `that` keyword. The reason is just the same as described in Section 4.6.2.2. Field assignment (TYPE-ASS) checks the type of the right hand side is a subtype of the expected type for the field (validity of the field is checked in the typing of $z.f$). By insisting the type of $z.f$ is of the form $c{<}\overline{d}{>}$ we ensure the field is not final and may be assigned to (similarly it prevents assignment to a final field).

In the case of object creation (TYPE-NEW) the type is simply the type given in the `new` expression. We require that the type is valid in the context of π and γ and that all parameters have appropriate types with respect to the declaration in the class's constructor.

Method call is typed in rule TYPE-CALL. The bulk of the work is done with a call to the look up function $methods$, which gives the expected types for parameters and the return type. We perform the substitution $[path, \overline{z}/\texttt{that}, \overline{x^{\dagger}}]$ to convert references to the formal receiver and parameters of the method call (in types and effects) to the actuals. The use of `that` avoids clashes as before and, in addition, the reserved variable names $\overline{x^{\dagger}}$ are used to avoid clashes between the names of the formal parameters in the same way. For compatibility, we check that the types of the actual parameters are subtypes of the declared types. The overall type is that declared in the method definition (up to the subsitutions).

There are two typing rules for `let` expressions. TYPE-LET is a more general form applicable to all `let` expressions and TYPE-LETFINAL is only applicable when the

$$\frac{}{\pi, \gamma \vdash z : \texttt{final}\ \gamma(z)}\ \text{TYPE-VAR}$$

$$\frac{\begin{array}{c}\pi, \gamma \vdash z : t \\ \mathit{fields}(\pi, t)[z/\texttt{that}] \ = \ \overline{t\ f}\end{array}}{\pi, \gamma \vdash z.f_i : t_i}\ \text{TYPE-FLD}$$

$$\frac{\begin{array}{c}\pi, \gamma \vdash z.f : c\texttt{<}\overline{d}\texttt{>} \\ \pi, \gamma \vdash z' : t \\ \pi \vdash t \leq c\texttt{<}\overline{d}\texttt{>}\end{array}}{\pi, \gamma \vdash z.f{=}z' : t}\ \text{TYPE-ASS}$$

$$\frac{\begin{array}{c}\mathit{cons}(\pi, t) = t(\overline{t}\ \overline{x})\{\texttt{this}.\overline{f} = \overline{\mathit{cexp}}\} \\ \pi, \gamma \vdash \overline{z} : \overline{t'} \\ \pi \vdash \overline{t'} \leq \overline{t} \quad \pi, \gamma \vdash t\end{array}}{\pi, \gamma \vdash \texttt{new}\ t(\overline{z}) : t}\ \text{TYPE-NEW}$$

$$\frac{\begin{array}{c}\pi, \gamma \vdash b : t' \\ x \notin \mathit{domain}(\gamma) \\ \gamma' = \gamma[x \mapsto t'] \\ \pi, \gamma' \vdash e : t'' \\ \pi \vdash t'' \leq t \\ \pi, \gamma \vdash t\end{array}}{\pi, \gamma \vdash \texttt{let}\ x = b\ \texttt{in}\ e : t}\ \text{TYPE-LET}$$

$$\frac{\begin{array}{c}\pi, \gamma \vdash b : \texttt{final}\ c\texttt{<}\overline{d}\texttt{>} \\ x \notin \mathit{domain}(\gamma) \\ \gamma' = \gamma[x \mapsto c\texttt{<}\overline{d}\texttt{>}] \\ \pi, \gamma' \vdash e : t' \\ t = t'[b/x]\end{array}}{\pi, \gamma \vdash \texttt{let}\ x = b\ \texttt{in}\ e : t}\ \text{TYPE-FINAL}$$

$$\frac{\begin{array}{c}\pi, \gamma \vdash z : t' \\ \pi, \gamma \vdash \overline{z} : \overline{t'} \\ \mathit{methods}(\pi, t', m) = t_r\ m(\overline{t\ x})\ \psi\ \{e\} \\ \pi \vdash \overline{t'} \leq \overline{t}[z, \overline{z}/\texttt{that}, \overline{x^{\dagger}}]\end{array}}{\pi, \gamma \vdash z.m(\overline{z}) : t_r[z, \overline{z'}/\texttt{that}, \overline{x^{\dagger}}]}\ \text{TYPE-CALL}$$

$$\frac{\begin{array}{c}\pi, \gamma \vdash \mathit{path} : \texttt{final}\ c\texttt{<}\overline{d}\texttt{>} \\ \mathit{fields}(\pi, c\texttt{<}\overline{d}\texttt{>}) = \overline{t\ f} \\ t_i[\mathit{path}/\texttt{that}] = \texttt{final}\ c'\texttt{<}\overline{d'}\texttt{>}\end{array}}{\pi, \gamma \vdash \mathit{path}.f_i : \texttt{final}\ c'\texttt{<}\overline{d'}\texttt{>}}\ \text{TYPE-PATH}$$

Figure 4.20: Typing rules of ODE

sub-expression b is final, that is, a look-up on a field marked final. When b is final it may be possible to get a more accurate type, we will show this with an example.

In both rules the subexpression b is typed and the type environment, γ is augmented with the type of b bound to x. The second subexpression e is typed with the new type environment. The type of e (t'') may include domain expressions rooted at x. Such types are not valid in the scope of γ and as such are not valid return types for the `let` expression. When b is final, x can be replaced in t'' with b to make a type valid in γ. Otherwise, as in TYPE-LET the type can be subsumed to a type valid in γ, i.e. which does not mention x.

Consider a class `C<owner, p> extends Object<owner>` ... with (for simplicity) no constraints between `owner` and `p`. Now consider the following expression in the context of the type environment in Figure 4.11:

`let x = this.f in let y = new C<o, x.u>() in y`

Because `this.f` is final the type of the expression is `C<o, this.f.u>` whereas if `this.f` was not final we would have to subsume the type of `y` and end up with the less accurate `Object<o>`. In more extreme cases TYPE-LETFINAL may be able to type expressions that TYPE-LET cannot, for example types like *Object*<`this.f.a`> where the owner domain cannot be subsumed away.

The final rule, TYPE-PATH is used to type final paths which, although not executable expressions, can occur in domain expressions and effects and need to be typed (see Section 4.6.2.2 for examples). Since we only apply path typing in contexts where paths must be final we restrict the rule to only type final paths. The type of the path is determined similarly to TYPE-FLD with a call to the $fields$ function. Additionally the sub-path *path* must have a final type (to ensure the entire path is immutable).

4.6.2.6 Well-Formed Effects

In this section we introduce the judgement for well-formedness of effects with respect to a program and type environment. The judgement is simple and consists mostly of checking validity of paths (that they are typable and final) and domains. The judgement will be used when calculating effects of expressions to check that subsumed effects are valid. Rules are given in Figure 4.21

4.6.2.7 Disjoint Effects

In Figure 4.22 we present rules for judging disjointness of effects. These rules correspond to the disjointness relation in Chapter 3 and, thus, can be used to deduce independence properties. Many of the rules have their roots in `Joe` but we have modified rules to cope with the differences between `Joe` and ODE, as well as adding new rules to cope with domains and effects including `sibling`. We design the rules so that, in any valid runtime configuration the projection of the effects (see Section 4.6.3.4) is disjoint.

Where we refer to Figure 4.11 consider the object instance diagram, which should clarify

$$\frac{\pi, \gamma \vdash \mathit{path} : \mathtt{final}\ t}{\begin{array}{c}\pi, \gamma \vdash \mathit{path} \\ \pi, \gamma \vdash \mathit{path}.\mathtt{all}\end{array}}\ \text{FX-PATH} \qquad \frac{\gamma(\mathtt{this}) = c\mathtt{<}\overline{p}\mathtt{>}}{\pi, \gamma \vdash \overline{p}}\ \text{FX-PARAM}$$

$$\frac{\begin{array}{c}\pi, \gamma \vdash \mathit{path} : \mathtt{final}\ t \\ p \in \mathit{domains}(\pi, t)\end{array}}{\pi, \gamma \vdash \mathit{path}.p}\ \text{FX-PATHDOM}$$

$$\frac{\pi, \gamma \vdash \theta}{\pi, \gamma \vdash \theta.\mathtt{under}}\ \text{FX-PATH} \qquad \frac{\pi, \gamma \vdash d}{\pi, \gamma \vdash d.\mathtt{siblings}}\ \text{FX-SIBLINGS}$$

$$\frac{\pi, \gamma \vdash \phi \quad \pi, \gamma \vdash \phi'}{\pi, \gamma \vdash \phi + \phi'}\ \text{FX-UNION} \qquad \frac{}{\pi, \gamma \vdash \mathtt{emp}}\ \text{FX-EMP}$$

Figure 4.21: Rules for well-formedness of effects

why effects are disjoint by reference to the objects they corrsepond to.

The simplest effects we have are paths, referring to only one object. If two paths have disjoint types (see Section 4.6.2.4) then the two paths are disjoint as effects (DIS-PTYPE). In Figure 4.11, `this.f` and `this.g` are disjoint effects since classes `A` and `B` are incompatible and thus the fields cannot possibly point to the same object. Additionally they are not even in the same domain and so, like objects 2 and 3, cannot be aliases.

In DIS-NEQ we capture that for any two domain expressions *path.p* and *path*$'$.*p*$'$, if the domains names (p, p') are not equal, the effects are disjoint. Even if the two path expressons refer to the same object, the domains must still be distinct as they have different names. Thus, even if `this.a` referred to the same object as `this`, `this.g.b` and `this.b` would still be disjoint. Similarly for any two disjoint paths i.e. that cannot possibly refer to the same object (by rule DIS-PTYPE), any domains they own must be disjoint, as they are owned by different objects (DIS-PATHS). So, following from the example for DIS-PATH, `this.f.u` and `this.g.a` are disjoint and would be even if both domains had the same name.

Using the containment relation, we can tell if a domain d is strictly beneath (<+) a root effect θ' (which can only be of the form *path* or d) . If this is the case then as effects d and θ are disjoint simply because the objects the effects refer to cannot be at the same level in the tree. We exploit this in rule DIS-CONT. We restrict the left hand side to d as to use θ would be unsound. In the containment relation two domains in the <+ relation must be distinct and so, as effects, they will always be disjoint. With θ' as a path, the effects will still be disjoint as the closest they could be is that θ' owns d but then they still refer to a disjoint set of objects. Were we to admit *path* on the left hand side and d on the right, the disjointness need not hold. d could be the owner of *path*, <+ would still hold but *path* is a sub-effect of d.

In DIS-OWNED we define a number of disjoint effects from the premise of two paths which have the same owner but are disjoint. Because the two objects are a) distinct and

$$\frac{}{\gamma, \pi \vdash \phi\#\texttt{emp}}\ \text{DIS-EMP} \qquad \frac{\gamma, \pi \vdash \phi\#\phi' \quad \gamma, \pi \vdash \phi\#\phi''}{\gamma, \pi \vdash \phi\#\phi'{+}\phi''}\ \text{DIS-UNION}$$

$$\frac{\gamma, \pi \vdash \phi'\#\phi}{\gamma, \pi \vdash \phi\#\phi'}\ \text{DIS-REF} \qquad \frac{\gamma, \pi \vdash \mathit{path} : t \quad \gamma, \pi \vdash \mathit{path}' : t' \quad \gamma, \pi \vdash t\#t'}{\gamma, \pi \vdash \mathit{path}\#\mathit{path}'}\ \text{DIS-PTYPE}$$

$$\frac{\pi, \gamma \vdash \mathit{path}\#\mathit{path}'}{\gamma, \pi \vdash \mathit{path}.p\#\mathit{path}'.p'}\ \text{DIS-PATHS} \qquad \frac{p \neq p'}{\gamma, \pi \vdash \mathit{path}.p\#\mathit{path}'.p'}\ \text{DIS-NEQ}$$

$$\frac{\gamma, \pi \vdash \phi\#\phi'' \quad \gamma, \pi \vdash \phi' \sqsubseteq \phi''}{\gamma, \pi \vdash \phi\#\phi'}\ \text{DIS-SUB} \qquad \frac{\gamma, \pi \vdash d <+ \theta'}{\gamma, \pi \vdash d\#\theta'}\ \text{DIS-CONT}$$

$$\frac{\begin{array}{c}\gamma, \pi \vdash \mathit{path} : t \\ \gamma, \pi \vdash \mathit{path}' : t' \\ \mathit{owner}(t) = \mathit{owner}(t') \\ \gamma, \pi \vdash \mathit{path}\#\mathit{path}'\end{array}}{\begin{array}{c}\gamma, \pi \vdash \mathit{path}.\texttt{under}\#\mathit{path}'.\texttt{under} \\ \gamma, \pi \vdash \mathit{path}.p.\texttt{under}\#\mathit{path}'.p'.\texttt{under} \\ \gamma, \pi \vdash \mathit{path}.\texttt{all}.\texttt{under}\#\mathit{path}'.\texttt{all}.\texttt{under}\end{array}}\ \text{DIS-OWNED}$$

Figure 4.22: Rules for judging disjointness of effects

b) at the same level in the tree we know there can be no overlap between the `under` effects. Supposing that `this.f` and `this.f` had the same owner in Figure 4.11 they would still be disjoint (due to their class). Then, any of the under effects described in DIS-OWNED rooted with these paths would be disjoint. Simply consider the boxes representing the object's local domains, all these effects are bounded by the boxes which are parallel.

The remaining rules are based on the sub-effect relation and simple set theory. The empty effect is disjoint from all others (including itself) (DIS-EMP). If two effects are disjoint from a third their union is also disjoint from the third (DIS-UNION). Disjointness of effects is commutative (DIS-REF). If two effects are disjoint any of their sub-effects are disjoint (DIS-SUB).

4.6.2.8 Sub Effects

The rules for judging sub-effects (that one effect is contained within another) are given in Figure 4.23. We give rules in two formats: i) $\pi, \gamma \vdash \phi \sqsubseteq \phi'$ states that one basic effect is contained within another whereas ii) $\pi, \gamma \vdash \psi \sqsubseteq \psi'$ compares full effects (by comparing the read and write components).

In SUB-PATHDOM we see that *path* (referring to a single object) is contained within its owner domain. A domain picked by name from an object i.e. *path.p* (meaning all the objects owned by that domain) is contained within the collection of all domains owned

$$\frac{}{\gamma, \pi \vdash \mathit{path}.p \sqsubseteq \mathit{path}.\texttt{all}}\ \text{SUB-LCLALL}$$

$$\frac{\gamma, \pi \vdash \mathit{path} : t \quad \mathit{owner}(t) = d}{\gamma, \pi \vdash \mathit{path} \sqsubseteq d}\ \text{SUB-PATHDOM}$$

$$\frac{\gamma, \pi \vdash \phi \sqsubseteq \phi'' \quad \gamma, \pi \vdash \phi' \sqsubseteq \phi''}{\gamma, \pi \vdash \phi{+}\phi' \sqsubseteq \phi''}\ \text{SUB-UNIONL}$$

$$\frac{\gamma, \pi \vdash \phi \sqsubseteq \phi'' \quad \gamma, \pi \vdash \phi'' \sqsubseteq \phi'}{\gamma, \pi \vdash \phi \sqsubseteq \phi'}\ \text{SUB-TRANS}$$

$$\frac{}{\gamma, \pi \vdash \phi \sqsubseteq \phi}\ \text{SUB-REF}$$

$$\frac{\gamma, \pi \vdash \phi \sqsubseteq \phi'}{\gamma, \pi \vdash \phi \sqsubseteq \phi'{+}\phi''}\ \text{SUB-UNIONR}$$

$$\frac{}{\gamma, \pi \vdash \texttt{emp} \sqsubseteq \phi}\ \text{SUB-EMP}$$

$$\frac{}{\gamma, \pi \vdash \theta \sqsubseteq \theta.\texttt{under}}\ \text{SUB-UNDER}$$

$$\frac{}{\pi, \gamma \vdash d \sqsubseteq d.\texttt{siblings}}\ \text{SUB-DSIB}$$

$$\frac{\pi, \gamma \vdash d\ \texttt{sibling}\ d'}{\pi, \gamma \vdash d' \sqsubseteq d.\texttt{siblings}}\ \text{SUB-SIBDSIB}$$

$$\frac{\gamma, \pi \vdash \theta <\!*\ \theta'}{\gamma, \pi \vdash \theta.\texttt{under} \sqsubseteq \theta'.\texttt{under}}\ \text{SUB-CONUND}$$

$$\frac{}{\gamma, \pi \vdash \mathit{path}.p.\texttt{siblings} \sqsubseteq \mathit{path}.\texttt{all}}\ \text{SUB-SIBALL}$$

$$\frac{\gamma, \pi \vdash d <\!+\ \theta'}{\gamma, \pi \vdash d.\texttt{siblings} \sqsubseteq \theta'.\texttt{under}}\ \text{SUB-SIBDUNDER}$$

$$\frac{}{\gamma, \pi \vdash \mathit{path}.\texttt{all} \sqsubseteq \mathit{path}.\texttt{under}}\ \text{SUB-ALLUNDER}$$

$$\frac{\gamma, \pi \vdash \phi_1 \sqsubseteq \phi_2 \quad \pi, \gamma \vdash \phi_3 \sqsubseteq \phi_4}{\gamma, \pi \vdash \texttt{rd}\ \phi_1\ \texttt{wr}\ \phi_3 \sqsubseteq \texttt{rd}\ \phi_2\ \texttt{wr}\ \phi_4}\ \text{SUB-EFFECT}$$

Figure 4.23: Sub-effect Rules

by that object, *path*.`all`. SUB-LCALL applies this fact. So, we find by these rules that, in Figure 4.11, `this.f` $\sqsubseteq$ `this.a` $\sqsubseteq$ `this.all`

Rule SUB-SIBDUNDER gives a general subsumption for effects involving siblings. The effect d.`siblings` is a sub-effect of d'.`under` for any d' which is above d in the ownership hierarchy. This is quite crude but caters for a general case where we do not know anything about d. A more specific and finer grained subsumption is given in SUB-SIBALL. In the case that we know precisely the domain which is the target of the `siblings` construction we can subsume to *path*.`all` where *path* is the owner of the target domain. This is, in fact an equality. Further more, *path*.`all` is contained in *path*.`under` (SUB-ALLUNDER).

The rules SUB-DSIB and SUB-DSIBSIB are two rules which allow other effects to be subsumed by effects involving `siblings`. In SUB-DSIB we see that d is a subeffect of d.`siblings` since d.`siblings` contains d and all sibling domains. Similarly SUB-SIBDSIB subsumes any sibling domain of d into d.`siblings`.

As one would expect, the sub-effect relationship is reflexive (SUB-REF) and transitive (SUB-TRANS). SUB-UNIONL and SUB-UNIONR are both reflections of basic properties of sets. In SUB-UNIONL if two effects are contained within a third, their union must also be and in SUB-UNIONR if an effect is contained within a second it is contained within the second plus any other effect. The empty shape is contained within any other, as in SUB-EMP.

SUB-EFFECT defines the relation for full read/write effects by simply checking the components are sub-effects.

4.6.2.9 Effects Rules

Rules for judging the effect of an expression are given in Figure 4.24. Again, these follow from `Joe` with modifications. Most of the 'hard work' is done by the appeals to the sub-effect rules. These allow more precise effects, only valid within certain scope, to be subsumed to a wider effect that is valid in the wider context.

Variable use (FX-VAR), null (FX-NULL) and object creation (FX-NEW) are all effectless. In the first two cases this should be obvious as no part of the heap is used. Effects track heap usage while variable access only reads the stack. Thus, variable access will succeed even with an empty heap so long as the stack contains the variable. Such a state would not be well-formed but this is not a concern for this judgement. The case of object creation is a little more involved. In ODE we restrict the syntax of constructors to avoid general computation. We allow (in fact, require) assignment to the fields of the object with either new objects or the arguments of the constructor call. As such, no existing objects in the heap are modified and we are able to give `new` expressions empty effects. Thus, an expression that includes a `new` has a smaller effect (describing fewer objects). This will allow us to deduce more non-interference relations with the disjoint effects rules. A constructor which allowed general computation would require non-trivial effects and the case would become similar to that of method call (with an effect annotation on each

$$\frac{}{\gamma, \pi \vdash z : \texttt{rd emp wr emp}}\ \text{FX-VAR} \qquad \frac{}{\gamma, \pi \vdash \texttt{null} : \texttt{rd emp wr emp}}\ \text{FX-NULL}$$

$$\frac{z \in \mathit{domain}(\gamma)}{\gamma, \pi \vdash z.f : \texttt{rd}\ z\ \texttt{wr emp}}\ \text{FX-FLD} \qquad \frac{z \in \mathit{domain}(\gamma)}{\gamma, \pi \vdash z.f{=}z' : \texttt{rd}\ z\ \texttt{wr}\ z}\ \text{FX-ASS}$$

$$\frac{\begin{array}{c}\gamma, \pi \vdash z : t \\ \mathit{methods}(\pi, t, m) = t_r\ m(\overline{t\ x})\ \psi\ \{e\} \\ \texttt{rd}\ \phi\ \texttt{wr}\ \phi' = \psi[z, \overline{z}/\texttt{that}, \overline{x^{\dagger}}]\end{array}}{\gamma, \pi \vdash z.m(\overline{z'}) : \texttt{rd}\ z{+}\phi\ \texttt{wr}\ \phi'}\ \text{FX-CALL}$$

$$\frac{}{\gamma, \pi \vdash \texttt{new}\ c\texttt{<}\overline{d}\texttt{>}(\overline{z}) : \texttt{rd emp wr emp}}\ \text{FX-NEW}$$

$$\frac{\begin{array}{c}\gamma, \pi \vdash b : \texttt{rd}\ \phi_1\ \texttt{wr}\ \phi_2 \\ \gamma, \pi \vdash b : t \\ \gamma[x \mapsto t], \pi \vdash e : \texttt{rd}\ \phi_3\ \texttt{wr}\ \phi_4 \\ \gamma[x \mapsto t], \pi \vdash e : t' \\ \gamma[x \mapsto t], \pi \vdash \texttt{rd}\ \phi_1{+}\phi_3\ \texttt{wr}\ \phi_2{+}\phi_4 \sqsubseteq \texttt{rd}\ \phi\ \texttt{wr}\ \phi' \\ \gamma, \pi \vdash \texttt{rd}\ \phi\ \texttt{wr}\ \phi' \\ \pi \vdash t' \leq t'' \qquad \pi, \gamma \vdash t''\end{array}}{\gamma, \pi \vdash \texttt{let}\ x = b\ \texttt{in}\ e : \texttt{rd}\ \phi\ \texttt{wr}\ \phi'}\ \text{FX-LET}$$

$$\frac{\begin{array}{c}\gamma, \pi \vdash b : \texttt{rd}\ \phi_1\ \texttt{wr}\ \phi_2 \\ \gamma, \pi \vdash b : [\texttt{final}]\ c\texttt{<}\overline{d}\texttt{>} \\ \gamma[x \mapsto c\texttt{<}\overline{d}\texttt{>}], \pi \vdash e : \texttt{rd}\ \phi_3\ \texttt{wr}\ \phi_4 \\ \texttt{rd}\ \phi\ \texttt{wr}\ \phi' = \texttt{rd}\ \phi_1{+}\phi_3[b/x]\ \texttt{wr}\ \phi_2{+}\phi_4[b/x]\end{array}}{\gamma, \pi \vdash \texttt{let}\ x = b\ \texttt{in}\ e : \texttt{rd}\ \phi\ \texttt{wr}\ \phi'}\ \text{FX-LETFINAL}$$

Figure 4.24: Judgement of effects for expressions

constructor).

For field access (FX-FIELD) the effect is `rd` z `wr emp` where z is the object whose field is being accessed. At first glance it might appear the effect should be `rd` $z.f$ `wr emp` or similar, but this would be a mistake. The object the field points at is not required to calculate the return value of the expression, only the object containing the field. Similarly in the field assignment case (FX-FLDASS) the effect is `rd` z `wr` z since in this case we also modify the object indicated by z. In both cases the existence check of z ensures that the effect is well-formed.

Method call is fairly simple and relies on a look-up to find the declared effect of the method. Substitutions are made to replace effects rooted at `this`, or any of the formal parameters with effects rooted at z or the formal parameters. The read effect is augmented with the receiver of the call. This is required to make ODE compliant with the model. LS5 states that the effects must describe a sub-heap which is sufficent to evaluate the expression. Inspection of the operational semantics will show that the receiver must be examined in order to choose a method to dispatch. Thus the effect must include the receiver.

As in the typing rules there are two rules for `let` expressions again handling the general case and that where b is final. When b is final we may be able to calulate more accurate effects (as for types). In both cases the effect is the union of the effects for each of the sub-expressions (b and e). The type environment is augmented for the recursive effect calculation of e (just as it would be for typing). The collected effects of b and e may include effects rooted with x but such effects will not be valid in the scope of γ. To counter this we either subsume the collected effects to a new effect (FX-LET) which is valid with respect to γ or replace references to x with b (FX-LETFINAL).

Consider the expression

```
let x = this.f in let y = x.m() in  y
```

where we assume the method `m` to have declared effect `rd this wr this` . The calculated effect of `x.m()` is thus `rd x wr x`. This effect is not valid in the outer scope (as `x` is not valid) and so must be subsumed to a valid effect or `x` must be substituted. If `this.f` is not final then we must subsume `x`. The smallest effect which contains `x` and is valid in the outer scope is `d` where `d` is the owner of `x`. Note that we will always know the owner of `x` as we have a type for `x`. This gives the overall effect `rd this+d wr d` for the expression. If `this.f` is final then we get `rd this+this.f wr this.f` . This effect refers to at most two objects whereas in the non-final case it refers to as many as are owned by `d`.

In FX-LET we include type checks for e. These are not strictly necessary for calculation of sound effects but are necessary to ensure property LS5 of the model. This is discussed further in Section 5.4.5.

4.6.2.10 Well-formed Programs

In Figure 4.25 we give rules for judging well-formedness of programs starting with WF-PROG. The basic requirements are that all types, effects and domain names used are valid. This includes satisfaction of class constraints. We omit a number of standard (and uninteresting) requirements in the formal rules and instead mention them in the following text.

A program (collection of classes) is well-formed if each of its class definitions is well-formed with respect to the rest of the program according to WF-CLASS. We also implicitly require that the class hierarchy induced by the `extends` clause is acyclic.

Classes are checked for validity with respect to a program so that the types referenced in the class definition can be validated. First, the declared superclass must be present in the program, with the expected parameters. Each domain declared in the class must be distinct from those declared in the superclasses (to avoid ambiguity). Similarly, the declared fields must be unique with respect to those in the superclass as we forbid overriding or shadowing of fields. We implictly assume that the list of field declarations in each class contains no duplicate field names. Furthermore we must check that the types declared for each field are well-formed when checked against a type environment which binds `this` to the class type. Link statements, methods and the constructor are checked for validity in sub rules.

Each link statement in the body of the class must be checked for validity. The WF-LINK rule performs this function. Link validity is a simple matter and is the same as in Ownership Domains. Link statements can be of three forms: linking a local domain to an external, linking an external to a local or linking two locals. Linking of two external domains is not permitted as this would break soundness as outlined in [AC04]. When linking an external domain to a local, the external domain must have permission to access the owner of the class (we don't want to give permissions to a domain about which we know nothing!). When linking a local to an external, the receiver must have permission to access the external domain, linking is, in this case, a delegation of the receiver's permissions. There are no restrictions on links between local domains (since the receiver may do what it likes with its own domains).

Constructors are checked for well-formedness with rule WF-CONS. The form $\pi \vdash cdef\ OK\ in\ c$ means that the definition $cdef$ is a valid constructor in class c. γ is constructed as a basic type environment where `this` is the type of the class being instantiated. A lookup of fields of the class is used to check that all fields are assigned in the body. The type system is used to check that the left hand side of each assignment is a subtype of the expected type for the field.

Methods are checked in WF-METH. The argument types, return types and declared effects of a method must correspond to those of any definition of that method in a superclass (checked with a call to *methods* with the superclass as the type argument). As with fields we tacitly assume there are no replicated methods in the list of declarations

$$\frac{\pi = \overline{cdef} \qquad \pi \vdash \overline{cdef}}{\vdash \pi}\ \text{WF-PROG}$$

$$\frac{\begin{array}{c}\pi(c') = \texttt{class}\ c'{<}\overline{p}{>} \ldots \\ \overline{dom} = [\texttt{public}]\ \texttt{domain}\ \overline{p''} \qquad \texttt{this}.\overline{p''} \notin \mathit{domains}(\pi, c'{<}\overline{p}{>}) \\ \pi \vdash \overline{lnk}\ \mathit{OK\ in}\ c \qquad \pi \vdash \mathit{constr\ OK\ in}\ c \\ \pi \vdash \overline{md}\ \mathit{OK\ in}\ c \qquad \overline{f} \notin \mathit{fields}(\pi, c'{<}\overline{p}{>}) \\ \pi, \texttt{this} \mapsto c{<}\overline{p}, \overline{p'}{>} \vdash \overline{t} \qquad \pi, \texttt{this} \mapsto c{<}\overline{p}, \overline{x'}{>} \vdash \texttt{this} \rightarrow \overline{t}\end{array}}{\begin{array}{l}\pi \vdash \texttt{class}\ c{<}\overline{p}, \overline{p'}{>}\ \texttt{extends}\ c'{<}\overline{p}{>}\ \texttt{assumes}\ \overline{con} \\ \quad \{\overline{dom}\ \ \overline{lnk}\ \ \overline{t}\ \overline{f}\ \ constr\ \ \overline{md}\}\end{array}}\ \text{WF-CLASS}$$

$$\frac{\begin{array}{c}\gamma = [\texttt{this} \mapsto c{<}\overline{p}{>}] \\ \{d_1, d_2\} \cap \mathit{domains}(\pi, c{<}\overline{p}{>}) \neq \emptyset \\ d_1 \notin \mathit{domains}(\pi, c{<}\overline{p}{>}) \ \Rightarrow\ \pi, \gamma \vdash d_1 \rightarrow \mathit{owner}(c{<}\overline{p}{>}) \\ d_2 \notin \mathit{domains}(\pi, c{<}\overline{p}{>}) \ \Rightarrow\ \pi, \gamma \vdash \texttt{this} \rightarrow d_2\end{array}}{\pi \vdash d_1 \rightarrow d_2\ \mathit{OK\ in}\ c}\ \text{WF-LINK}$$

$$\frac{\begin{array}{c}\gamma = \texttt{this} \mapsto c{<}\overline{p}, \overline{p'}{>} \qquad \pi, \gamma \vdash \texttt{this} \rightarrow \mathit{owner}(\overline{t}) \\ \gamma' = \gamma[\overline{x} \mapsto \overline{t}] \qquad \mathit{fields}(\pi, c{<}\overline{p}, \overline{p'}{>}) = \overline{t'}\ \overline{f} \\ \pi, \gamma' \vdash \overline{cexp} : \overline{t''} \qquad \pi \vdash \overline{t'} \leq \overline{t''}\end{array}}{\pi \vdash c{<}\overline{p}, \overline{p'}{>}(\overline{t}\ \overline{x})\{\ \texttt{this}.\overline{f} = \overline{cexp}\}\ \mathit{OK\ in}\ c}\ \text{WF-CONS}$$

$$\frac{\begin{array}{c}\pi(c{<}\overline{p}, \overline{p'}{>}) = \texttt{class}\ c{<}\overline{p}, \overline{p'}{>}\ \texttt{extends}\ c'{<}\overline{p}{>}\ \ldots \\ \mathit{methods}(\pi, c'{<}\overline{p}{>}, m)[\texttt{this}, \overline{x}/\texttt{that}, \overline{x^\dagger}] = t'_r\ m(\overline{t'}\ \overline{x})\texttt{rd}\ \phi''\ \texttt{wr}\ \phi'''\{e'\} \\ \Rightarrow\ t'_r = t_r,\ \overline{t'} = \overline{t},\ \phi'' = \phi,\ \phi''' = \phi' \\ \gamma = \texttt{this} \mapsto c{<}\overline{p}, \overline{p'}{>}, \overline{p} \mapsto \overline{t} \qquad \pi, \gamma \vdash e : t \qquad \pi \vdash t \leq t_r \\ \pi, \gamma \vdash \texttt{this} \rightarrow \mathit{owner}(\overline{t}) \\ \pi, \gamma \vdash e : \texttt{rd}\ \phi_r\ \texttt{wr}\ \phi_w \qquad \pi, \gamma \vdash \texttt{rd}\ \phi_r\ \texttt{wr}\ \phi_w \leq \texttt{rd}\ \phi\ \texttt{wr}\ \phi' \\ \pi, \gamma \vdash \texttt{rd}\ \phi\ \texttt{wr}\ \phi' \qquad \pi, \gamma \vdash t_r\end{array}}{\pi \vdash t_r\ m(\overline{t}\ \overline{x})\texttt{rd}\ \phi\ \texttt{wr}\ \phi'\{e\}\ \mathit{OK\ in}\ c{<}\overline{p}, \overline{p'}{>}}\ \text{WF-METH}$$

Figure 4.25: Well-formedness of Programs

(we do not allow overloading). We next check that the effect and type of the method body agrees with the declarations. The environment γ is formed from the types of the formal parameters and the class c. We check that the type of the expression is a subtype of the declared return type. We also check that the calculated effect is a sub-effect of the declared effect.

4.6.3 Runtime System

The runtime system of ODE consists of runtime structures (e.g. heaps, runtime types), rules for runtime calculation of permission and containment, operational semantics, projection of effects onto heaps and judgement of well-formed states. The rules for permission and containment complement the static rules, deducing relationships from the ownership structure present in the heap and also from the basic rules of the language. In general the rules are simpler, as we have complete knowledge of the heap and can construct the

ι	$\in$	$Addr$	heap addresses
v	$\in$	$Val = Addr \cup \{\texttt{null}\}$	values
fm	$\in$	$FM = \{fm \mid fm : f \rightharpoonup Val\}$	field maps
δ	$::=$	$\texttt{world} \mid \iota.p$	runtime domains
τ	$\in$	$Rtype ::= c < \overline{\delta} >$	runtime types
o	$\in$	$\texttt{Object} = Rtype \times FM$	objects
h	$\in$	$Heap = \{h \mid h : Addr \rightharpoonup \texttt{Object}\}$	heaps
s	$\in$	$Stack = \{s \mid s : z \rightharpoonup Val\}$	stacks
ω	$\in$	$\mathcal{P}(Addr)$	sets of addresses

Figure 4.26: Runtime Structures of ODE

relation from the basic forms in the heap and build them compositionally. As one expects the static system cannot infer all such relations (due to the conservative nature of typing) but in our soundness results we will require that in a well-formed state any containment or permission relation inferred in the static system also holds in the runtime state (after conversion of paths and parameter domains to their runtime equivalents).

The effect projection function serves in proofs of soundness of subeffects, disjoint effects and the effects rules for expressions. With it, we check that the static effect covers the actual effect at runtime.

The operational semantics are similar to `Joe` except for handling of domains and in rule B-NEW.

The rules for well-formed states provide a link between well-formed programs and valid states for those programs, these are fundemental to proofs of soundness, both in types and effects.

4.6.3.1 Runtime Structures

Figure 4.26 gives the definitions for the structures used in the runtime system of ODE.

Addresses index objects in the heap and are chosen from an infinite, countable set. In ODE the only values are addresses and `null`.

An object consists of its runtime type and a field map. Runtime types are just like their static counterparts except that parameters are instantiated with runtime domains. In turn domains are either `world` or $\iota.p$ indicating the object (address) that owns the domain and the name of the domain (as given in declaring class). Field maps are partial functions mapping field names to their values. Heaps are partial functions from addresses to objects and stacks are partial functions from program variables to values.

Stacks provide a mapping from program variables (including `this`) to their values at runtime. In this formalisation the term ‘frame’ is more appropriate than stack as, due to our large step semantics, we do not need to keep the frames from previous contexts. The operational semantics rules for method call and let expressions deal with this.

$$\frac{}{\pi, h \vdash \texttt{world}}\ \textsc{R-WORLD} \qquad \frac{h(\iota) = (\tau, fm) \quad p \in domains(\pi, \tau)}{\pi, h \vdash \iota.p}\ \textsc{R-DOM}$$

Figure 4.27: Well-formed runtime domains

$$\frac{\begin{array}{c}\pi, h \vdash \iota : \tau \\ \delta_1 \rightarrow \delta_2 \in links(\pi, \tau)[\texttt{this}/\iota]\end{array}}{\pi, h \vdash \delta_1 \rightarrow \delta_2}\ \textsc{RLINK-DECL} \qquad \frac{\delta}{\pi, h \vdash \delta \rightarrow \delta}\ \textsc{RLINK-REF}$$

$$\frac{\begin{array}{c}\pi, h \vdash \iota : \tau \\ \pi, h \vdash \delta \rightarrow owner(\tau) \\ public(\pi, \tau, p)\end{array}}{\pi, h \vdash \delta \rightarrow \iota.p}\ \textsc{RLINK-PUB} \qquad \frac{\begin{array}{c}\pi, h \vdash \iota : \tau \\ p \in domains(\pi, \tau)\end{array}}{\pi, h \vdash \iota \rightarrow \iota.p}\ \textsc{RLINK-LOCAL}$$

$$\frac{\begin{array}{c}\pi, h \vdash \iota : \tau \\ \pi, h \vdash owner(\tau) \rightarrow \delta\end{array}}{\pi, h \vdash \iota \rightarrow \delta}\ \textsc{RLINK-DEL} \qquad \frac{\begin{array}{c}\pi, h \vdash \iota' : \tau \\ \pi, h \vdash \iota \rightarrow owner(\tau) \\ public(\pi, \tau, p)\end{array}}{\pi, h \vdash \iota \rightarrow \iota'.p}\ \textsc{RLINK-OPUB}$$

Figure 4.28: Runtime rules for object/domain permissions

$$\frac{h(\iota) = (c{<}\overline{\delta}{>}, fm)}{h, \pi \vdash \iota < \delta_1}\ \textsc{ROD-DIRECT} \qquad \frac{\begin{array}{c}h(\iota) = (\tau, fm) \\ p \in domains(\pi, \tau)\end{array}}{h, \pi \vdash \iota.p < \iota}\ \textsc{RDO-DIRECT}$$

$$\frac{\begin{array}{c}h(\iota) = (\tau, fm) \\ p, p' \in domains(\pi, \tau)\end{array}}{h, \pi \vdash \iota.p\ \texttt{sibling}\ \iota.p'}\ \textsc{R-SIBLING} \qquad \frac{\begin{array}{c}h, \pi \vdash ?\ \$\ ?'' \\ h, \pi \vdash ?''\ \$'\ ?'\end{array}}{h, \pi \vdash ?\ \$ \sqcap \$'\ ?'}\ \textsc{R-COMPOSE}$$

$$\frac{\pi, h \vdash \delta}{\begin{array}{c}\pi, h \vdash \delta\ \texttt{sibling}\ \delta \\ \pi, h \vdash \delta <* \delta\end{array}}\ \textsc{R-SIBREF} \qquad \frac{h(\iota) = o}{\pi, h \vdash \iota <* \iota}\ \textsc{R-REF}$$

Figure 4.29: Runtime rules for object/domain containment

4.6.3.2 Permission and Containment Rules

As counterparts to the static versions, we have rules for determining permissions and containment between objects and domains. Rules for permissions can be found in Figure 4.28 and rules for containment in Figure 4.29. First we introduce the supplementary definitions of well-formed runtime domains in Figure 4.27. Valid runtime domains are `world` or $\iota.p$ where ι is a valid address and p is a domain of class of the object at ι.

The rules for link permissions are straightforward and replicate those in [AC04]. As in the static system the relation has two forms describing when an object may access a domain and when a domain may access a domain. The relation is populated primarily by rule RLINK-DECL which gives links that have been explicitly granted (with a `link` declaration) by some object in the heap (RLINK-DECL). This rule is the counterpart of LINK-ENV in the static system with the difference that RLINK-DECL may consider

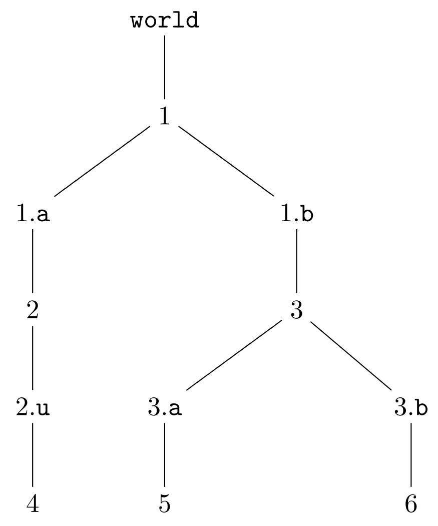


Figure 4.30: Containment Relation from heap in Figure 4.11

any object in the heap, rather than just those reachable by a final path.

The remaining rules deal with implicit permissions granted by the rules of Ownership Domains. Reflexivity is handled in rule RLINK-REF. Domains or objects may access domains of any object whose owning domain they may access (RLINK-PUB, RLINK-OPUB) as in the static rules LINK-DOMPATH and LINK-PATHPUBLIC. An object may access any of its own domains (RLINK-LOCAL, the counterpart to LINK-LOCAL) and any domains its owner may access (RLINK-DEL) just as in LINK-OWNERDEL .

The rules for containment are very similar to those in the static system, with only the assumes rule having no counterpart. The containment relation $<$ induced is thus exactly the tree structure of the heap. Figure 4.30 represents the containment relation in Figure 4.11. This relation is generated entirely by rules ROD-DIRECT and RDO-DIRECT. RDO-DIRECT states that a domain is below the object that created it (found just by removing the p from $\iota.p$), hence 2.u is below 2. ROD-DIRECT places an object below its owner e.g. 6 below 6.b by looking up the owner in the object's runtime type.

The R-COMPOSE rule provides transitive closure/reflex-transitive closure as in the static system, with the same rules for combining relations. Observe that we do not need a specific rule to cope with `world` since in well-formed heaps the containment relation will necessarily be rooted at `world` (otherwise there would be a cycle in the ownership hierarchy). Two domains are siblings when they are valid domains of the same object as in rule R-SIBLING. R-SIBREF and R-REF add reflexivity for sibling domains and between domains and addresses for containment.

4.6.3.3 Operational Semantics

The operational semantics of ODE are given in large step format. This choice is partly one of preference but we also find it helpful when dealing with effects as we are always considering the execution to termination, and thus the full effect. The rewrite relation is the smallest relation defined by the rules in Figure 4.31. Because all local variables are immutable and only `let`, `new` and method call involve any recursion, most of the rules are simple.

The rewrite relation is in the form $\varepsilon, h, s \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h'$ meaning that ε (an expression or block) evaluates, in the context of heap h, stack s, and program π to the result v and the heap is modified to h'. The sets of addresses ω and ω' give the addresses of the object read (ω) and written (ω') by the execution. They do not influence evaluation at all and will be used to calculate the required heap and in the soundness proofs.

The operational semantics follow those of `Joe` with some differences in presentation and adjustments to deal with domains.

B-NULL is trivial. The expression `null` evaluates to the value `null` and has no side-effects. Field lookup (B-FLD) is also straightforward, simply lookup the appropriate field in the object indicated by z. The recorded effect a read of the receiver of the lookup. Field assignment (B-ASS) replaces the target field in the receiver with the value of the RHS. The recorded effect is a read and a write in the receiver of the update.

Method call (B-CALL) follows the usual pattern. The receiver is evaluated and its type is found. Using the type of the receiver and the name of the method the appropriate method body is found. A new stack is built using the values of the receiver and parameters of the call. The method body is executed in this new context and the result of this execution is the result of the call. The effect of the method call is that of the method body augmented with the receiver. We include the receiver for the same reason as in FX-CALL - the receiver is required in order to choose a method to dipatch.

B-NEW is the most complicated rule, due to the complexity of constructors in ODE. First the constructor for the class to be instantiated is looked up and a new address (not already in use) is found. The heap is augmented with the new address and a prototype object of the appropriate class: the static type given in the `new` expression is converted using *map* to its runtime equivalent and all fields are initialized to `null`. The next step is a little contrived due to the syntax of constructors (see Section 4.6.1). The right hand side of each field assignment is evaluated and the appropriate field is updated with the result. After the evaluation of each of the right hand sides the value is put into the appropriate field of the new object. The update must be done 'manually' since the assignments are not valid expression syntax and so cannot be evaluated recursively. A runtime check is required for final fields. Because the types of other objects may use a final field in a path describing a domain we do not allow final fields to be `null`. Otherwise, the types of objects using such a field might be malformed and we would not have type soundness. An alternative would be a runtime check on the types of all newly created objects. We see it as a matter of taste as to which of these is employed. The effect of creating a new object is the empty set since the only objects which are modified are newly allocated and are not in the original heap. None of the right hand sides of the field assignments are side effecting (since they are variable lookup, `null` or object creation, which all have

$$\frac{}{\texttt{null}, s, h \overset{\emptyset,\emptyset}{\rightsquigarrow}_{\pi} \texttt{null}, h}\ \text{B-NULL} \qquad \frac{s(z) = \iota \qquad h(\iota)(f) = v}{z.f, s, h \overset{\{\iota\},\emptyset}{\rightsquigarrow}_{\pi} v, h}\ \text{B-FLD}$$

$$\frac{\begin{array}{c} s(z) = \iota \qquad s(z') = v \\ h(\iota) = (\tau, fm) \\ h' = h[\iota \mapsto (\tau, fm[f \mapsto v])] \end{array}}{z.f{=}z', s, h \overset{\{\iota\},\{\iota\}}{\rightsquigarrow}_{\pi} v, h'}\ \text{B-ASS} \qquad \frac{s(z) = v}{z, s, h \overset{\emptyset,\emptyset}{\rightsquigarrow}_{\pi} v, h}\ \text{E-VAR}$$

$$\frac{\begin{array}{c} cons(\pi, c\texttt{<}\overline{d}\texttt{>}) = c\texttt{<}\overline{d}\texttt{>}(\overline{t}\ \overline{x})\{\texttt{this}.\overline{f'} = \overline{cexp}\} \\ fields(\pi, c\texttt{<}\overline{d}\texttt{>}) = \overline{t}\ \overline{f} \qquad \iota \notin domain(h) \\ s(\overline{z}) = \overline{v} \qquad s' = [\texttt{this} \mapsto \iota][\overline{x} \mapsto \overline{v}] \\ h_0 = h[\iota \mapsto (c\texttt{<}map(\pi, s, h, \overline{d})\texttt{>}, [\overline{f'} \mapsto \texttt{null}])] \\ \forall i = 1, \ldots n: \quad cexp_i, s', h_{i-1} \overset{\emptyset,\emptyset}{\rightsquigarrow}_{\pi} v_i, h'_i \\ \qquad\qquad f'_i = f_j,\ t_j = \texttt{final}\ t \ \Rightarrow\ v_i \neq \texttt{null} \\ \qquad\qquad h_i = h'_i[\iota \mapsto h'_i(\iota)[f'_i \mapsto v_i]] \end{array}}{\texttt{new}\ c\texttt{<}\overline{d}\texttt{>}(\overline{z}), s, h \overset{\emptyset,\emptyset}{\rightsquigarrow}_{\pi} \iota, h_n}\ \text{B-NEW}$$

$$\frac{\begin{array}{c} s(z) = \iota \qquad h(\iota) = (\tau, fm) \\ methods(\pi, \tau, m) \ = \ \_\ m(\overline{\_\ x})\ \_\ \{e\} \\ s' = \texttt{this} \mapsto \iota, \overline{x} \mapsto \overline{s(z)} \\ e, s', h \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h' \end{array}}{z.m(\overline{z}), s, h \overset{\omega \cup \{\iota\},\omega'}{\rightsquigarrow}_{\pi} v, h'}\ \text{B-CALL}$$

$$\frac{\begin{array}{c} b, s, h \overset{\omega_1,\omega_2}{\rightsquigarrow}_{\pi} v', h'' \\ x \notin \texttt{domain}(s) \\ s' = s[x \mapsto v'] \\ e, s', h'' \overset{\omega_3,\omega_4}{\rightsquigarrow}_{\pi} v, h' \end{array}}{\texttt{let}\ x = b\ \texttt{in}\ e, s, h \overset{\omega_1 \cup \omega_3, \omega_2 \cup \omega_4}{\rightsquigarrow}_{\pi} v, h'}\ \text{E-LET}$$

$$\begin{array}{lcl} map(\pi, s, h, z) & = & s(z) \\ map(\pi, s, h, path.f) & = & h(\iota)(f) \text{ where } map(\pi, s, h, path) = \iota \\ map(\pi, s, h, p_i) & = & \delta_i \text{ where } h(s(\texttt{this})) \ = \ (c\texttt{<}\overline{\delta}\texttt{>}, fm), \\ & & \quad \pi(c) \ = \ \texttt{class}\ c\texttt{<}\overline{p}\texttt{>}\ \ldots \\ map(\pi, s, h, path.p) & = & \iota.p \text{ where } map(\pi, s, h, path) = \iota \\ map(\pi, s, h, \texttt{world}) & = & \texttt{world} \end{array}$$

Figure 4.31: Operational Semantics of ODE

$$\mathit{proj} \ : \ \mathit{Heap} \rightarrow \mathit{Stack} \rightarrow \mathit{\Phi} \rightarrow \mathcal{P}(\mathit{Addr})$$

$$\begin{array}{lcl}
\mathit{proj}(h, s, \texttt{emp}) & = & \emptyset \\
\mathit{proj}(h, s, \mathit{path}) & = & \{\ \iota\ \} \text{ if } \mathit{map}(\pi, h, s, \mathit{path}) = \iota \\
\mathit{proj}(h, s, \mathit{path}.f) & = & \{h(\iota)(f)\ \mid\ \mathit{proj}(h, s, \mathit{path}) = \{\iota\}\ \} \\
\mathit{proj}(h, s, d) & = & \{\ \iota \mid h(\iota) = (c{<}\overline{\delta}{>}, \mathit{fm}),\ \delta_1 = \mathit{map}(\pi, h, s, d)\ \} \\
\mathit{proj}(h, s, \mathit{path}.\texttt{all}) & = & \{\ \iota \mid h(\iota) = (c{<}\overline{\delta}{>}, \mathit{fm}),\ \delta_1 = \iota'.p\} \\
 & & \quad \text{if } \mathit{proj}(h, s, \mathit{path}) = \{\iota'\} \\
\mathit{proj}(h, s, d.\texttt{siblings}) & = & \{\ \iota \mid h(\iota) = (c{<}\overline{\delta}{>}, \mathit{fm}),\ \delta_1 = \iota'.p\} \\
 & & \quad \text{if } \mathit{map}(\pi, h, s, d) = \iota'.p' \\
\mathit{proj}(h, s, d.\texttt{siblings}) & = & \mathit{proj}(h, s, \texttt{world}) \text{ if } \mathit{map}(\pi, h, s, d) = \texttt{world} \\
\mathit{proj}(h, s, \theta.\texttt{under}) & = & \bigcup_{\iota \in \omega}\{\ \iota' \mid h, \pi \vdash \iota' <\!*\ \iota\ \} \text{ where } \omega = \mathit{proj}(h, s, \theta) \\
\mathit{proj}(h, s, \phi{+}\phi') & = & \mathit{proj}(h, s, \phi) \cup \mathit{proj}(h, s, \phi') \\
\mathit{proj}(h, s, \phi) & = & \emptyset \text{ in all other cases} \\
 & & \\
\mathit{proj}(h, s, \phi) & = & h \downarrow_{\mathit{proj}(h,s,\phi) \cap \mathit{domain}(h)}
\end{array}$$

Figure 4.32: Definition of projection of effects to heaps

empty effect).

Evaluation of a variable is just lookup of the variable in the stack. There is no side effect as the heap is not used. Let expressions are handled in E-LET and the subexpression b is first evaluated in the initial stack/heap. The result is added to the stack so long as the variable is not already in use (since we wish all variables to be final). The subexpression e is evaluated in the context of the modified heap and the augmented stack. The recorded effects are the unions of the effects of the subexpressions.

4.6.3.4 Projection of Effects

Figure 4.32 gives the projection function for effects. Projection of effects onto heaps $\mathit{proj}(h, s, \phi)$ takes an effect, program, heap and stack and returns the sub-heap of h the effect describes. The program is necessary in order to use *map*, we discuss in Section 5.4.1. The projection function is defined in terms of $\mathit{proj}(h, s, \phi)$, which gives the set of addresses of the objects referred to by ϕ. We will refer to the heap in Figure 4.11 and the diagram in Figure 4.30 in the following descriptions.

$\mathit{proj}(h, s, \phi)$ is the restriction of h to the addresses given by $\mathit{proj}(h, s, \phi)$. Thus it gives the sub-heap described by ϕ. The work is done by $\mathit{proj}(h, s, \phi)$.

The empty effect (`emp`) is mapped to the empty set as it represents an empty sub-heap. A simple path effect, *path*, is mapped to the address of the object it represents by appealing to the *map* function. As we would expect $\mathit{proj}(\mathsf{h}, s, \texttt{this.f})$ $=\{2\}$ and $\mathit{proj}(h, s, \texttt{this.g.f})$ $=\{5\}$. A domain expression d, is mapped to the addresses of all objects owned by the domain. d is translated into its runtime equivalent with *map* and then we simply choose all adresses where the type of the object lists that domain as the owner.

$$\frac{\pi(c) = \texttt{class}\ c\texttt{<}\overline{p}\texttt{>}\ \ldots \qquad |\{\overline{p}\}| = |\{\overline{\delta}\}| \qquad \pi, h \vdash \mathit{assumptions}(\pi, c\texttt{<}\overline{\delta}\texttt{>})}{\pi, h \vdash c\texttt{<}\overline{\delta}\texttt{>}}\ \text{WF-RTYPE}$$

$$\frac{\pi, h \vdash \tau}{\pi, h \vdash \texttt{null} : \tau}\ \text{TYPE-NULL} \qquad \frac{h(\iota) = (\tau, fm)}{\pi, h \vdash \iota : \tau}\ \text{TYPE-ADDR}$$

$$\frac{\pi, h \vdash \iota : \tau' \qquad \pi \vdash \tau' \leq \tau}{\pi, h \vdash \iota : \tau}\ \text{TYPE-SUBS}$$

Figure 4.33: Typing rules for runtime values

Effects of the form $path$.`all` and d.`siblings` are treated similarly. $path$.`all` means all object belonging to all domains owned by $path$. We simply find $path$ (ι') in the heap and collect all the addresses owned by domains beginning ι'. So, `this.all` is projected to $\{2, 3\}$ since 2 is owned by 1.`a` and 3 is owned by 1.`b`. Sibling effects are similar, in that we want all object in domains owned by the owner of d. Using map we calculate the runtime version of d (requiring it to be of the form $\iota'.p'$). Then, as with $path$.`all`, we collect all the addresses with owners owned by ι'. Thus both `this.g.all` and `this.g.b.siblings` give $\{5, 6\}$

Effects of the form θ.`under` are calculated by appealing to the runtime containment judgement. First we find the projection of θ with a recursive call. Then, for each adresses in the projection of θ, we collect all the addresses transitively and reflexively below it in the containment hierarchy. The result is the union of all these adresses. Thus $proj(h, s, \texttt{this.under}) = \{1, 2, 3, 4, 5, 6\}$ as all addresses in the heap are under 1 (the projection of `this`). $proj(h, s, \texttt{this.b.siblings}) = \{2, 3, 4, 5, 6\}$ since the projection of `this.b.siblings` is $\{2, 3\}$ and 4 is under 2 and 5 and 6 are under 3.

The projection of a union of effects is the set union of the projections of the sub-effects.

4.6.3.5 Runtime Typing and Well-formed States

Figure 4.33 gives rules for typing runtime values. These rules provide a 'weak' form of runtime typing which will be combined with a stronger judgement of conformance to determine well-formedness of heaps.

Rule TYPE-NULL states that `null` is compatible with any valid runtime type. Rule TYPE-ADDR states that an address can be given the type declared in the object it refers to. TYPE-SUBS allows subsumption of runtime types.

Figure 4.34 shows rules for determining well-formedness of heaps, with respect to a program and heap/stack pairs with respect to a program and a type environment.

A heap is well formed (WF-HEAP) if every object it contains *conforms* to the type

$$\frac{\begin{array}{c}\forall \iota \in \mathit{domain}(h).\ \ h(\iota) = (\tau, \mathit{fm}) \Rightarrow \pi, h \vdash \iota \lhd \tau \\ \forall \iota, \iota' :\ \ \pi, h \vdash \iota <\!*\ \iota',\ \ \pi, h \vdash \iota' <\!*\ \iota\ \Leftrightarrow\ \iota = \iota'\end{array}}{\pi \vdash h}\ \textsc{wf-heap}$$

$$\frac{\begin{array}{c}h(\iota) = (c{<}\overline{\delta}{>}, \mathit{fm}) \qquad \pi, h \vdash \mathit{assumptions}(\pi, c{<}\overline{\delta}{>}) \\ \mathit{fields}(\pi, c{<}\overline{p}{>}) = \overline{t}\ \overline{f} \qquad \mathit{domain}(\mathit{fm}) = \{\overline{f}\} \\ \forall i :\ \ t_i = \texttt{final}\ t'_i\ \Rightarrow\ \mathit{fm}(f_i) \neq \texttt{null} \\ \pi, h \vdash \mathit{fm}(\overline{f}) : \mathit{map}(\pi, h, \texttt{this} \mapsto \iota, \overline{t}) \\ \pi, h \vdash \iota \rightarrow \mathit{owner}(\mathit{map}(\pi, h, \texttt{this} \mapsto \iota, \overline{t}))\end{array}}{\pi, h \vdash \iota \lhd c{<}\overline{\delta}{>}}\ \textsc{conf-addr}$$

$$\frac{\begin{array}{c}\pi \vdash \gamma \qquad \mathit{domain}(s) = \mathit{domain}(\gamma) = \{\texttt{this}, \overline{x}\} \qquad \pi \vdash h \\ \pi, h \vdash s(\texttt{this}) \lhd \mathit{map}(\pi, h, s, \gamma(\texttt{this})) \qquad \pi, h \vdash s(\overline{x}) : \mathit{map}(\pi, h, s, \gamma(\overline{x}))\end{array}}{\pi, \gamma \vdash s, h}\ \textsc{wf-conf}$$

Figure 4.34: Judgements for well-formedness of runtime configurations

it declares. Heap well-formedness relies on the program to check existence and well-formedness of object and field types. Conformance of an object (CONF-ADDR) to a type is given by the relation $\pi, h \vdash \iota \lhd \tau$, meaning that, in the context of program π and heap h, the object at address ι has the features expected of type τ. These are that τ is a valid instantiation of a class in π with all constraints satisfied and that the object has precisely the fields expected by τ and that the objects referred to have the expected runtime type. The runtime type of the fields is calculated using a call to *map* with a stack which points `this` to the object in question. We also check that final fields are not `null` so as to ensure that types that refer to them are not badly formed. The final check is that the object has sufficent permission to access each of its fields.

Configurations of a program, type environment, heap and stack, are well formed (WF-CONF) in the relation $\pi, \gamma \vdash s, h$ if the heap is well-formed with respect to the program and that the stack and type environments agree. The specific conditions are i) that the type environment t is well-formed for the program π ii) the stack and type environment have the same domain i.e. they define the same set of program variables iii) that h is well-formed for π iv) the value of `this` conforms to the runtime equivalent of the type in γ v) every other value in the stack has the runtime type calculated from the static type in γ. This is enough to ensure that any value that can be reached in the heap from the stack will have the expected type.

4.6.3.6 Soundness

In the next chapter we show that ODE is an instance of our model of effects. This includes soundness results for ODE. Type soundness is given by L1. Soundness of effects is given by LS4.

Chapter 5

ODE as an instance of the effects model

In this chapter we show that ODE as described in Chapter 4 is an instance of our model of effects. To show this we prove that ODE obeys each of the properties required for heaps, languages, local languages and effects. In some places ODE does not exactly fit the format expected by the model e.g. the use of a type environment and program where the model expects just an environment. In these cases we either provide a translation to the format expected by the model or describe how such a translation can be achieved.

We omit treatment of predicates in this chapter. Proof techniques would be very similar to those used in the proofs for ODE.

5.1 Heaps

The structure of heaps, H, as in 3.1 is instantiated in ODE as:

$$\mathsf{HC}_{\mathrm{ODE}} = (\mathit{Heap}, \emptyset, *)$$

where $\emptyset$ is the heap with empty domain and $h * h'$ is defined if and only if $\mathit{domain}(h) \cap \mathit{domain}(h') = \emptyset$ and

$$h * h'(\iota) = \begin{cases} h(\iota) & \text{if } \iota \in \mathit{domain}(h) \\ h'(\iota) & \text{if } \iota \in \mathit{domain}(h') \\ \perp & \text{otherwise} \end{cases}$$

Thus the disjointness relation on heaps, $\#$, is defined for any two heaps with disjoint domains. The sub-heap relation, $\sqsubseteq$, holds when one heap is a restriction of the other.

Proof of satisfaction of the properties SH1 - SH6 are straightforward through application of the definitions and simple set/relational theory.

Theorem 5.1.1 $\mathsf{HC}_{\mathrm{ODE}}$ *is a structure of separable heaps.*

Proof: sketch

SH6 holds by the distributivity of set disjointness over set union. **SH1** and **SH2** hold by the corresponding commutativity and associativity of set union (along with **SH6**). The empty heap $\emptyset$ is an obvious identity under $*$ (**SH3**). For **SH4** consider the domains of h_1, h_2 and h_3. By simple set theory the domain of h_1 has a part in the domains of each of h_2 and h_1. The rest follows by the definition of $*$ and $\sqsubseteq$. **SH5** is straightforward as the domains of h and h' must be disjoint if the heaps are disjoint and thus their intersection is empty.

$\square$

5.2 Languages

ODE as presented in Chapter 4 is not quite in the format required by the model although a conversion/translation is easily achieved. We will first outline how this is achieved (with some auxiliary definitions) before proceeding to prove conformance with the properties L1 - L5.

5.2.1 Expressions, Stacks, Heaps and Values

For the set of expressions expected by the model we take the set consisting of ε. Stacks, heaps and values are exactly those described in Figure 4.26.

5.2.2 Programs, Environments and Well-Formedness

In the model we expect a set of programs and a set of environments. Programs are used in the operational semantics and environments are used to determine well-formedness of expressions, and configurations of stacks and heaps. Moreover an environment is checked for validity against a program.

In ODE programs π are purely syntactic and we have the judgement $\vdash \pi$ to decide if a program is reasonable. In ODE type environments, γ, contain only bindings from variables to types. In judgements where the model expects just an environment ODE uses a program and a type environment.

It is easy to solve this mismatch. We shall consider an environment in the sense of the model to be a pair of an ODE program and type environment. Validity of such an environment with respect to a program is given in Figure 5.1. A program will judge a

$$\frac{\pi \vdash \gamma}{\pi \vdash \pi, \gamma}\ \text{MODEL-ENV}$$

Figure 5.1: Well-formed Environments and Expressions

$$\frac{e, s, h \overset{\omega}{\rightsquigarrow}_{\pi} v, h'\omega' \quad h'' = h \downarrow_{domain(h) \cap \omega}}{e, s, h \overset{h''}{\rightsquigarrow}_{\pi} v, h'}$$

Figure 5.2: ODE operational semantics with required heaps

$$\frac{\begin{array}{c}\forall z.\ \ s(z) = v \ \Rightarrow\ v = \texttt{null} \ \vee\ v \in domain(h) \\ \forall \iota, f.\ \ h(\iota)(f) = v \ \Rightarrow\ v = \texttt{null} \ \vee\ v \in domain(h)\end{array}}{s \models h}\ \text{COMPLETE}$$

Figure 5.3: Judgement for Complete Heaps in ODE

program/type environment pair well-formed if and only if the two programs are equal and they judge the type-environment well-formed.

Further for the relation the model expects for well-formedness of expressions we simply use the ODE typing judgement and ignore the type assigned.

5.2.3 Operational Semantics

The operational semantics of ODE given in Section 4.6.3.3 differ from those expected by the model as ODE does not record a required heap but instead two sets of runtime effects. We can easily produce the expected form, as shown in Figure 5.2. The runtime read effect, ω, provides a record of every object read or written by the execution. This may include objects not allocated in h e.g. if an object is created and one of it's fields is then written. To construct the required heap, h'' we take the intersection of ω and the domain of h and restrict the domain of h to that set. Thus h'' contains all objects which are read/written and are present in h.

The version of ODE operational semantics with runtime effects is more convenient in many of the proofs that follow. Thus we will prove corresponding results with this version and then show that they imply the result expected by the model.

5.2.4 Complete Heaps

The rule found in Figure 5.3 defines the judgement for complete heaps. We require that every element of the range of s be `null` or in the domain of h. Further we require that every element of the range of the field map of every object in h be `null` or in the domain of h. This ensures that there are no dangling pointers in s, h and is sufficient to ensure L4.

A more sophisticated and less conservative definition could require only that all addresses reachable through variable and field accesses in s, h are not 'dangling'. This again would be sufficient for L4 but would complicate proofs.

5.2.5 L1

To show L1 we first prove the stronger property in Lemma 5.2.1

Lemma 5.2.1 *If*

135: $\pi, \gamma \vdash h, s$
136: $\pi, \gamma \vdash \varepsilon : t$
137: $\varepsilon, s, h \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h''$

then

138: $\pi, \gamma \vdash h', s$
139: $\pi, \gamma \vdash v : map(\pi, h, s, t)$

We omit full proof but provide the following sketch.

Proof: (sketch)

Proof is by induction on the structure of the derivation of 137 focusing on the last rule used. Most cases are straightforward although for B-ASS some work must be done to compare the types expected for well-formedness of the heap and those found in the type rule. A similar though more involved process is required in B-CALL where we must show that the types of the dynamically dispatched method match those found in the static lookup in the typing rule.

The most interesting case is B-NEW where we must appeal to a supplementary Lemma. Recall from B-NEW that all fields of the new object are initially set to `null`. This intermediate state may be badly formed as final fields may be `null`. Other fields in the new object may refer to one of these fields in a type, thus making it badly formed (as *map* will fail to calculate a runtime domain). Thus we are unable to apply our induction hypothesis to the right hand sides of the field assignments and we are stuck. We overcome this problem by application of Lemma 5.2.2, statement of which follows. We will discuss its application to Lemma 5.2.1 after the statement. □

Lemma 5.2.2 *If*

140: $\pi \vdash \gamma$
141: $domain(\gamma) = domain(s)$
142: $\pi, h \vdash s(\overline{z}) : map(\pi, h, s, \gamma(\overline{z}))$
143: $\pi, \gamma \vdash \texttt{new}\ t(\overline{z'}) : t$
144: $\texttt{new}\ t(\overline{z'}), s, h \overset{\emptyset;\emptyset}{\rightsquigarrow}_{\pi} \iota, h'$

then

145: $\forall \iota' \in domain(h')/domain(h).\;\; h'(\iota') = (\tau, fm) \;\Rightarrow\; \pi, h' \vdash \iota' \lhd \tau$

Lemma 5.2.2 is proved via induction on 144. The result holds since the fields of the new objects can only get instantiated with `null`, z or another new object. We know the first two options have the types we expect, trivially for `null` and by 142 for z and by induction for new objects. Thus we can show incrementally that each field holds a value of the correct type. Then finally we have constructed a new object which conforms to the expected types, moreover all other new objects do as well. In Lemma 5.2.1 where we could not use induction we can here as we do not care about the heap (in particular the partially formed object we are creating).

We have L1 proper by the following Lemma

Lemma 5.2.3 *If*

146: $\pi \vdash \pi, \gamma$
147: $\pi, \gamma \vdash \varepsilon : t$
148: $\pi, \gamma \vdash h, s$
149: $\varepsilon, s, h \overset{h'}{\rightsquigarrow}_{\pi} v, h''$

then

150: $\pi, \gamma \vdash h'', s$

Proof: By 149 and the definition of the operational semantics with a required heap there is an equivalent execution to 149 of the form in 137. Then we can simply apply Lemma 5.2.1. Note that $\pi, \gamma \vdash h, s$ is a little strong and has $\pi \vdash \gamma$ as a consequence.

□

5.2.6 L2

We prove L2 via the following lemma using the basic ODE operational semantics.

Lemma 5.2.4 *If*

151: $e, s, h \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h'$
152: $h' \# h''$

then

153: $h \# h''$

Proof: Proof is by induction on the structure of the derivation of 151 focusing on the last rule used. Cases B-NULL, B-FLD, E-VAR are trivial since in these cases $h' = h$.

Case: B-ASS

From B-ASS

154: $\varepsilon \;=\; z.f{=}z'$

155: $s(z) = \iota$

156: $s(z') = v$

157: $h(\iota) = (\tau, fm)$

158: $h' = h[\iota \mapsto (\tau, fm[f \mapsto v])]$

By 157

159: $\iota \in domain(h)$

and since 158, 159

160: $domain(h) = domain(h')$

Thus by 152, 160 and the definition of #

161: $h\#h''$

Case: B-CALL

153 follows simply by induction using the operational semantics derivation for the method body.

Case: B-NEW

From B-NEW

162: $\varepsilon = \texttt{new}\ t(\overline{z})$

163: $cons(\pi, t) \;=\; t(\overline{t}\ \overline{x})\ \{\overline{\texttt{this}.f \;=\; cexp}\}$

164: $\iota \notin domain(h)$

165: $s' \;=\; \texttt{this} \mapsto \iota,\ \overline{x} \mapsto s(\overline{z})$

166: $h_0 \;=\; h[\iota \mapsto (map(\pi, h, s, t), \overline{f} \mapsto \texttt{null})]$

167: $\forall i \in 1..n : \quad cexp_i, s', h_{i-1} \overset{\emptyset,\emptyset}{\rightsquigarrow}_{\pi} v_i, h'_i$

168: $\forall i \in 1..n : \quad h_i = h'_i[\iota \mapsto h'_i(\iota)[f_i \mapsto v_i]]$

169: $h' = h_n$

170: $\omega = \omega' = \emptyset$

By 152 and 169

171: $h''\#h_n$

Now since 168, 171

172: $h'_n[\iota \mapsto h'_n(\iota)[f_n \mapsto v_n]]\#h''$

Whether $\iota \in domain(h'_n)$ or not we still have

173: $h'_n \# h''$

as the domain of h'_n is a subset of $\mathit{domain}(h_n)$ and thus will still be disjoint from $\mathit{domain}(h'')$. From 167 we have

174: $\mathit{cexp}_n, s', h_{n-1} \overset{\emptyset,\emptyset}{\leadsto}_\pi v_n, h'_n$

and by 174, 173 and induction

175: $h_{n-1} \# h''$

Repeating from 172 to 175 $n-1$ times we get

176: $h_0 \# h''$

By 164, 166 see that

177: $\mathit{domain}(h) \subseteq \mathit{domain}(h_0)$

So by the definition of #, 177 and 176

178: $h \# h''$

Case: E-LET

From E-LET

179: $\varepsilon \ = \ \texttt{let}\ x = b\ \texttt{in}\ e$
180: $b, s, h \overset{\omega_1;\omega_2}{\leadsto}_\pi v', h'''$
181: $x \notin \mathit{domain}(s)$
182: $s' = s[x \mapsto v']$
183: $e, s', h''' \overset{\omega_3;\omega_4}{\leadsto}_\pi v, h'$
184: $\omega = \omega_1 \cup \omega_3$

By 152, 182 and induction get

185: $h''' \# h''$

and by 180, 185 and induction

186: $h \# h''$

$\square$

We also have the following corollary

Lemma 5.2.5

$$\varepsilon, s, h \overset{\omega,\omega'}{\leadsto}_\pi v, h', h' \# h'' \quad \Rightarrow \quad h \# h''$$

L2 follows as follows

187: $\varepsilon, s, h \overset{h'}{\rightsquigarrow}_\pi v, h''$
188: $h''\#h'''$

189: $h\#$

Lemma 5.2.6

Proof: From 187 there exists an equivalent derivation in the form of 151 so by that derivation, 188 and Lemma 5.2.4 we get 189. □

5.2.7 L3

L3 is a corollary of the following lemma using the basic (non-required heap) operational semantics for ODE.

Lemma 5.2.7 *If*

190: $\varepsilon, s, h \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h''$
191: $h' = h \downarrow_{domain(h)\cap\omega}$

then there exist $\mathsf{h'''}$, $\mathsf{h_r}$ *such that*

192: $\varepsilon, s, h' \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h'''$
193: $h = h' * h_r$
194: $h'' = h''' * h_r$

Proof: Proof is by induction on the structure of the derivation of 190 focusing on the last rule used. First let

195: $h_r = h \downarrow_{domainh/\omega}$

and notice by construction

196: $domain(h_r) \cap domain(h') = \emptyset$
197: $domain(h_r) \cup domain(h') = domain(h)$

Thus we find that

198: $h = h' * h_r$

and have 193. We now proceed with the case analysis of the last rule used in 190.

Cases B-NULL and E-VAR are trivial since i) $\omega = \emptyset$ (and so h_r is the empty heap) ii) $h = h''$ iiii) neither rule relies on the heap at all.

Case: B-FLD

From B-FLD

199: $\varepsilon \;=\; z.f$

200: $s(z) = \iota$

201: $h(\iota)(f) = v$

202: $\omega = \{\iota\}$

203: $\omega' = \emptyset$

204: $h'' = h$

By 202, 191 and 201

205: $h' = \iota \mapsto h(\iota)$

and so

206: $h'(\iota)(f) = h(\iota)(f) = v$

By 200, 206 and B-FLD

207: $z.f, s, h' \overset{\{\iota\},\emptyset}{\rightsquigarrow}_\pi v, h'$

which is 192. By 198 and 204 we get 194 and with 198 we are done.

Case: B-ASS

From B-ASS we have

208: $\varepsilon \;=\; z.f{=}z'$

209: $s(z) = \iota$

210: $s(z') = v$

211: $h(\iota) = (\tau, fm)$

212: $h'' = h[\iota \mapsto (\tau, fm[f \mapsto v])]$

213: $\omega = \omega' = \{\iota\}$

By 191, 213, 211

214: $h' = \iota \mapsto (\tau, fm)$

Let

215: $h''' = \iota \mapsto (\tau, fm[f \mapsto v])\ (= h'[\iota \mapsto (\tau, fm[f \mapsto v])])$ (214)

So by 209, 210, 214, 215 and B-ASS

216: $z.f{=}, s, h' \overset{\{\iota\},\{\iota\}}{\rightsquigarrow}_\pi v, h'''$

Now by 215

217: $domain(h') = domain(h''')$

218: $h'''\#h_r$ (217, 198, def. #)
219: $h''' * h_r = h'[\iota \mapsto (\tau,])fm[f \mapsto v] * h_r$ (215, 218)

As 218, ι is not in the domain of h_r and so from 219

220: $h''' * h_r = h' * h_r[\iota \mapsto (\tau, fm[f \mapsto v])]$
221: $h''' * h_r = h[\iota \mapsto (\tau, fm[f \mapsto v])]$ (198220)
222: $h''' * h_r = h''$ (212, 221)

Thus we have 192 by 216, 193 by 198 and 221.

Case: B-CALL

From B-CALL have

223: $\varepsilon \;=\; z.m(\overline{z})$
224: $s(z) = \iota$
225: $h(\iota) = (\tau, fm)$
226: $methods(\pi, \tau, m) = \_\ m(\_\ \overline{x})\_\{e'\}$
227: $s' = \texttt{this} \mapsto \iota, \overline{x} \mapsto s(\overline{z})$
228: $e', s', h \overset{\omega'',\omega'}{\rightsquigarrow}_\pi v, h''$
229: $\omega = \omega'' \cup \{\iota\}$

Let

230: $h'_1 = h \downarrow_{domain(h) \cap \omega''}$

Then by 228, 230 and induction there exist h_{r1} and h'''_1 such that

231: $e', s', h'_1 \overset{\omega'',\omega'}{\rightsquigarrow}_\pi v, h'''_1$
232: $h = h'_1 * h_{r1}$
233: $h'' = h''_1 * h_{r1}$

Since 229 and by construction of h' and h'_1 find

234: $h'_1 \sqsubseteq h'$

Say

235: $h' = h'_1 * h'_2$

By 198, 235 and SH2

236: $(h'_1 * h'_2) * h_r = h'_1 * (h'_2 * h_r)$

and by 234, 232 and Lemma 3.1.3

237: $h_{r1} = h'_2 * h_r$
238: $h'_2 \sqsubseteq h_{r1}$

By 238, 233 and Lemma 3.1.2

239: $h_1''\#h_2'$

By 239, 231 and Lemma 5.3.1

240: $e', s', h_1' * h_2' \overset{\omega'',\omega'}{\rightsquigarrow}_\pi v, h_1'' * h_2'$

and let

241: $h''' = h_1'' * h_2'$

We thus get 192 from 240, 235 and 241 and get 194 from 233, 237 and 241, .

Case: E-LET

From E-LET

242: $\varepsilon \ = \ \texttt{let}\ x = b\ \texttt{in}\ e$
243: $b, s, h \overset{\omega_1,\omega_2}{\rightsquigarrow}_\pi v', h_1$
244: $x \notin domain(s)$
245: $s' = s[x \mapsto v']$
246: $e, s', h_1 \overset{\omega_3,\omega_4}{\rightsquigarrow}_\pi v, h''$
247: $\omega = \omega_1 \cup \omega_3$
248: $\omega' = \omega_2 \cup \omega_4$

Let

249: $h_1' = h \downarrow_{\omega_1 \cap domain(h)}$

By 243, 249 and induction we have

250: $b, s, h_1' \overset{\omega_1,\omega_2}{\rightsquigarrow}_\pi v', h_1'''$
251: $h = h_1' * h_{r1}$
252: $h_1 = h_1''' * h_{r1}$

Notice that by 249

253: $h = h_1' * h \downarrow_{domain(h)/\omega_1}$
254: $h_{r1} = h \downarrow_{domain(h)/\omega_1}$ (251, 253, Lemma 3.1.3)

Let

255: $h_2' = h_1 \downarrow_{domain(h_1) \cap \omega_3}$

By 246, 255 and induction

256: $e, s', h_2' \overset{\omega_3,\omega_4}{\rightsquigarrow}_\pi v, h_2'''$
257: $h_1 = h_2' * h_{r2}$
258: $h'' = h_2''' * h_{r2}$

Notice that by 255

259: $h_1 = h'_2 * h \downarrow_{domain(h_1)/\omega_3}$
260: $h_{r2} = h_1 \downarrow_{domain(h_1)/\omega_3}$ (257, 259, Lemma 3.1.3)

Pick ω_5, ω_6 and ω_7 so that

261: $\omega_5 = \omega_1 \cap \omega_3$
262: $\omega_6 = \omega_1/\omega_5$
263: $\omega_7 = \omega_3/\omega_5$

and let

264: $h_5 = h \downarrow_{domain(h)\cap\omega_5}$
265: $h_6 = h \downarrow_{domain(h)\cap\omega_6}$
266: $h_7 = h \downarrow_{domain(h)\cap\omega_7}$

Now notice that

267: $\omega_1 = \omega_5 \cup \omega_6$ (261,262)
268: $\omega_3 = \omega_5 \cup \omega_7$ (261,263)
269: $\omega = \omega 5 \cup \omega 6 \cup \omega 7$ (247,267,268)
270: $h' = h_5 * h_6 * h_6$ (191,268,264,265,266 and def. $*$)
271: $h'_1 = h_5 * h_6$ (249, 267, 264, 265)
272: $h = h_5 * h_6 * h_7 * h_r$ (198, 270)

By 251, 271, 272 and Lemma 3.1.3

273: $h_{r1} = h_r * h_7$

and thus by 252, 273 and Lemma 3.1.2

274: $h_7 \# h'''_1$

By 274, 250 and Lemma 5.2.4

275: $b, s, h'_1 * h_7 \overset{\omega_1,\omega_2}{\rightsquigarrow}_\pi v', h'''_1 * h_7$

By 271, 270 and 275 we get

276: $b, s, h' \overset{\omega_1,\omega_2}{\rightsquigarrow}_\pi v', h'''_3$
277: $h'''_3 = h'''_1 * h_7$

Moreover, by 252, 273 and 277

278: $h_1 = h'''_3 h_r *$

We now aim to show that $h'_2 \sqsubseteq h'''_3$. By 255

279: $h'_2 \sqsubseteq h_1$

and so by 279, 278 and SH4 there exist h_8, h_9 such that

280: $h'_2 = h8 * h9$
281: $h_8 \sqsubseteq h'''_3$
282: $h_9 \sqsubseteq h_r$

By construction (195, 247)

283: $domain(h_r) = domain(h)/(\omega_1 \cup \omega_3)$

and thus

284: $domain(\phi_r) \cap \omega_3 = \emptyset$

but as 255

285: $domain(h'_2 \subseteq \omega_3)$

but $domain(h_9) \subseteq domain(h_r)$ (282) and $domain(h_9 \subseteq domain(h'_2))$ (280) but since 284

286: $h_9 = \emptyset$

By 280, 286 and SH3

287: $h'_2 = h_8$

and thus by 281

288: $h'_2 \sqsubseteq h'''_3$
289: $h'''_3 = h'_2 * h_{10}$

for some h_{10}.

By 257, 278 and 289

290: $h_1 = h'''_3 * h_r = h'_2 * h_{10} * h_r = h'_2 * h_{r2}$

and so by 290 and Lemma 3.1.3

291: $h_{10} * h_r = h_{r2}$

291 and 258 gives

292: $h_{10} * h_r \# h'''_2$

and with Lemma 3.1.2

293: $h_{10} \# h'''_2$
294: $h_r \# h'''_2$

By 256, 293 and Lemma 5.2.4

295: $e, s', h'_2 * h_{10} \overset{\omega_3;\omega_4}{\rightsquigarrow}_\pi v, h'''_2 * h_{10}$

Since 289

296: $e, s', h'''_3 \overset{\omega_3;\omega_4}{\rightsquigarrow}_\pi v, h'''$
297: $h''' = h''_2 * h_{10}$

By 276, 244, 245, 296, 247, 248 and E-LET we have

298: $\texttt{let } x = b \texttt{ in } e, s, h' \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h'''$

By 258, 291, 293, 294 and 297

299: $h'' = h''' * h_r$

298 gives 192, 198 gives 193 and 299 gives 194.

Case: B-NEW

From B-NEW

300: $\varepsilon = \texttt{new } t(\overline{z})$
301: $cons(\pi, t) \ = \ t(\bar{t}\ \overline{x})\ \{\overline{\texttt{this}.f \ = \ cexp}\}$
302: $\iota \notin domain(h)$
303: $s' \ = \ \texttt{this} \mapsto \iota,\ \overline{x} \mapsto s(\overline{z})$
304: $h_0 \ = \ h[\iota \mapsto (map(\pi, h, s, t), \overline{f} \mapsto \texttt{null})]$
305: $\forall i \in 1..n : \quad cexp_i, s', h_{i-1} \overset{\emptyset,\emptyset}{\rightsquigarrow}_\pi v_i, h'_i$
306: $\forall i \in 1..n : \quad h_i = h'_i[\iota \mapsto h'_i(\iota)[f_i \mapsto v_i]]$
307: $h' = h_n$
308: $\omega = \omega' = \emptyset$

Since 308 and 191 and 195

309: $h' = \emptyset$
310: $h_r = h$

Notice from from 309 that

311: $\iota \notin domain(h')$

and let

312: $h_0^\dagger = \iota \mapsto (map(\pi, h, s, t), \overline{f} \mapsto \texttt{null})$

and since 302, 304 and 312

313: $h_0 = h * h_0^\dagger$
314: $h_0^\dagger \sqsubseteq h_0$

Notice that in 305 all runtime effects are $\emptyset$ and so, by induction

315: $\forall i \in 1..n: \quad cexp_i, s', \emptyset \overset{\emptyset,\emptyset}{\rightsquigarrow}_\pi v_i, h_i'^\dagger$
316: $\forall i \in 1..n: \quad h_i' = h_i'^\dagger * h_{i-1}$

as the required heap in each case is the empty heap and the remainder is the initial heap. By 316, 314 and Lemma 3.1.2

317: $h_0^\dagger \# h_1'^\dagger$

Thus by Lemma 315, 317 and 5.3.1

318: $cexp_0, s', h_0^\dagger \overset{\emptyset,\emptyset}{\rightsquigarrow}_\pi v_1, h_i'^\dagger * h_0^\dagger$

Since 312, 317

319: $h_i'^\dagger * h_0^\dagger[\iota \mapsto h_i'^\dagger * h_0^\dagger(\iota)[f_1 \mapsto v_1]] = h_0^\dagger[\iota \mapsto h_0^\dagger(\iota)[f_1 \mapsto v_1]] * h_1'^\dagger$

So, by 313, 316, 319 and 306

320: $\mathsf{h}_1 = h_0^\dagger[\iota \mapsto h_0^\dagger(\iota)[f_1 \mapsto v_1]] * h_1'^\dagger * h$

Now, again by 316, 320 and Lemma 3.1.2

321: $h_0^\dagger[\iota \mapsto h_0^\dagger(\iota)[f_1 \mapsto v_1]] * h_1'^\dagger \# h_2'^\dagger$

If we repeat from 317 to 318 $n-1$ times we find

322: $\forall i \in 1..n: \quad cexp_i, s', h_{i-1}'' \overset{\emptyset,\emptyset}{\rightsquigarrow}_\pi v_i, h_i'''$
323: $h_i'' = h_1'^\dagger * \ldots * h_{i-1}'^\dagger * \mathsf{h}_0[f_1 \mapsto v_1]\ldots[f_{i-1} \mapsto v_{i-1}]$
324: $h_i''' = h_1'^\dagger * \ldots * h_i'^\dagger * \mathsf{h}_0[f_1 \mapsto v_1]\ldots[f_{i-1} \mapsto v_{i-1}]$
325: $h_1'^\dagger * \ldots * h_i'^\dagger * \mathsf{h}_0[f_1 \mapsto v_1]\ldots[f_i \mapsto v_i] * h = h_i$

301, 312, 322 gives 192. 325 gives 194 and 309, 310 gives 193.

□

L3 follows easily from Lemma 5.2.7

Lemma 5.2.8 *If*

326: $\varepsilon, s, h \overset{h'}{\rightsquigarrow}_\pi v, h''$

then

327: $h = h * h_r$
328: $h'' = h''' * h_r$
329: $\varepsilon, s, h' \overset{h'}{\rightsquigarrow}_\pi v, h'''$

Proof: From 326 and the definition of the operational semantics with required heaps there exist ω, ω' such that

330: $\varepsilon, s, h \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h''$
331: $h' = h \downarrow_{\omega \cap domain(h)}$

By Lemma 5.2.7, 330 and 331 we have

332: $\varepsilon, s, h' \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h'''$
333: $h = h' * h_r$
334: $h'' h''' * h_r$

By 331

335: $h' = h' \downarrow_{\omega \cap domain(h')}$

and thus by 335 and 332 and the definition of the operation semantics with required heaps

336: $\varepsilon, s, h' \overset{h'}{\rightsquigarrow}_{\pi} v, h'''$

which along with 333 and 334 gives us L3. $\square$

5.2.8 L4

Proving that ODE satisfies L4 has proved to be the most challenging part of showing the conformance of ODE to the model. Initial attempts to prove it directly failed and resulted in the development of additional theory (of no relevance to the rest of ODE) to assist the proof.

Abstractly L4 states that for any execution on a complete heap/stack pair, if the execution creates any new sub-heap, subsequent executions can only access the new part of the heap via the part of the original heap that was modified. This introduces the concept of reachability in the stack/heap and also the impact of modification of the heap on reachability.

Figure 5.4 gives the definition for relations describing the reachability of addresses in the heap. We also introduce a short hand for calculation of *path* entities (the definition is as in *map* but as we don't need π we simplify). The relation $s, h \models path$ via ι means that *path* uses the object ι somewhere in the calculation of $|path|_{h,s}$. The definition exploits the recursive nature of the path lookup function. $s, h \models path$ via ω is similar but states that the path uses at least one address in ω. The final relation $s, h \models \omega$ dom ι states that all paths which reach ι must go through ω. Thus ω dominates ι. As a sanity condition we require that ι is in $domain(h)$.

Lemma 5.2.9 considers the first execution in L4 and shows that in the resulting heap if h_3 is reachable it must be dominated by h_2'.

Lemma 5.2.9 *If*

$$s, h \models path \text{ via } \iota \iff \exists path'.\ path = path'.f_1 \dots f_n,\ |path'|_{h,s} = \iota,\ n \geq 0$$

$$s, h \models path \text{ via } \omega \iff s, h \models path \text{ via } \iota,\ \iota \in \omega$$

$$s, h \models \omega \text{ dom } \iota \iff \iota \in domain(h),\ \forall path.\ |path|_{h,s} = \iota \Rightarrow s, h \models path \text{ via } \omega$$

$$\begin{array}{lcl} |z|_{h,s} & = & \begin{cases} s(z) & \text{if } z \in domain(s) \\ \bot & \text{otherwise} \end{cases} \\ |path.f|_{h,s} & = & \begin{cases} v & \text{if } |path|_{h,s} = \iota,\ h(\iota)(f) = v \\ \bot & \text{otherwise} \end{cases} \end{array}$$

Figure 5.4: Judgements for reachability relations

337: $s \models h_1 * h_2$
338: $\varepsilon, s, h_1 * h_2 \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h_1 * h_2' * h_3$
339: $domain(h_2) = domain(h_2')$
340: $\iota \in domain(h_3)$

then

341: $s, h_1 * h_2' * h_3 \models domain(h_2') \text{ dom } \iota$

Proof: First notice that if there are no paths in $s, h_1 * h_2' * h_3$ then 341 is trivial. So instead suppose

342: $|path|_{h_1 * h_2' * h_3, s} = \iota$
343: $path = z.f_1 \dots f_n$

We will prove 341 by contradiction so suppose

344: $s, h_1 * h_2' * h_3 \not\models path \text{ via } domain(h_2')$

and therefore

345: $\forall i \in 0..n-1.\ |z.f_1 \dots f_i|_{h_1 * h_2' * h_3, s} \notin domain(h_2')$

Now, since 337

346: $\forall z'.\ s(z') \in domain(h_1 * h_2) = domain(h_1) \cup domain(h_2)$

By 338

347: $h_1 * h_2' \# h_3$

and so with 339 and the definitions of $*$, $\#$

348: $h_1 * h_2 \# h_3$
349: $domain(h_3) \cap domain(h_1 * h_2) = \emptyset$

Now by 349, 346, 342, 343 there exists $k \in 1 \dots n-1$ such that

350: $\forall j \in 1 \dots k.\ |z.f_1 \dots f_k|_{h_1 * h_2' * h_3, s} \notin \mathit{domain}(h_3)$
351: $|z.f_1 \dots f_{k+1}|_{h_1 * h_2' * h_3, s} \in \mathit{domain}(h_3)$

i.e. there must be a prefix of *path* which does not use h_3. It could be as short as z or as long as $z.f_1 \dots f_{n-1}$.

By 337

352: $\forall \iota' \in \mathit{domain}(h_1 * h_2).\forall f.\ h_1 * h_2(\iota')(f) = v \ \Rightarrow v = \texttt{null} \ \vee \ v \in \mathit{domain}(h_1 * h_2)$

so, in particular

353: $\forall \iota' \in \mathit{domain}(h_1).\forall f.\ h_1(\iota')(f) = v \ \Rightarrow v = \texttt{null} \ \vee \ v \in \mathit{domain}(h_1 * h_2)$

By the definition of the path lookup function and 351

354: $|z.f_1 \dots f_k|_{h_1 * h_2' * h_3, s} \in \mathit{domain}(h_1 * h_2' * h_3)$

since otherwise $|z.f_1 \dots f_{k+1}|_{h_1 * h_2' * h_3, s}$ would not be defined.

By 345

355: $|z.f_1 \dots f_k|_{h_1 * h_2' * h_3, s} \notin \mathit{domain}(h_2')$

Since 354, 355, 339, 350 we must find that

356: $|z.f_1 \dots f_k|_{h_1 * h_2' * h_3, s} \in \mathit{domain}(h_1)$

but by 352

357: $h_1 * h_2' h_3(|z.f_1 \dots f_k|_{h_1 * h_2' * h_3, s})(f_{k+1}) \in \mathit{domain}(h_1 * h_2)$

then by definition of the path lookup function

358: $|z.f_1 \dots f_{k+1}|_{h_1 * h_2' * h_3, s} \in \mathit{domain}(h_1 * h_2)$

which contradicts 351. Thus 344 cannot hold and we have 341. $\square$

Lemma 5.2.15 will be used to consider the second execution in L5. Following from Lemma 5.2.9 we will then see that if any of h_3 is required in the second execution, since it is dominated by h_2', some of h_2' must be required as well. First we present Lemmas 5.2.10 and 5.2.14 which will be used in the proof of Lemma 5.2.15.

Lemma 5.2.10 *If*

359: $\varepsilon, s, h \overset{\emptyset,\emptyset}{\rightsquigarrow}_\pi v, h'$
360: $x \notin \mathit{domain}(s)$

361: $s' = s[x \mapsto v]$

then

362: $s, h \models \omega \text{ dom } \iota \Leftrightarrow s', h' \models \omega \text{ dom } \iota$

We omit proof of Lemma 5.2.10 which is also by induction on the derivation of the execution. The only case of real interest is for B-NEW where we show that as each field of the new object is updated any new paths that are generated are still dominated by ω. This follows since the fields of new objects can only be initiated with the parameters to the constructor, `null` or another new object (which has the same property).

In subsequent proofs we will use the following simple Lemmas

Lemma 5.2.11 *If* $|path|_{h,s} = |path'|_{h,s}$ *then* $|path.f_1 \ldots f_n|_{h,s} = |path'.f_1 \ldots f_n|_{h,s}$

Lemma 5.2.12 *If*

363: $\varepsilon, s, h \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h'$
364: $\overline{x} \notin domain(s)$
365: $\overline{x}$ are not declared in ε
366: $s' = s[\overline{x} \mapsto \overline{v}]$

then

367: $\varepsilon, s', h \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h'$

Lemma 5.2.13 is a little more involved than the previous two. It states that for any arbitrary path we can construct another path with the same value which has no cycles. Further any set of addresses which the new path uses must also be used by the old path.

Lemma 5.2.13 *If*

368: $|path|_{h,s} = \iota$

then there exists $path' = z.f_1 \ldots f_n$ *such that*

369: $|path'|_{h,s} = \iota$
370: $\forall i, j \in 1 \ldots n. \;\; |z.f_1 \ldots f_i|_{h,s} = |z.f_1 \ldots f_j|_{h,s} \Leftrightarrow i = j$
371: $\forall \omega. \;\; s, h \models path' \text{ via } \omega \;\Rightarrow\; s, h \models path \text{ via } \omega$

Now follows Lemma 5.2.14. This proved to be the most challenging of all. The original formulation did not consider the extended stack, s' and as such the case for E-LET proved impossible. The version presented here is stronger than the original and dispenses with a great deal of additional theory that was developed to prove the original.

The original version also included a second clause that if $s, h \models \omega \text{ dom } v$ then $\omega \cap \omega_1 \neq \emptyset$.

This was proved alongside the result below and was needed in some inductive cases. Again, the strengthened version did not require this.

Lemma 5.2.14 *If*

372: $\varepsilon, s, h \overset{\omega_1,\omega_2}{\leadsto}_\pi v, h'$
373: $\omega, \omega_1 \neq \emptyset$
374: $x \notin domain(s)$
375: $s' = s[x \mapsto v]$
376: $s, h \models \omega \text{ dom } \iota$
377: $s', h' \not\models \omega \text{ dom } \iota$

then

378: $\omega \cap \omega_1 \neq \emptyset$

Proof: Proof is by induction on the structure of the derivation of 372 focusing on the last rule used. Cases B-NULL, E-VAR and B-NEW are degenerate as in these cases $\omega_1 = \emptyset$.

Case: B-FLD

From B-FLD

379: $\varepsilon \ = \ z.f$
380: $s(z) = \iota'$
381: $h(\iota')(f) = v$
382: $h' = h$
383: $\omega_1 = \{\iota\}$
384: $\omega_2 = \emptyset$

Since 377 there exists *path* such that

385: $|path|_{h',s'} = \iota$
386: $h', s' \not\models path$ via ω

Since 382 paths in h,s are unchanged in h',s' so we must have

387: $path = x.f_1 \dots f_n$

otherwise *path* would be in h,s and by 376 would go through ω, contradicting 386. By 385, 386 and 387

388: $\forall i \in 0 \dots n-1.\ |x.f_1 \dots f_i|_{h',s'} \notin \omega$

Notice that by 380, 381 and definition of the path lookup function

389: $|z..|_{h,s} = v$
390: $|z..|_{h',s} = v$ (382, 389)

and by 375 (and 390)

391: $|x|_{h',s'} = \mathsf{v}$
392: $|z.f|_{h',s'} = v$

From 391, 392 and Lemma 5.2.11

393: $\forall i \in 0..n |z.f.f_1 \dots f_i|_{h',s'} = |x.f_1 \dots f_i|_{h',s'}$

and in particular since 385 and 387

394: $|z.f.f_1 \dots f_n|_{h',s'} = |x.f_1 \dots f_n|_{h',s'} = \iota$

and since $z.f.f_1 \dots f_n$ does not mention x and 382

395: $|z.f.f_1 \dots f_n|_{h,s} = |x.f_1 \dots f_n|_{h',s'} = \iota$

Now, since 376 and 395

396: $h, s \models z.f.f_1 \dots f_n$ via $h\omega$

By 388 and 393 and 388

397: $\forall i \in 0 \dots n.\ |z.f.f_1 \dots f_i|_{h',s'} \notin \omega$

So if 396 it must be the case that

398: $|z|_{h',s'} \in \omega$

By 380

399: $|z|_{h',s'} = \iota'$

so by 398, 399

400: $\iota' \in \omega$

and by 383, 400

401: $\omega_1 \cap \omega = \{\iota'\}$

Case: B-ASS

From B-ASS

402: $\varepsilon = z.f{=}z'$
403: $s(z) = \iota$
404: $s(z') = v$
405: $h(\iota') = (\tau, fm)$
406: $h' = h[\iota' \mapsto (\tau, fm[f \mapsto v])]$
407: $\omega_1 = \omega_2 = \{\iota'\}$

If 377 is to hold there must exist $path$ such that

408: $|path|_{h',s'} = \iota$
409: $h', s' \not\models path$ via ω

Since 404 we can assume that $path$ does not begin at x and so

410: $|h'|_{s,=}\iota$
411: $h', s \not\models path$ via ω

By Lemma 5.2.13 and 408 there exists $path'$ such that

412: $|path'|_{h',s} = \iota$
413: $s, h' \models path'$ via $\omega \Rightarrow s, h' \models path$ via ω

By 413 and 411

414: $h', s \not\models path'$ via ω

By 406 the only difference between h and h' is in field f of ι' and so if $h', s \not\models$ $path'$ via ι' $path'$ is the same in s,h but then we have a contradiction with 414 and 376. So we must find that there exists k such that $0 \leq k \leq n$ and

415: $path' = z''.f_1 \ldots f_n$
416: $|z''.f_1 \ldots f_k|_{h',s} = \iota'$
417: $f_{k+1} = f$
418: $|z''.f_1 \ldots f_{k+1}|_{h',s} = v$

By 403

419: $|z'|_{h',s} = v$

and by 419, 416 and Lemma 5.2.11

420: $\forall i \in 0..n.\ |z'.f_{k+1} \ldots f_n|_{h',s} = |z''.f_1 \ldots f_n|_{h',s}$

Since $path'$ is cycle free and since 416

421: $\forall i \in k+1 \ldots n.\ |z''.f_1..f_i|_{h',s} \neq \iota'$

and so by 421, 420

422: $\forall i \in k+1 \ldots n.\ |z'.f_{k+1} \ldots f_i|_{h',s} \neq \iota'$

By 422 $z'.f_{k+1} \ldots f_n$ is a path in s,ha nd so by 376

423: $h', s \models z'.f_{k+1} \ldots f_n$ via ω

but then by 5.2.11 and 423

424: $h', s \models path'$ via ω

which contradicts 414.

Case: E-LET

From E-LET

425: $\varepsilon \ = \ \texttt{let}\ x' = b\ \texttt{in}\ e$
426: $b, s, h \overset{\omega_3;\omega_4}{\rightsquigarrow}_\pi v', h'''$
427: $x' \notin \mathit{domain}(s)$
428: $s'' = s[x' \mapsto v']$
429: $e, s'', h'' \overset{\omega_5;\omega_6}{\rightsquigarrow}_\pi v, h'$
430: $\omega_1 = \omega_3 \cup \omega_5$
431: $\omega_2 = \omega_4 \cup \omega_6$

First assume, without loss of generality that

432: $x \neq x'$

so by 427, 428

433: $x \notin \mathit{domain}(s'')$

and let

434: $s''' = s''[x \mapsto v]$

Since induction requires ω_3, ω_5 to be non-empty we perform case-analysis on them

Case: $\omega_3 \neq \emptyset$, $\omega_5 = \emptyset$

Suppose

435: $\omega_3 \neq \emptyset$
436: $\omega_5 = \emptyset$

Notice by 376, 433, 434 and the definition of *dom* that

437: $s''', h' \not\models \omega \text{ dom } \iota$

and so from 429, 433, 434, 437 and Lemma 5.2.10

438: $s'', h'' \not\models \omega \text{ dom } \iota$

then by 426, 373, 435, 427, 428 and induction

439: $\omega \cap \omega_3 \neq \emptyset$

439 and 435 give 378

Case: $\omega_3 = \emptyset$, $\omega_5 \neq \emptyset$

Suppose

440: $\omega_3 = \emptyset$

441: $\omega_5 \neq \emptyset$

By 426, 440, 427, 428, Lemma 5.2.10 and 376

442: $s'', h'' \models \omega$ dom ι

and since 433, 434 and 377 we easily see

443: $s''', h' \not\models \omega$ dom ι

since the paths in s''',h' are a superset of those in s',h'.

By 429, 373, 441, 433, 434, 442, 443 and induction we have

444: $\omega_5 \cap \omega \neq \emptyset$

444 and 430 gives 378.

Case: $\omega_3 \neq \emptyset$, $\omega_5 \neq \emptyset$

Suppose

445: $\omega_3 \neq \emptyset$

446: $\omega_5 \neq \emptyset$

We must have either $s'', h'' \models \omega$ dom ι or $s'', h'' \not\models \omega$ dom ι

Case: $s'', h'' \models \omega$ dom ι

Suppose

447: $s'', h'' \models \omega$ dom ι

By the same argument as in 437

448: $s''', h' \not\models \omega$ dom ι

then by 429, 373, 446, 433, 434, 447, 448 and induction

449: $\omega \cap \omega_5 \neq \emptyset$

449 and 430 gives 378.

Case: $s'', h'' \not\models \omega$ dom ι

Suppose

450: $s'', h'' \not\models \omega$ dom ι

By 426, 373, 445, 427, 428, 376 and 450 and induction we have

451: $\omega \cap \omega_3 \neq \emptyset$

and 451 and 430 gives 378.

Case: B-CALL

From B-CALL

452: $\varepsilon = z.m(\overline{z})$
453: $s(z) = \iota'$
454: $h(\iota') = (\tau, fm)$
455: $methods(\pi, \tau, m) = \_\ m(\_\ \overline{x})\ \_\ \{e'\}$
456: $s'' = \texttt{this} \mapsto \iota, \overline{x} \mapsto s(\overline{z})$
457: $e', s'', h \overset{\omega'', \omega'}{\rightsquigarrow}_\pi v, h'$
458: $\omega_1 = \omega_3 \cup \{\iota'\}$

Let

459: $Z = domain(s)/\{z, \overline{z'}\} = \{\overline{z'}\}$
460: $X = \{\overline{x'}\}$
461: $|X| = |Z|$
462: $x' \in X \ \Rightarrow\ x' \notin domain(s''),\ x'$ is not mentioned in e'

Define s''' such that

463: $s'''(z) = \begin{cases} s''(z) & \text{if } z \in domain(s'') \\ s(z_i) & \text{if } z = x'_i \\ \perp & \text{otherwise} \end{cases}$

By Lemma 5.2.12, 457 and 463

464: $e', s''', h \overset{\omega_3, \omega_2}{\rightsquigarrow}_\pi v, h'$

Notice that the paths in s''',h are the same as in s,h up to renaming of variables so

465: $s, h \models \omega \text{ dom } \iota \Leftrightarrow s''', h \models \omega \text{ dom } \iota$

Now pick x' such that

466: $x' \notin domain(s''')$

and let

467: $s'''' = s'''[x' \mapsto v]$

Again paths in s'''',h' are the same as in s',h' up to renaming of variables so

468: $s''', h' \models \omega \text{ dom } \iota \Leftrightarrow s', h' \models \omega \text{ dom } \iota$

By 376 and 465 and 377 and 468

469: $s''', h \models \omega \text{ dom } \iota$
470: $s''', h' \models \omega \text{ dom } \iota$

We now consider whether $\omega_2 = \emptyset$ or not.

Case: $\omega_2 = \emptyset$

Suppose

471: $\omega_2 = \emptyset$

then by 464, 466, 467, Lemma 5.2.10 and 469 get

472: $s'''', h' \models \omega \text{ dom } \iota$

a contradiction!

Case: $\omega_2 \neq \emptyset$

Suppose

473: $\omega_2 \neq \emptyset$

then by 464, 373, 473, 466, 467, 469, 470 and induction gives

474: $\omega \cap \omega_2 \neq \emptyset$

then 474 and 458 gives 378.

$\square$

Lemma 5.2.15 *If*

475: $\varepsilon, s, h \overset{\omega_1;\omega_2}{\rightsquigarrow}_{\pi} v, h'$

476: $\iota \in \omega_1$

477: $s, h \models \omega \text{ dom } \iota$

then

478: $\omega \cap \omega_1 \neq \emptyset$

Proof: Proof is by induction on the structure of the derivation of 475 focusing on the last rule used.

The cases B-NULL, E-VAR and B-NEW are degenerate as in these cases $\omega_1 = \emptyset$.

Case: B-FLD

From B-FLD

479: $\varepsilon = z.f$

480: $s(z) = \iota'$

481: $h(\iota')(f) = v$

482: $h = h'$

483: $\omega_1 = \{\iota'\}$

484: $\omega_2 = \emptyset$

By 483 and 476

485: $\iota' = \iota$

By 480 and the definition of the path lookup function (and 485)

486: $|z|_{h,s} = \iota'$

but then there is no ω such that 477

Case: B-ASS

The same as for B-FLD.

Case: E-LET

From E-LET

487: $\varepsilon \;=\; \texttt{let}\ x = b\ \texttt{in}\ e$
488: $b, s, h \overset{\omega_3;\omega_4}{\rightsquigarrow}_{\pi} v', h'''$
489: $x \notin domain(s)$
490: $s' = s[x \mapsto v']$
491: $e, s', h''' \overset{\omega_5;\omega_6}{\rightsquigarrow}_{\pi} v, h'$
492: $\omega_1 = \omega_3 \cup \omega_5$
493: $\omega_2 = \omega_2 \cup \omega_6$

Consider the following cases

Case: $\iota \in \omega_3$

Suppose

494: $\iota \in \omega_3$

then by 488, 494 and 477

495: $\omega \cap \omega_3 \neq \emptyset$

So by 495 and 492 have 478.

Case: $\iota \in \omega_5$

Suppose

496: $\iota \in \omega_5$

Consider two further cases; either $s', h'' \models \omega$ dom ι or $s'', h'' \not\models \omega$ dom ι:

Case: $s', h'' \models \omega$ dom ι

Suppose

497: $s', h'' \models \omega$ dom ι

then by 491, 496, 497 and induction

498: $\omega_5 \cap \omega \neq \emptyset$

and by 498 and 492 we have 478.

Case: $s', h'' \not\models \omega \text{ dom } \iota$

Suppose

499: $s', h'' \not\models \omega \text{ dom } \iota$

Now since 499 and 477 we must find that

500: $\omega_3 \neq \emptyset$

as otherwise 499, 477, 488, 489 and 490 contradict Lemma 5.2.10. So by 488, 496, 500, 489, 490 and Lemma 5.2.14

501: $\omega \cap \omega_3 \neq \emptyset$

501 and 492 gives 478.

Case: B-CALL

From B-CALL

502: $\varepsilon = z.m(\overline{z})$
503: $s(z) = \iota'$
504: $h(\iota') = (\tau, fm)$
505: $methods(\pi, \tau, m) = \_ \; m(\_ \; \overline{x}) \; \_ \; \{e'\}$
506: $s' = \texttt{this} \mapsto \iota', \overline{x} \mapsto s(\overline{z})$
507: $e', s', h \overset{\omega_3,\omega_2}{\rightsquigarrow}_\pi v, h'$
508: $\omega_1 = \omega_3 \cup \{\iota\}$

We shall consider two cases; $\iota \in \omega_3$ and $\iota \notin \omega_3$

Case: $\iota \in \omega_3$

Suppose

509: $\iota \in \omega_3$

By 506

510: $range(s') \subseteq range(s)$

and so the paths in s',h are all in s, h up to renaming of variables and thus since 477

511: $s', h \models \omega \text{ dom } \iota$

then by 507, 509, 511 and induction

512: $\omega_3 \cap \omega \neq \emptyset$

which with 508 gives 478.

Case: $\iota \notin \omega_3$

513: $\iota \notin \omega_3$

then by 476 and 508

514: $\iota = \iota'$

but then by 503

515: $|z|_{h,s} = \iota'$

and so we can't have 477.

$\square$

Now we tie Lemmas 5.2.9 and 5.2.15 in Lemma 5.2.16 which in turn gives us L4.

Lemma 5.2.16 *If*

516: $s \models h_1 * h_2$
517: $e, s, h_1 * h_2 \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h_1 * h_2' * h_3$
518: $e', s, h_1 * h_2' * h_3 \overset{\omega_1,\omega_2}{\rightsquigarrow}_\pi v', h'$
519: $domain(h_2) \subseteq domain(h_2')$
520: $domain(h_3) \cap h_1 \neq \emptyset$

then

521: $domain(h_2') \cap \omega_1 \neq \emptyset$

Proof: Since 519 let

522: $h_2' = h_2'' * h_2'''$
523: $domain(h_2'') = domain(h_2)$

then from 517

524: $e, s, h_1 * h_2 \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h_1 * h_2'' * (h_2''' * h_3)$

By 524, 516, 522 and Lemma 5.2.9

525: $\forall \iota \in domain(h_2''' * h_3).\ s, h_1 * h_2' * (h_2''' * h_3) \models domain(h_2'')$ dom ι

and so

526: $\forall \iota \in domain(h_3).\ s, h_1 * h_2' * (h_2''' * h_3) \models domain(h_2'')$ dom ι

By 522 and definition of $*$

527: $domain(h_2'') \subseteq domain(h_2')$

and so by 527, 526 and the definition of domination

528: $\forall\iota \in domain(h_3).\ s, h_1 * h'_2 * (h'_2 * h_3) \models domain(h'_2)$ dom ι

Pick ι such that

529: $\iota \in domain(h_3) \cap \omega_1$

From 529 and 525

530: $s, h_1 * h'_2 * h_3 \models domain(h'_2)$ dom ι

and by 524, 530, 518 and Lemma 5.2.15

531: $\omega_1 \cap domain(h'_2) \neq \emptyset$

□

Finally we use Lemma 5.2.16 to show that ODE conforms to L4 as per the model

Lemma 5.2.17 *If*

532: $s \models h_1 * h_2$
533: $e, s, h_1 * h_2 \overset{h}{\rightsquigarrow}_\pi v, h_1 * h'_2 * h_3$
534: $e', s, h_1 * h'_2 * h_3 \overset{h'}{\rightsquigarrow}_\pi v', h''$
535: $h_3 \# h_2$
536: $h_3 \sqcap h' \neq \emptyset$

then

537: $h'_2 \sqcap h' \neq \emptyset$

Proof: From 533

538: $h_3 \# h_1$
539: $h_3 \# h'_2$
540: $h'_2 \# h_1$

By 533 and definition of $*$

541: $domain(h_1 * h'_2 * h_3) = domain(h_1) \cup domain(h'_2) \cup domain(h_3)$
542: $domain(h_1 * h_2) = domain(h_1) \cup domain(h_2)$

By Lemma 5.2.5

543: $domain(h_1 * h_2) \subseteq domain(h_1 * h'_2 * h_3)$

and so along with 541, 542

544: $domain(h_2) \subseteq domain(h_2' * h_3)$

but since 535

545: $domain(h_2) \cap domain(h_3) =\neq$

so by 546, 544

546: $domain(h_2) \subseteq domain(h_2')$

By the definition of 534 there exist ω_1 and ω_2 such that

547: $e', s, h_1 * h_2' * h_3 \overset{\omega_1;\omega_2}{\rightsquigarrow}_\pi v', h''$
548: $h' = h_1 * h_2' * h_3 \downarrow_{domain(h_1 * h_2' * h_3) \cap \omega_1}$

Since 536 there exists a non-empty h''' such that

549: $h''' \sqsubseteq h_3$
550: $h''' \sqsubseteq h'$

and so

551: $domain(h''') \subseteq domain(h_3)$
552: $domain(h''') \subseteq domain(h')$

From 552 and 548

553: $domain(h''') \subseteq domain(h_1 * h_2' * h_3) \cap \omega_1$

so

554: $domain(h'') \subseteq \omega_1$

By 551 and 553

555: $domain(h_3) \cap \omega_1 \neq \emptyset$

since h'' is non-empty.

Now by 532, 533, 547 , 546, 555 and Lemma 5.2.16 we have

556: $domain(h_2') \cap \omega_1 \neq \emptyset$

Pick h^* so that

557: $h^* = h_2' \downarrow_{domain(h_2' \cap \omega_1)}$

so

558: $h^* \sqsubseteq h'_2$

and from 548 and 557

559: $h^* \sqsubseteq h'$

and thus by 558 and 559

560: $h^* \sqsubseteq h'_2 \sqcap h'$

Since 556 h^* is non-empty and thus with 560 we have 537. $\square$

5.2.9 L5

L5 requires that in a well-formed environment, stack and heap configuration the heap is complete with respect to the stack.

Lemma 5.2.18 *If*

561: $\pi, \gamma \vdash h, s$

then

562: $s \models h$

Proof:

From 561 we have

563: $\pi, h \vdash s(\texttt{this}) : c{<}\overline{\delta}{>}$
564: $\pi, h \vdash s(\overline{x}) : \mathit{map}(\pi, h, s, \gamma(\overline{x}))$
565: $\pi \vdash h$

From 563, 564 we see that

566: $\forall z.\ \ s(z) = v \ \Rightarrow\ v = \texttt{null} \ \vee\ v \in \mathit{domain}(h)$

(it is a simple induction on the derivation of the runtime typing judgement to see that the value being typed must be `null` or in $\mathit{domain}(h)$).

From 565 find

567: $\forall \iota \in \mathit{domain}(h).\ \ h(\iota) = (c{<}\overline{\delta}{>}, fm) \ \Rightarrow \pi, h \vdash \iota \lhd c{<}\overline{\delta}{>}$

and in turn from 567, for all ι in $\mathit{domain}(h)$

568: $\pi(c) = \texttt{class}\ c{<}\overline{p}{>}...$

569: $map(\pi, h, \texttt{this} \mapsto \iota, fields(\pi, c\texttt{<}\overline{p}\texttt{>})) = \overline{\tau}\ \overline{f}$
570: $\overline{f} = domain(fm)$
571: $\pi, h \vdash fm(\overline{f}) : \overline{\tau}$

As in 566, 571 gives

572: $fm(\overline{f}) = \texttt{null} \ \vee \ fm(\overline{f}) \in domain(h)$

Since ι is an arbitrary member of $domain(h)$ and since 570 guarantees all f in $domain(fm)$ are checked in 571, by 572, 566 and COMPLETE we get 562. □

Theorem 5.2.1

$(L_{\text{ODE}} = \pi\ ,\ \varepsilon\ ,\ Stack\ ,\ Heap\ ,\ Val\ ,\ \_,\ (\pi,\gamma)\ ,\ \_,\_,\_ \overset{\bar{}}{\rightsquigarrow}_{\_}\ \_,\_\ ,\ \_ \vdash \_,\_\ ,\ \_,\_ \vdash \_ : \_\ ,\ \_,\_ \vdash \_,\_\ ,\ \_\ ,\ \_ \models \_)$

is a Programming Language

Proof: Theorem 5.2.1 follows by Theorem 5.1.1 and Lemmas 5.2.3, 5.2.6, 5.2.8, 5.2.17 and 5.2.18.

□

5.3 Local Languages

Since we are not considering predicates in this chapter this section consists only of the proof of LL2.

Lemma 5.3.1 *If*

573: $\varepsilon, s, h \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h'$
574: $h' \# h''$

then

575: $\varepsilon, s, h * h'' \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h' * h''$

Proof: Proof is by induction on the structure of the derivation of 573 focusing on the last rule used. Cases B-NULL and E-VAR are trivial since they do not use h.

Case: B-FLD

From B-FLD

576: $\varepsilon = z.f$
577: $s(z) = \iota$
578: $h(\iota)(f) = v$

579: $h = h'$

Since 579 and 574 $h * h'$ is defined and by 578 $\iota \in domain(h)$ and definition of $*$

580: $h * h''(\iota) = h(\iota)$
581: $h * h''(\iota)(f) = v$ (580,578)

Then by 577, 581 and B-FLD we get 575.

Case: B-ASS

From B-ASS

582: $\varepsilon = z.f{=}z'$
583: $s(z) = \iota$
584: $s(z') = v$
585: $h(\iota) = (\tau, fm)$
586: $h' = h[\iota \mapsto (\tau, fm[f \mapsto v])]$
587: $\omega = \omega' = \{\iota\}$

Since $\iota \in domain(h)$ by 585 and by 586

588: $domain(h) = domain(h')$
589: $h\#h''$ (574,588)
590: $h * h'(\iota) = h(\iota) = (\tau, fm)$ (589,585, def. $*$)

Now

591: $h * h''[\iota \mapsto (\tau, fm[f \mapsto v])] = h[\iota \mapsto (\tau, fm[f \mapsto v])] * h''$ (585,588)
592: $h' * h'' = h[\iota \mapsto (\tau, fm[f \mapsto v])] * h''$ (586)

So by 583, 584, 590, 591 and 592, 587 and B-ASS we get 575

Case: B-CALL

From B-CALL

593: $\varepsilon = z.m(\overline{z})$
594: $s(z) = \iota$
595: $h(\iota) = (\tau, fm)$
596: $methods(\pi, \tau, m) = \_\ m(\_\ \overline{x})\ \_\ \{e'\}$
597: $s' = \texttt{this} \mapsto \iota, \overline{x} \mapsto s(\overline{z})$
598: $e', s', h \overset{\omega'',\omega'}{\rightsquigarrow}_{\pi} v, h'$
599: $\omega = \omega'' \cup \{\iota\}$

By 598, 574 and induction

600: $e', s', h \overset{\omega'',\omega'}{\rightsquigarrow}_{\pi} v, h'$

From 600

601: $h\#h''$

and so by 595, 601 and def. $*$

602: $h * h''(\iota) = (\tau, fm)$

Then by 594, 602, 596, 597, 600 and B-CALL get 575.

Case: E-LET

From E-LET

603: $\varepsilon \ = \ \texttt{let}\ x = b\ \texttt{in}\ e'$
604: $b, s, h \overset{\omega_1;\omega_2}{\rightsquigarrow}_\pi v', h'''$
605: $x \notin \mathit{domain}(s)$
606: $s' = s[x \mapsto v']$
607: $e', s', h''' \overset{\omega_3;\omega_4}{\rightsquigarrow}_\pi v, h'$
608: $\omega = \omega_1 \cup \omega_3$
609: $\omega' = \omega_2 \cup \omega_4$

By 607, 574 and induction

610: $e', s', h''' * h'' \overset{\omega_3;\omega_4}{\rightsquigarrow}_\pi v, h' * h''$
611: $h'''\#h''$ (610)

Then by induction with 604 and 611

612: $b, s, h * h'' \overset{\omega_1;\omega_2}{\rightsquigarrow}_\pi v', h''' * h''$

610, 605, 606, 612, 608 and E-LET give 575.

Case: B-NEW

From B-NEW

613: $\varepsilon \ = \ \texttt{new}\ t(\overline{z})$
614: $\mathit{cons}(\pi, t) \ = \ t(\overline{t}\ \overline{x})\ \{\overline{\texttt{this}.f \ = \ \mathit{cexp}}\}$
615: $\iota \notin \mathit{domain}(h)$
616: $s' \ = \ \texttt{this} \mapsto \iota,\ \overline{x} \mapsto s(\overline{z})$
617: $h_0 \ = \ h[\iota \mapsto (\mathit{map}(\pi, h, s, t), \overline{f} \mapsto \texttt{null})]$
618: $\forall i \in 1..n: \quad \mathit{cexp}_i, s', h_{i-1} \overset{\emptyset,\emptyset}{\rightsquigarrow}_\pi v_i, h'_i$
619: $\forall i \in 1..n: \quad h_i = h'_i[\iota \mapsto h'_i(\iota)[f_i \mapsto v_i]]$
620: $h' = h_n$
621: $\omega = \omega' = \emptyset$

From 619

622: $\mathit{domain}(h'_n) \subseteq \mathit{domain}(h_n)$

623: $\iota \in domainh_n$

Thus by 622, 620 and 574

624: $h'_n \# h''$
625: $\iota \notin domainh''$

Since 625

626: $h'_n * h''(\iota) = h'_n(\iota)$

By 626 and 624

627: $(h'_n * h'')[\iota \mapsto (h'_n * h'')(\iota)[f_n \mapsto v_n]] = (h'_n[\iota \mapsto h'_n(\iota)[f_n \mapsto v_n]]) * h''$

By 624, 618 and induction

628: $cexp_n, s', h_{n-1} * h'' \overset{\emptyset,\emptyset}{\rightsquigarrow}_\pi v_n, h'_n * h''$

and from 628

629: $h_{n-1} \# h''$

Repeating from 622 to 629 $n-1$ times gives

630: $\forall i \in 1..n: \quad cexp_i, s', h_{i-1} * h'' \overset{\emptyset,\emptyset}{\rightsquigarrow}_\pi v_i, h'_i * h''$
631: $\forall i \in 1..n: \quad (h'_i * h'')[\iota \mapsto (h'_i * h'')(\iota)[f_i \mapsto v_i]] = (h'_i[\iota \mapsto h'_i(\iota)[f_i \mapsto v_i]]) * h''$

From 630

632: $h_0 \# h''$

By 617

633: $domain(h) \subseteq h_0$

and thus with 625 and 615 we find both

634: $h \# h''$
635: $\iota \notin domain(h*)$
636: $(h * h'')[\iota \mapsto (map(\pi, h, s, t), \overline{f} \mapsto \texttt{null})] = h[\iota \mapsto (map(\pi, h, s, t),]) * h''$

By 614, 635, 616, 636, 630, 631 621 and B-NEW get 575.

$\square$

Lemma 5.3.2 *If*

637: $\varepsilon, s, h \overset{h'}{\rightsquigarrow}_\pi v, h''$
638: $h'' \# h'''$

then

639: $\varepsilon, s, h * h''' \overset{h'}{\rightsquigarrow}_\pi v, h'' * h'''$

Proof: LL2 follows from Lemma 5.3.1 after applying the definition of the operational semantics with required heaps. □

Theorem 5.3.1 ODE *is a local language.*

By Lemma 5.3.2

5.4 Languages with Effects

With the caveat which follows in Section 5.4.1 ODE fits the format expected of a 'Language With Effects'. We show satisfaction of the properties LS1 - LS5 in the following sections.

5.4.1 The Effect Projection Function

The reader will notice that the projection function for effects $proj(h, s, \phi)$ is not in the format expected by the model. The function requires the program π which in turn is required when the projection function appeals to *map* to resolve paths and domains. The program is required by *map* in order to match static parameter domains to the corresponding runtime domain. Had ODE been designed to treat types in the style of `Joe`, with a function mapping formal parameters to actuals this problem could have been avoided. Alternatively this could be viewed as a suggestion that the model should be revised to include the program or environment in the effect projection function. We shall proceed in the knowledge that the problem can be fixed.

We now show that ODE is a Language with Effects.

5.4.2 LS1

Lemma 5.4.1 *If* $proj(h, s, \phi) = h'$ *then* $h' \sqsubseteq h$.

Proof: The result follows directly from the definition of $proj(h, s, \phi)$ since h' is a restriction of h. □

5.4.3 LS2

We now prove LS2 which gives soundness of the judgement for disjoint effects. First we introduce Lemma 5.4.2 which gives soundness of the sub-effect judgement.

Lemma 5.4.2 *If*

640: $\pi, \gamma \vdash h, s$
641: $\pi, \gamma \vdash \phi \leq \phi'$

then

642: $proj(h, s, \phi) \subseteq proj(h, s, \phi')$

Proof: (sketch) Proof is by induction on the structure of the derivation of 641 and using the definition of $proj(h, s, \phi)$ and $proj(h, s, \phi)$. Even cases involving `under` are not difficult as inclusion is evident in the definition of the projection function. □

Lemma 5.4.3 *If*

643: $\pi, \gamma \vdash h, s$
644: $\pi, \gamma \vdash \phi \# \phi'$

then

645: $proj(h, s, \phi) \cap proj(h, s, \phi') = \emptyset$

Proof: (sketch) Proof is by induction on the structure of the derivation of 644 focusing on the last rule used. Most cases are straightforward. DIS-CONT and DIS-OWNED are the most complicated cases as they use results on the ownership structure of the heap and the containment relations. In DIS-OWNED we must take care to show the sub-trees under each of the targets of the `under` shape have no objects in common. This follows from 643. □

LS2 proper easily follows since the heaps projected from each effect will have disjoint domains and thus are disjoint heaps and have no intersection.

Lemma 5.4.4 *If*

646: $\pi, \gamma \vdash h, s$
647: $\pi, \gamma \vdash \phi \# \phi'$

then

648: $proj(h, s, \phi) \sqcap proj(h, s, \phi') = \emptyset$

5.4.4 LS4

LS4 captures the general soundness of effects for expressions. Proof is given in 5.4.17 which relies heavily on 5.4.12. A number of smaller supplementary Lemmas are also required and we introduce them as appropriate. We omit proofs of these Lemmas but discuss or sketch in their stead. A number of these results are also used in proofs of earlier Lemmas most notably in Lemma 5.2.1.

Lemma 5.4.5 is a standard property of the type system, that in any well-formed environment (and thus well-formed program) any type derived for some expression must be well-formed. Proof is by induction and relies on the well-formedness of declared types in the program.

Lemma 5.4.5 *If*

649: $\pi \vdash \gamma$
650: $\pi, \gamma \vdash \varepsilon : t$

then

651: $\pi, \gamma \vdash t$

Lemma 5.4.6 is the analogue of 5.4.5 for effects. Proof is also by induction and works in much the same way.

Lemma 5.4.6 *If*

652: $\pi \vdash \gamma$
653: $\pi, \gamma \vdash \varepsilon : \texttt{rd}\ \phi\ \texttt{wr}\ \phi'$

then

654: $\pi, \gamma \vdash \texttt{rd}\ \phi\ \texttt{wr}\ \phi'$

Lemma 5.4.7 gives the property we expect of our final paths, that they do not change during execution.

Lemma 5.4.7 *If*

655: $\pi, \gamma \vdash h, s$
656: $\pi, \gamma \vdash \varepsilon : t$
657: $\varepsilon, s, h \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h'$
658: $\pi, \gamma \vdash \mathit{path} : \texttt{final}\ t$

then

659: $|\mathit{path}|_{h,s} = |\mathit{path}|_{h',s}$

Proof: (sketch) Since ε is typeable no field assignments occurring during its execution can change any field marked final. Because *path* is final every field mentioned in *path* must be marked `final`. Thus every field lookup performed in $|\mathit{path}|_{h,s}$ has the same value in h' and we have the result. □

Lemma 5.4.8 is a small result which links the result of a field lookup with the behaviour of the path lookup function. This Lemma is used in case FX-LETFINAL in Lemma 5.4.12 and along with Lemma 5.4.11 to show that the effect after substitution of the final path is the same as before.

Lemma 5.4.8 *If* $|z.f|_{h,s}$ *is defined then* $z.f, s, h \overset{|z|_{h,s},\emptyset}{\rightsquigarrow}_{\pi} |z.f|_{h,s}, h$

Lemma 5.4.9 is an important result for soundness of effects. It states that the projection of a well-formed effect in a heap before execution is equal to the projection in the heap after execution restricted to the domain of the starting heap.

Lemma 5.4.9 *If*

660: $\pi, \gamma \vdash h, s$
661: $e, s, h \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h'$
662: $\pi, \gamma \vdash e : \psi$
663: $\pi, \gamma \vdash \phi$

then

664: $\mathit{proj}(h', s, \phi) \cap \mathit{dom}(h) = \mathit{proj}(h, s, \phi)$

Proof: (sketch) Since ϕ is well-formed it contains only references to final paths and valid domains. Since ε is well-formed the value of all final paths is the same in h as in h' (Lemma 5.4.7). No objects are removed from the heap by execution and no objects can change their position in the ownership hierarchy. Thus all objects in the projection on h are also in the projection on h'. No objects in h can have been 'missed' by $pathhs\phi$ since the properties that define their inclusion in a shape (references from final fields and position in the ownership hierarchy) have not changed. The projection on h' can be larger since new objects may be created in domains mentioned by ϕ or below the target of an `under` effect. □

The following is a corrolary to Lemma 5.4.9

Lemma 5.4.10 *If*

665: $\pi, \gamma \vdash s, h$
666: $\pi, \gamma \vdash \phi$
667: $\pi, \gamma \vdash \varepsilon : t$
668: $\varepsilon, s, h \overset{4}{\rightsquigarrow}_{\pi} v, h'$

then

669: $\mathit{proj}(h, s, \phi) \subseteq \mathit{proj}(h', s, \phi)$

Lemma 5.4.11 is a substitution Lemma for effects. It states that the projection of an effect where a path has been substituted for one with the same value is the same as the projection before. Proof is by case analysis on the projection function.

Lemma 5.4.11 *If* $|path|_{h,s} = |path'|_{h,s}$ *then* $proj(h, s, \phi) = proj(h, s, \phi[path/path'])$

Lemma 5.4.12 captures the most basic requirement of soundness of effects - that the projection of effects covers all the objects read and written by execution. Notice that Lemma 5.4.12 uses the final heap h' in the projection of effects. This is because ω and ω' might include addresses created during execution. By combining Lemma 5.4.12 with other Lemmas we will get the corresponding result for h.

Lemma 5.4.12 *If*

670: $\pi, \gamma \vdash h, s$
671: $\pi, \gamma \vdash \varepsilon : \texttt{rd}\ \phi\ \texttt{wr}\ \phi'$
672: $\varepsilon, s, h \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h'$

then

673: $\omega \subseteq proj(h', s, \phi)$
674: $\omega' \subseteq proj(h', s, \phi)$

Proof: Proof is by induction on the derivation of 672 focusing on the last rule used. First notice from 670

675: $\pi \vdash \gamma$
676: $\vdash \pi$

Cases B-NULL, E-VAR and B-NEW are all trivial as in these cases $\omega = \omega' = \emptyset$.

Case: B-FLD

From B-FLD

677: $\varepsilon = z.f$
678: $s(z) = \iota$
679: $h(\iota)(f) = v$
680: $\omega = \{\iota\}$
681: $\omega' = \emptyset$
682: $h = h'$

Since 677 the rule used to derive 671 must be FX-FLD so

683: $\phi = z$
684: $\phi' = \texttt{emp}$

By 678, and 683 and the definition of *proj*

685: $proj(h', s, \phi) = \{\iota\}$

By 684 and the definition of $proj$

686: $proj(h', s, \phi') = \emptyset$

By 680, 685 we have 673. By 681 and 686 we have 674.

Case: B-ASS

From B-ASS

687: $\varepsilon = z.f{=}z'$
688: $s(\iota)$
689: $s(z') = v$
690: $h(\iota) = (\tau, fm)$
691: $h' = h[\iota \mapsto (\tau, fm[f \mapsto v])]$
692: $\omega = \omega' = \{\iota\}$

Since 687 the rule used in 671 must be FX-ASS thus

693: $\phi = \phi' = z$

Since 688, 693 and definition of $proj$

694: $proj(h', s, \phi) = proj(h', s, \phi') = \{\iota\}$

692 and 694 give 673 and 674.

Case: E-LET

From E-LET

695: $\varepsilon \ = \ \texttt{let}\ x = b\ \texttt{in}\ e'$
696: $b, s, h \overset{\omega_1;\omega_2}{\rightsquigarrow}_{\pi} v', h'''$
697: $x \notin domain(s)$
698: $s' = s[x \mapsto v']$
699: $e', s', h''' \overset{\omega_3;\omega_4}{\rightsquigarrow}_{\pi} v, h'$
700: $\omega = \omega_1 \cup \omega_3$
701: $\omega' = \omega_2 \cup \omega_4$

From 695 the rule used in 671 could be FX-LET or FX-LETFINAL, we shall perform a case analysis!

Case: FX-LET

Suppose rule used in 671 is FX-LET, then

702: $\pi, \gamma \vdash b : \texttt{rd}\ \phi_1\ \texttt{wr}\ \phi_2$
703: $\pi, \gamma \vdash b : t$
704: $x \notin domain(\gamma)$
705: $\gamma' = \gamma[x \mapsto t]$
706: $\pi, \gamma' \vdash e' : \texttt{rd}\ \phi_3\ \texttt{wr}\ \phi_4$

707: $\pi, \gamma \vdash \texttt{rd}\ \phi_1{+}\phi_3\ \texttt{wr}\ \phi_2{+}\phi_4 \leq \texttt{rd}\ \phi\ \texttt{wr}\ \phi'$
708: $\pi, \gamma \vdash \texttt{rd}\ \phi\ \texttt{wr}\ \phi'$

By 670, 696, 702 and induction

709: $\omega_1 \subseteq \mathit{proj}(h'', s, \phi_1)$
710: $\omega_2 \subseteq \mathit{proj}(h'', s, \phi_2)$

By 675, 703 and Lemma 5.4.5

711: $\pi, \gamma \vdash t$

so by 675, 711, 704 and TENV-VAR

712: $\pi \vdash \gamma'$

Now by 670, 696, 703 and L1

713: $\pi, \gamma \vdash s, h''$
714: $\pi, h'' \vdash v' : \mathit{map}(\pi, h'', s, t)$

and so by 712, 713, 697, 698, 714 and WF-CONF

715: $\pi, \gamma \vdash s', h''$

715, 706, 699 and induction gives us

716: $\omega_3 \subseteq \mathit{proj}(h', s', \phi_3)$
717: $\omega_4 \subseteq \mathit{proj}(h', s', \phi_4)$

Since 675, 702 and Lemma 5.4.6

718: $\pi, \gamma \vdash \texttt{rd}\ \phi_1\ \texttt{wr}\ \phi_2$

and thus ϕ_1 and ϕ_2 cannot contain any paths starting with x and so

719: $\mathit{proj}(h'', s', \phi_1) = \mathit{proj}(h'', s, \phi_1)$
720: $\mathit{proj}(h'', s', \phi_2) = \mathit{proj}(h'', s, \phi_2)$

Again, since ϕ_1, ϕ_2 do not mention x and since γ' contains γ, we trivially find

721: $\pi, \gamma' \vdash \texttt{rd}\ \phi_1\ \texttt{wr}\ \phi_2$

and by 699, 715, 721 and Lemma 5.4.10

722: $\mathit{proj}(h'', s', \phi_1) \subseteq \mathit{proj}(h', s', \phi_1)$
723: $\mathit{proj}(h'', s', \phi_2) \subseteq \mathit{proj}(h', s', \phi_3)$

By the definition of *proj*

724: $\mathit{proj}(h', s', \phi_1{+}\phi_3) = \mathit{proj}(h', s', \phi_1) \cup \mathit{proj}(h', s', \phi_3)$
725: $\mathit{proj}(h', s', \phi_2{+}\phi_4) = \mathit{proj}(h', s', \phi_2) \cup \mathit{proj}(h', s', \phi_4)$

Now, since 700, 709, 719, 722, 716 and 724

726: $\omega \subseteq \mathit{proj}(h', s', \phi_1+\phi_3)$

and similarly by 701, 710, 720, 723, 717 and 725

727: $\omega' \subseteq \mathit{proj}(h', s', \phi_2+\phi_4)$

As 707, 715 and Lemma 5.4.2

728: $\mathit{proj}(h', s', \phi_1+\phi_3) \subseteq \mathit{proj}(h', s', \phi)$
729: $\mathit{proj}(h', s', \phi_2+\phi_4) \subseteq \mathit{proj}(h', s', \phi')$

Since 708, ϕ and ϕ' cannot mention x and so

730: $\mathit{proj}(h', s, \phi) = \mathit{proj}(h', s', \phi)$
731: $\mathit{proj}(h', s, \phi') = \mathit{proj}(h', s', \phi')$

then from 726, 728, 730

732: $\omega \subseteq \mathit{proj}(h', s, \phi)$

and from 727, 729 and 731

733: $\omega' \subseteq \mathit{proj}(h', s, \phi')$

Case: FX-LETFINAL

From FX-LETFINAL

734: $\pi, \gamma \vdash b : \texttt{final}\ c\texttt{<}\overline{d}\texttt{>}$
735: $\pi, \gamma \vdash b : \texttt{rd}\ \phi_1\ \texttt{wr}\ \phi_2$
736: $x \notin \mathit{domain}(\gamma)$
737: $\gamma' = \gamma[x \mapsto c\texttt{<}\overline{d}\texttt{>}]$
738: $\pi, \gamma \vdash e' : \texttt{rd}\ \phi_3\ \texttt{wr}\ \phi_4$
739: $\texttt{rd}\ \phi\ \texttt{wr}\ \phi' = \texttt{rd}\ \phi_1+\phi_3[b/x]\ \texttt{wr}\ \phi_2+\phi_4[b/x]$

By 670, 696, 735 and induction

740: $\omega_1 \subseteq \mathit{proj}(h'', s, \phi_1)$
741: $\omega_2 \subseteq \mathit{proj}(h'', s, \phi_2)$

By 675, 734 and Lemma 5.4.5

742: $\pi, \gamma \vdash c\texttt{<}\overline{d}\texttt{>}$

so by 675, 742, 736 and TENV-VAR

743: $\pi \vdash \gamma'$

Now by 670, 696, 734 and L1

744: $\pi, \gamma \vdash s, h''$
745: $\pi, h'' \vdash v' : \mathit{map}(\pi, h'', s, t)$

and so by 743, 744, 745, 697, 698, and WF-CONF

746: $\pi, \gamma \vdash s', h''$

746, 738, 699 and induction gives us

747: $\omega_3 \subseteq \mathit{proj}(h', s', \phi_3)$
748: $\omega_4 \subseteq \mathit{proj}(h', s', \phi_4)$

Since 675, 735 and Lemma 5.4.6

749: $\pi, \gamma \vdash \texttt{rd}\ \phi_1\ \texttt{wr}\ \phi_2$

and so ϕ_1 and ϕ_2 can contain no paths beginning with x and so

750: $\mathit{proj}(h'', s', \phi_1) = \mathit{proj}(h', s, \phi_1)$
751: $\mathit{proj}(h'', s', \phi_2) = \mathit{proj}(h', s, \phi_2)$

Trivially from 749

752: $\pi, \gamma' \vdash \texttt{rd}\ \phi_1\ \texttt{wr}\ \phi_2$

so by 699, 746, 752 and Lemma 5.4.10

753: $\mathit{proj}(h'', s', \phi_1) \subseteq \mathit{proj}(h', s', \phi_1)$
754: $\mathit{proj}(h'', s', \phi_2) \subseteq \mathit{proj}(h', s', \phi_2)$

So combining, 740, 750 and 753 and also 741, 751 and 754

755: $\omega_1 \subseteq \mathit{proj}(h', s, \phi_1)$
756: $\omega_2 \subseteq \mathit{proj}(h', s, \phi_2)$

Since the type of b is `final`, b must be a path of the form $z.f$ and by Lemma 5.4.8 and 696

757: $\mathit{map}(\pi, h, s, b) = v'$

Because b is final it's value does not change after execution and by 670, 671 and Lemma 5.4.18, 672, 734 and Lemma 5.4.7

758: $\mathit{map}(\pi, h, s, b) = \mathit{map}(\pi, h', s, b)$

Since b has a type w.r.t. γ it cannot mention x and so

759: $\mathit{map}(\pi, h', s', b) = \mathit{map}(\pi, h', s, b)$

Notice, by construction of s'

760: $\mathit{map}(\pi, h', s', x) = v'$

So by 757, 758, 759 and 760

761: $\mathit{map}(\pi, h', s', x) = \mathit{proj}(h', s', b)$

and by 761 and Lemma 5.4.11

762: $\mathit{proj}(h', s', \phi_3) = \mathit{proj}(h', s', \phi_3[b/x])$
763: $\mathit{proj}(h', s', \phi_4) = \mathit{proj}(h', s', \phi_4[b/x])$

and since the substitution on ϕ_3 and ϕ_4 removes all usage of x

764: $\mathit{proj}(h', s', \phi_3[b/x]) = \mathit{proj}(h', s, \phi_3[b/x])$
765: $\mathit{proj}(h', s', \phi_4[b/x]) = \mathit{proj}(h', s, \phi_4[b/x])$

By 747, 762 and 764

766: $\omega_3 \subseteq \mathit{proj}(h', s, \phi_3[b/x])$

and similarly, since 748, 763 and 765

767: $\omega_4 \subseteq \mathit{proj}(h', s, \phi_4[b/x])$

By the definition of *proj* and 739

768: $\mathit{proj}(h', s, \phi) = \mathit{proj}(h', s, \phi_1) \cap \mathit{proj}(h', s, \phi_3[b/x])$
769: $\mathit{proj}(h', s, \phi') = \mathit{proj}(h', s, \phi_2) \cap \mathit{proj}(h', s, \phi_4[b/x])$

So finally by 768, 755, 766 and 700 we have

770: $\omega \subseteq \mathit{proj}(h', s, \phi)$

and similar by 769, 756, 767 and 701 we have

771: $\omega' \subseteq \mathit{proj}(h', s, \phi')$

Case: B-CALL

We omit this case as it is mostly concerned with checking that the effect of the dynamically chosen method body match those found by the lookup in the effect rule. The details are long and tedious. The case ends with a straightforward induction.

□

Before proving LS4 we introduce a few more helpful Lemmas.

Lemma 5.4.13 states that for all executions the runtime read-effect contains the runtime write-effect. This is an easy induction.

Lemma 5.4.13 *If* $\varepsilon, s, h \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h'$ *then* $\omega' \subseteq \omega$

Lemma 5.4.14 is the static analogue of 5.4.13 stating that in a well-formed environment the read-effect calculated for an expression contains the write-effect. Proof is by induction on the derivation of the effects and appeals to well-formedness of the program.

Lemma 5.4.14 *If*

772: $\pi \vdash \gamma$
773: $\pi, \gamma \vdash \varepsilon : \texttt{rd}\ \phi\ \texttt{wr}\ \phi'$

then

774: $\pi, \gamma \vdash \phi' \leq \phi$

In Lemma 5.4.15 we capture soundness of the runtime write-effect - that it includes all objects that have changed during execution. The equivalent result for the runtime read-effect is embodied in Lemma 5.2.7. The proof is by induction on the derivation of the execution and is very straightforward.

Lemma 5.4.15 *If*

775: $\varepsilon, s, h \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h'$
776: $h(\iota) \neq h'(\iota)$
777: $\iota \in domain(h)$

then

778: $\iota \in \omega'$

Lemma 5.4.16 follows from Lemma 5.4.12.

Lemma 5.4.16 *If*

779: $\pi, \gamma \vdash h, s$
780: $\pi, \gamma \vdash \varepsilon : \texttt{rd}\ \phi\ \texttt{wr}\ \phi'$
781: $\varepsilon, s, h \overset{\omega,\omega'}{\rightsquigarrow}_{\pi} v, h'$

then

782: $\omega \cap domain(h) \subseteq proj(h, s, \phi)$
783: $\omega' \cap domain(h) \subseteq proj(h, s, \phi')$

Lemma 5.4.17 *If*

784: $\pi, \gamma \vdash s, h$
785: $\pi, \gamma \vdash \varepsilon : \texttt{rd}\ \phi\ \texttt{wr}\ \phi'$
786: $\varepsilon, s, h \overset{\omega,\omega''}{\rightsquigarrow}_{\pi} v, h''$

then there exist h_1, h_2, h_3, h_4, h_2' *such that*

787: $proj(h, s, \phi) = h_1 * h_2$
788: $proj(h, s, \phi') = h_2$
789: $h = h_1 * h_2 * h_3$
790: $h'' = h_1 * h_2' * h_3 * h_4$

791: $\varepsilon, s, h_1 * h_2 \overset{\omega,\omega''}{\rightsquigarrow}_\pi v, h_1 * h_2' * h_3$
792: $h_2 \# h_4$
793: $h_1 * h_2' \sqsubseteq proj(h'', s, \phi)$
794: $h_2' \sqsubseteq proj(h'', s, \phi')$

Proof: From 784

795: $\pi \vdash \gamma$
796: $\vdash \pi$

By 784, 785, 786 and Lemma 5.4.16

797: $\omega \cap domain(h) \subseteq proj(h, s, \phi)$
798: $\omega' \cap domain(h) \subseteq proj(h, s, \phi')$

By 784, 785, 786 and Lemma 5.4.12

799: $\omega \subseteq proj(h, s, \phi)$
800: $\omega' \subseteq proj(h, s, \phi')$

By 786 and Lemma 5.4.13

801: $\omega' \subseteq \omega$

From 795, 785 and Lemma 5.4.14 we get

802: $\pi, \gamma \vdash \phi' \sqsubseteq \phi$

and so by 784, 802 and Lemma 5.4.2

803: $proj(h, s, \phi') \subseteq proj(h, s, \phi')$

So let

804: $h' = h \downarrow_{\omega \cap domain(h)}$
805: $h_2 = proj(h, s, \phi') = h \downarrow_{proj(h,s,\phi')}$
806: $h_1 = h \downarrow_{proj(h,s,\phi)/proj(h,s,\phi')}$
807: $h_3 = h \downarrow_{domain(h)/proj(h,s,\phi)}$

Notice by 805, 806

808: $h_1 \# h_2$

and that, with 803, 805, 806,807

809: $h_1 * h_2 = proj(h, s, \phi)$
810: $h = h_1 * h_2 * h_3$

By Lemma 5.4.18, 795, 785

811: $\pi, \gamma \vdash \varepsilon : t$

then by 784, 811, 786 and L1

812: $\pi, \gamma \vdash h'', s$

By 784, 785, 802 and Lemma 5.4.10

813: $proj(h, s, \phi') \subseteq proj(h', s, \phi')$
814: $proj(h, s, \phi) \subseteq proj(h', s, \phi)$

Let

815: $h_2' = h' \downarrow_{domain(h_2)}$
816: $h_2'' = proj(h'', s, \phi')h'' \downarrow_{proj(h'',s,\phi')}$
817: $h_1'' = h'' \downarrow_{proj(h'',s,\phi)/proj(h'',s,\phi')}$

Notice (similarly to 807) that

818: $h_1'' \# h_2''$
819: $h_1'' * h_2'' = proj(h'', s, \phi)$

Since 798 and 806

820: $domain(h_1) \cap \omega' = \emptyset$

so by 786, 820 and Lemma 5.4.15

821: $\forall \iota \in domain(h_1) : \quad h(\iota) = h'(\iota)$

By 785, 795 and Lemma 5.4.6

822: $\pi, \gamma \vdash \texttt{rd}\ \phi\ \texttt{wr}\ \phi'$

and by 784, 786, 785, 822 and Lemma 5.4.9

823: $proj(h'', s, \phi) \cap domain(h) = proj(h, s, s)\phi$
824: $proj(h'', s, \phi') \cap domain(h) = proj(h, s, s)\phi'$

Thus by 806, 823 and 824 we find

825: $domain(h_1) = (proj(h'', s, \phi)/proj(h'', s, \phi')) \cap domain(h)$

and by construction of h_1'', h_1, 821

826: $h_1 \sqsubseteq h_1''$

By construction

827: $domain(h_2) \subseteq proj(h, s, \phi')$

and so with 813

828: $domain(h_2) \subseteq proj(h'', s, \phi')$

and then 815 and 816 give

829: $h'_2 \sqsubseteq h''_2$

From 807 and 815

830: $h_1 \# h'_2$

and so by 826, 829 and 830

831: $h_1 * h'_2 \sqsubseteq h''_1 * h''_2$

i.e.

832: $h_1 * h'_2 \sqsubseteq proj(h'', s, \phi)$
833: $h'_2 \sqsubseteq proj(h'', s, \phi')$

By Corollary 5.2.5

834: $domain(h)h''$

so let

835: $h_4 = h'' \downarrow_{domain(h'')/domain(h)}$

By construction of h_4

836: $h_4 \# h$
837: $h_4 \# h_2$ (836, 810)
838: $domain(h'') = domain(h) \cup domain(h_4)$

Notice that by construction (805, 807)

839: $domain(h_3) \cap domain(h_2) = \emptyset$

and thus, since 798

840: $domain(h_3) \cap \omega' = \emptyset$

Because of 840, with 786 and Lemma 5.4.15

841: $\forall \iota \in domain(h_3): \quad h(\iota) = h''(\iota)$

so, with 834

842: $h_3 \sqsubseteq h''$

By 826 and 817 and 817 and by 829 and 816 we have

843: $h_1 \sqsubseteq h''$
844: $h'_2 \sqsubseteq h''$

By 815 and 810

845: $domain(h_1 * h'_2 * h_3) = domain(h)$

and with 835

846: $domain(h_1 * h'_2 * h_3 * h_4) = domain(h'')$

so by 843, 844, 842, 835 and 846

847: $h'' = h_1 * h'_2 * h_3 * h_4$

Now, by Lemma 5.2.7 and 786 and 804 there exist h''' and h_r such that

848: $\varepsilon, s, h' \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h'''$
849: $h = h' * h_r$
850: $h'' = h''' * h_r$

Since 804, 797 and 809

851: $h' \sqsubseteq h_1 * h_2$

By SH4 there exist h_1''' and h_2''' s.t.

852: $h_1''' \sqsubseteq h_1$
853: $h_2''' \sqsubseteq h_2$
854: $h' = h_1''' * h_2'''$

and by definition of $\sqsubseteq$ we must also have h_1'''' and h_2'''' such that

855: $h_1 = h_1''' * h_1''''$
856: $h_2 = h_2''' * h_2''''$

By 854, 855, 856 and 810

857: $h = h' * h_1'''' * h_2'''' * h_3$

and by 849, 857 and Lemma 3.1.3

858: $h_r = h_1'''' * h_2'''' * h_3$

combined with 850, 847 and 858 we see that

859: $h''' * h_1'''' * h_2'''' * h_3 = h_1 * h_2' * h_3 * h_4$

further, by 855

860: $h''' * h_1'''' * h_2'''' * h_3 = h_1''' * h_1'''' * h_2' * h_3 * h_4$

and so by 860 and Lemma 3.1.3

861: $h''' * h_2'''' = h_1''' * h_2' * h_4$

By 856, 855 and 857

862: $h_2''''\#h_1''' * h_4$

and with 862, 861 and Lemma 3.1.3

863: $h_2'''' \sqsubseteq h_2'$

so let

864: $h_2' = h_2'''' * h_2'''''$

So, by 861 and 864

865: $h''' * h_2'''' = h_1''' * h_2'''' * h_2''''' * h_4$

and by Lemma 3.1.3

866: $h''' = h_1''' * h_2''''' * h_4$

By 865 and ...

867: $h'''\#h_1'''' * h_2''''$

then by Lemma 5.3.1, 848 and 867

868: $\varepsilon, s, h' * h_1'''' * h_2'''' \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h''' * h_1''''h_2''''$

so by 854, 866, 855, 856, 864

869: $\varepsilon, s, h_1 * h_2 \overset{\omega,\omega'}{\rightsquigarrow}_\pi v, h_1 * h_2'h_4$

We have 805, 809, 810, 847, 869, 836, 832, 833 so we are finished! □

5.4.5 LS5

Lemma 5.4.18 *If* $\pi,\gamma \vdash \varepsilon$:`rd` ϕ `wr` ϕ' *then there exists a* t *such that* $\pi, \vdash <\varepsilon> : t$.

The omitted proof is by induction on the derivation of $\pi,\gamma \vdash \varepsilon$:`rd` ϕ `wr` ϕ' . As noted in Section 4.6.2.9 the rule FX-LET includes some type checks which are not necessary in order to calculate effects but are required for this result. We do not need similar checks in FX-LETFINAL as induction gives a well-formed type in the larger environment and substitution with the final path will always give a well-formed type in the initial environment.

Theorem 5.4.1

$$LE_{\text{ODE}} = (L_{\text{ODE}}\ ,\ \phi\ ,\ proj(\_,\_,\_)\ ,\ \_,\_\vdash\_\#\_\ ,\ \_,\_\vdash\_:\texttt{rd}\ \_\ \texttt{wr}\ \_\ ,\ \_)$$

is a Language With Effects

Proof: Theorem 5.4.1 follows from Theorem 5.2.1 and Lemmas 5.4.1, 5.4.4, 5.4.17 and 5.4.18 □

5.5 Experience with the Model

Having proved a reasonably large formal system to be an instance of the model we make a number of observations regarding possible improvements to the model.

5.5.1 L4

As we have noted in Section 5.2.8 proving ODE conformed to **L4** proved a challenge. The proofs as they stand are relatively straightforward although the technique we developed was orthogonal to the rest of ODE. We think that the approach we have used for ODE should be applicable in most languages with pointers but proofs would still be fairly involved. An alternative to **L4** which could be proved in a more straightforward fashion is certainly desirable.

5.5.2 Complete Heaps

Complete heaps, $\mathsf{s} \models \mathsf{h}$ were originally intended to have a more explicit meaning. An earlier version of the model gave a semantics to complete heaps whereby for any complete heap, joining it with another heap did not allow any further expressions to run. This certainly tallies with the ODE definition since, as there are no dangling pointers, adding another heap does not change the fields that can be accessed. Some refinement of the model meant that this specification was no longer necessary and it was abandoned though complete heaps remained. Eliminating them and replacing $\mathsf{s} \models \mathsf{h}$ with $\gamma \vdash \mathsf{s}, \mathsf{h}$ in **L4** seems like a possible simplification of the model.

5.5.3 Stacks and Heaps

On reflection, stacks and heaps, whilst convenient for ODE, could be amalgamated in to a single *store* in the model. Particularly if we were to do away with complete heaps then heaps and stacks always appear together and could be amalgamated. Definitions for disjointness and joining of heaps might become a little more complicated. In ODE we could model stores as pairs of heaps and stacks and only allow joining of two stores with identical stacks, for example. We note that, as for environments in ODE, such a simplification in the model may necessitate a little refactoring of the language.

5.5.4 Local Languages

We have discussed the repercussions of **LL1** in Section 3.6.5. **LL2** causes no such problems and in fact is useful in proving other properties such as **L3** and **LS4**. Since **LL2** seems linked to **L3** and also since **L2** follows as a consequence of **LL2** we think the model should probably abandon the separation of the Local Languages definitions and integrate them into the definition for Programming Languages with **LL2** replacing **L2**.

Chapter 6

Related Work

We now discuss related work in the areas of effects, ownership and encapsulation, language independent abstractions and reasoning in the presence of shared mutable state.

6.1 Effects Systems

Effects systems have their origins in the late eighties with the work of Lucassen and Gifford [LG88] on the FX language [GJLS87]. The aim was to calculate scheduling constraints for expressions and thus assist a compiler for parallel computers. FX is a functional language with imperative features. Effects are given in terms of regions which group together store locations. The system records read and write effects but also *allocation* effects stating if and in what regions an expression allocates new store locations. A subsequent system [TJ92] provides inference of types, regions and effects. The effects systems described in [TJ92] and [LG88] appear to fit our model. They do not require the read effect to contain the write effect but this would be an easy restriction of the system.

The discipline established by [LG88] and [TJ92] is greatly expanded in [ANN99]. Effects are developed into *behaviours* which include a temporal element, thus statically providing a trace of the actions of the program. The formal system presented explicitly considers concurrency and a case study shows how the system can help prove safety properties of a program.

More recently, the object oriented community has considereD effects. Leino [Lei98] and Boyland and Greenhouse [GB99] presented similar systems that divide an object into a number of *datagroups* or *regions* (we shall use datagroups from now on). Special 'pivot' fields allow datagroups of one object to represent other objects. This allows, for example, the internal representation (as an array or linked nodes) of a list to be described by a datagroup of the list object. This provides encapsulation, which mirrors ownership in `Joe` and ODE, and allows simple effects to be given to programs. The system of Greenhouse and Boyland is an effects system proper (as we consider them) including read and write effects whilst datagroups do not record read effects. Datagroups are intended to track the modifications made by an execution in order to aid program analysis and verification [LPHZ02]. The interest only in modification here mirrors the fact that we are only concerned with write effects for determining non-interference of predicates. We believe the Boyland-Greenhouse system is an instance of the model of effects. Soundness results for a reduced system have been given by Parkinson and Bierman [BPP03, BP03] and they have made some progress towards inference of effects. This system is particularly interesting in the context of our model as heap joining and disjointness will involve fragments of objects rather than whole objects, as in `Joe` and ODE. Datagroups describe sets of fields of an object so, an effect might describe a sub heap consisting only of incomplete objects. Heap disjointness would consider not only the address of an object in the heap but also its fields. Two heaps containing the same address but with the objects in each defining a disjoint set of fields are disjoint.

Boyland and Retert [BR05] present a system where effects are represented via permissions to access fields of particular objects. If an expression does not have permission for a piece of state it cannot read or write it. Thus, the permissions an expression *does* have gives an approximation of the state that will be used at runtime. Boyland and Retert show a property related to our non-interference where, for two expressions with compatible permissions i.e. they do not share permissions to the same state, when executed sequentially there is no state accessed by both expressions. This system does not distinguish read and write effects (this remains further work) so the result is a little coarser than in our model. The system is intended as a low-level, complex and general approach to representation of effects. They suggest that effects of the type in [GB99] may be supported as an abstraction and translated into the permissions system. However, in more recent work, [Boy03], Boyland considers *fractional permissions* which allow distinction between reading and writing. He goes on to reject effects in favour of permissions. He notes that although judgement of non-interference appears similar in both disciplines, the differences between disjointness of effects and splitting of permissions between expressions make comparison impossible. Further, in the presence of effects systems like [GB99], detecting aliasing is hard[1]. Boyland presents a non-interference result in a concurrent setting. He shows that, for an expression with valid permissions (which may spawn concurrent threads), any ordering of execution gives an equivalent heap.

Other systems track computational effect to achieve ends other than determination of non-interference. The Fickle language [DDDCG01] allows objects to change class at runtime. The type system includes an 'effect' consisting of the set of classes which may have been reclassified. This effect is necessary for soundness of the type system to prevent invalid access to an object that has changed class. Some type systems e.g. [Wri92] record only the effects of allocation and utilise the information to assist in the generation of types.

[1]In comparison, the type systems of `Joe` and ODE give greater support.

6.2 Encapsulation and Ownership

The background on encapsulation systems is extensive and well-documented. We focus on current developments especially with respect to their possible use in effects systems.

Recent works on ownership types have sought to increase flexibility compared to the original systems [CNP01, CPN98]. Lu and Potter [LP06b, LP06a] have introduced two systems which place objects in a standard ownership tree structure but allow a wider variety of references between objects. Both systems introduce capabilities to the types of programs. In [LP06b] an annotation on a method restricts which objects may call it. This restricts side-effecting executions as only objects with sufficient capability can call methods that might tamper with another object's internal state. In [LP06a] field declarations are augmented with an *accessor* as well as an owner. The owner determines the object's position in the ownership hierarchy but the accessor determines which objects may refer to it. This allows some objects e.g. iterators to be accessible by external clients but still having access to the list representation by virtue of sharing an owner. In both of these systems wildcards can be introduced for ownership contexts thus obscuring some details of an object's position in the ownership hierarchy.

[LP06a] gives a similar level of flexibility to Ownership Domains although without requiring final fields to express idioms such as iterators. However the variance and wild cards allowed in types suggest that it will, in some cases, prove hard to give an accurate effect for expressions.

Simple Loose Ownership Domains (SLOD) [SPH06] present a simplified but also extended version of Ownership Domains. SLOD forces each object to have only two domains, one private and one public, with the public domain implicitly granted permission to access the contents of the private domain. Because there is complete knowledge of the domain structure of every object, SLOD is able to offer an alternative to our `siblings` construct. The syntax `owner.local` in a type declaration describes the local domain of the object that owns the current object's owner domain. Used in the iterator example, this removes the need to include the parameter domain that gives the owner of the list nodes. This avenue was not open to us as we allowed arbitrary numbers of domains to be declared and so could not name the owner of list nodes without a parameter. The use of `owner.local` implicitly gives the siblings relation and so would allow for calculation of effects for iterators without the sibling constraint or sibling effect. Despite the differences in approach we feel SLOD validates the need for the sibling effect as it promotes the relationship between public and private domains as an idiom of Ownership Domains. Another feature of SLOD allows for imprecise expressions as domain parameters, allowing a type to describe objects in multiple domains. Much like in [LP06a] this increases flexibility but will make calculation of effects harder.

Generic Ownership [PNCB06] provides a version of Ownership Types which sits atop the generic types of Java [BOSW98]. The system does not offer any refinement over original ownership types in terms of the encapsulation offered. However, it may offer an opportunity to calculate more accurate effects than were possible in `Joe`. By mixing type and ownership information it should be possible to describe effects not only in terms of ownership but in terms of types. The description of the heap in an effect becomes more precise as the ownership based effect is restricted by type information.

6.3 Language Independent Abstraction

In Chapter 3 we presented a language independent abstraction of the features of a programming language with effects. Work in other fields have made similar abstractions, giving abstract descriptions of languages and features and deriving meta-level results.

In [Fel91] Felleisen takes a very abstract view of programming languages in order to compare their expressiveness. He defines a language as consisting of phrases, (programs are a a recursive subset of phrases) and a predicate defining the semantics of the language which is satisfied if the program terminates. Felleisen proceeds to consider transformations between languages and contextual equivalence of phrases. His main results gives conditions whereby one language cannot express the programs of another in a structure preserving way.

The closest abstraction to ours in terms of style is the Fragment Calculus which was a source of inspiration, though the area of application is very different. The Fragment Calculus [DEW99] describes the interaction of 'fragments', arbitrary units of a source language or program binaries which participate in compilation of a program. Fragments may be updated with other fragments and compiled into programs. The fragment calculus defines systems as tuples of sets, relations and functions just as in our model. Properties of the functions and relations are given as axioms of the system. Results on binary compatibility follow from the definitions of the relations and functions.

In [CC89] Cousot and Cousot give a language independent proof of the soundness of Lamport's Generalized Hoare Logic. Their abstraction, though still not describing any particular language makes greater assumptions about the structure of the language. A function that decomposes a program fragment into its sub-fragments is used and kinds of program fragments are distinguished. An operational semantics is assumed and there is a detailed treatment of the flow of control during execution. This abstraction seems closest to ours in intent, as it considers the details of executions rather than just termination as in [Fel91]. It does, however, make many more assumptions about the details of the language. We are unsure exactly how different languages could be whilst still fitting this abstraction.

6.4 Reasoning in the presence of Aliasing and Mutable State

As Reynolds comments in [Rey02a] the problem of reasoning in the presence of shared mutable state has been an open problem in computer science for over thirty years. Modern approaches seem to fall into two camps: languages with support from types which provide encapsulation and restrict aliasing and that of separation logic. Both approaches are very much concerned with frame conditions and locality of reasoning.

Our interest in effects allies us more strongly with systems supported by types and encapsulation. The ownership system Universes [MPH99] has been used to encapsulate the abstract state of objects to support verification [Mül01, DM05]. Such systems include *modifies* clauses which act as write effects for methods. Consideration of the modifies clause of a method and encapsulation is closely integrated into the verification process rather than being decoupled as we suggest with effects. Universes offer a less precise form of ownership than ownership types or domains. Read-only references are allowed to arbitrary parts of the heap but side-effects must be confined within ownership boundaries. Read-only references lose information as there is no record of where in the heap the referenced objects are. This would make an effects system with useful read and write effects impossible but consideration of just the write effect should be possible. This is, perhaps, not surprising as the intent of the system is to limit the scope of modifications to assist in verification.

Separation Logic [Rey02a] provides an alternative approach to reasoning where the assertions of the Hoare logic have not only a logical but also a *spatial* element. The spatial conjunction $p * q$ states that both p and q hold, but in disjoint parts of the heap. Normal conjunction $p \wedge q$ states that p and q both hold, but satisfaction of each is determined by *exactly* the same part of the heap. The building blocks of spatial assertions are of the form $(e \mapsto e')$ stating that the heap location referred to by e holds the value referred to by e'. Such an assertion is satisfied by exactly one heap cell. Thus $x \mapsto 12$, the assertion that the value held in the address referred to by x is 12, is satisfied by a heap containing exactly one cell, with address x that holds a value 12. A heap containing more than one cell cannot satisfy this assertion. By use of the spatial conjunction, specifications that refer to larger and more realistic heaps can be built.[2]

The small axioms of separation logic give Hoare triples for assignments and (de)allocation of mememory. For example, assignment to a pointer allows derivation of the following triple:

$$\{E \mapsto \_\}[E] = E'\{E \mapsto E'\}$$

The command $[E] = E'$ assigns the value of E' to the address referred to by E. The precondition is that address E exists, though we do not care what value it holds. The postcondition states that E now holds the value of E'. Both the pre- and postcondition are satisfied by a heap of only one cell, with address E.

The small axioms appear to preclude any useful reasoning as the specification can only refer to precisely the state modified by an assignment. The problem is solved by the frame rule:

$$\frac{\{p\}c\{q\}}{\{p * r\}c\{q * r\}}$$

[2]Separation Logic also includes the assertion *true* that is satisfied by any heap.

The frame rule allows any predicate (which will not be altered by c) to be spatially conjoined to the pre- and post-conditions of any derivable Hoare triple. By combining the frame rule with the small axioms, more complicated specifications can be proved. Spatially conjoined predicates are propogated through the proof of a specification. When an assignment is encountered the frame rule is used to strip off the 'excess' specification, a small axiom is used and then the 'excess' can be spatially conjoined with the postcondition dervied by the small axiom.. The similarity to the Rule of Constancy should be obvious and Reynolds recognizes the frame rule as a successor to the Rule of Constancy [Rey02b] which more directly addresses issues of aliasing, sharing of state and interference.

The key difference between separation logic and reasoning with non-spatial specifications is that, in separation logic, issues of aliasing are dealt with automatically. Specifications in separation logic necessarily contain information on sharing of state and aliasing, by use of the spatial (and non-spatial) conjunction. Consider the frame rule: by assumption p and r cannot depend on the same state otherwise they would not be disjoint (and thus could not be joined with $*$). Because of the requirements of the small axioms, in any Hoare triple the pre- and post-conditions must describe all of the heap used by the expression. Thus any predicate conjoined to a specification with the frame rule cannot possibly be affected by execution - if it were it would have to refer to the same part of the heap as the initial specification and then could not be disjoint. In contrast, the rule of constancy must appeal to effects to check that the new conjunct is not affected by the command being executed.

The similarities between the approach of separation logic and our use of effects for predicates are clear but the differences are also great. We view effects as a lightweight way to address spatial concerns in non-spatial logics. Sepearation logic is heavier, as all spatial concerns are explicitly encoded. Separation logic is, however, more generally applicable as our approach relies on the precision of the effects system to yield useful results. Nonetheless, the details of the systems have much in common; both are concerned with disjointness and joining of heaps and both systems require some similar language properties - **LL2** in the model is very similar to Heap Locality property required by Separation Logic.

Separation Logic and our approach also differ over what specifications it is possible to prove for a given program. As already described, separation logic specifications *must* mention all the state that is used by an execution and thus all the behaviour of a system. It is easy to conceive of a situation where this is not desirable. Within a single program fragment, there may exist code which address orthogonal concerns. Consider the sequence `doSomething(); log()` where the first function call does something 'important' i.e. contributing to the desired behaviour of the program. The call to `log()` performs some ancillary function e.g. logging for performance profiling. Note that `log()` does not contribute to the desired behaviour of the program and as such we might not want to describe its action in the specification. By not including the behaviour of `log()` in our specification we can reduce the complexity of the specification and thus, also the proof burden. If `log()` does not interfere with the post-condition of `doSomething()` we should be able to ignore the behaviour of `log()` with respect to the specification of `doSomething()`. We shall assume this to be the case.

Suppose we want to verify `doSomething(); log()` in terms of the behaviour of `doSomething()` for example:

$$\{I\}\texttt{doSomething();log()}\{I'\}$$

where I and I' describe the behaviour of `doSomething()` but do not mention the behaviour of `log()`. From our previous assumption of non-interference between `log()` and I' we expect that if we can prove:

$$\{I\}\texttt{doSomething();}\{I'\}$$

then our specification of `doSomething(); log()` will also hold. Indeed, supposing we are able to deduce with effects that the write effect of `log()` is disjoint from the effect of I'. Then using our Rule of Constancy:

$$\dfrac{\dfrac{\ldots}{\{I\}\texttt{doSomething()}\{I'\}} \qquad \dfrac{\dfrac{\{\mathit{true}\}\texttt{log()}\{\mathit{true}\} \quad I' : \phi \quad \texttt{log()} : \phi'\phi'' \quad \phi\#\phi''}{\{\mathit{true} \wedge I'\}\texttt{log()}\{\mathit{true} \wedge I'\}}}{\{I'\}\texttt{log()}\{I'\}}}{\{I\}\texttt{doSomething(); log()}\{I'\}}$$

If we try to prove the same property (upto translation) using separation logic, we run into problems. For both `doSomething()` and `log()` there is some minimum provable specification (this is implied by the small axioms). Let's say:

$$\{I_S\}\texttt{doSomething()}\{I'_S\}$$
$$\{L_S\}\texttt{log()}\{L'_S\}$$

Now, when we try to prove a specification for `doSomething(); log()` we find that we can only prove e.g.

$$\dfrac{\dfrac{\dfrac{\ldots}{\{I_S\}\texttt{doSomething()}\{I'_S\}}}{\{I_S * L_S\}\texttt{doSomething()}\{I'_S * L_S\}} \quad \dfrac{\dfrac{\ldots}{\{L_S\}\texttt{log()}\{L'_S\}}}{\{I'_S * L_S\}\texttt{log()}\{I'_S * L'_S\}}}{\{I_S * L_S\}\texttt{doSomething(); log()}\{I'_S * L'_S\}}$$

or

$$\dfrac{\dfrac{\dfrac{\ldots}{\{I_S\}\texttt{doSomething()}\{I''_S * I'''_S\}}}{\{I_S * L''_S\}\texttt{doSomething()}\{I''_S * I'''_S * L''_S\}} \quad \dfrac{\dfrac{\ldots}{\{L''_S * I'''_S\}\texttt{log()}\{L'_S\}}}{\{I''_S * I''''_S * L''_S\}\texttt{log()}\{I''_S * L'_S\}}}{\{I_S * L''_S\}\texttt{doSomething(); log()}\{I''_S * L'_S\}}$$

where $I'_S = I''_S * I'''_S$, $L_S = L''_S * I'''_S$ and $L''_S = L''_S$.

The first proof works in situations where the specifications of `doSomething()` and `log()` describe completely separate parts of the heap. The second case allows for the possiblity that some part of the heap is mentioned by both. Note that this is possible even under our assumption of non-interference as, if both expressions read some part of the heap then, by the small axioms, it must be mentioned in the specification. The rule of constancy covers both possiblities.

In Separation Logic we are unable to ignore the behaviour of `log()`, in fact we must verify it in its entirety if we are to reason at all about a program where `log()` is used[3]. The rule of constancy allowed us to ignore the behaviour of `log()` all together, we do not even need to know of a specification for it. Separation logic obliges us to derive a specification for `log()` which, because of the small axioms, must describe its behaviour in all detail. It appears that separation logic rules out potentially useful idioms in verification that can be allowed with effects and the Rule of Constancy.

Although we have outlined some disadvantages, Separation Logic is a very powerful and emerging technology. Above, we showed that the need to specify the entire behaviour of the program to be a weakness but it also contributes to the strength of separation logic. It is the spatial information and the requirements of the small axioms that make complicated reasoning in the presence of aliasing and shared state possible. Proofs for non-trivial pointer programs have been produced [ORY01]. The approach has been adapted to object oriented languages [Par05] enabled by 'abstract predicates'[PB05] which support modular reasoning. Abstract predicates allow visiblity of specifications to be constrained. To the client of a module or class, a method specification is given in terms of 'atomic predicates' e.g. $\mathit{listEmpty}(l)$. These are given a precise definition in separation logic, but this is not visible to the client. From the client side, reasoning is performed in terms of these predicates, which can be spatially conjoined with other predicates and used to satisfy the preconditions of other method calls. The module is verified in terms of the precise specification of the predicates and thus modularity is acheived. Despite the ability of abstract predicates to hide the details of a module's specification, they do not remove the need for the specification to cover all the behaviours of the module. The internal specification of a module is still verified by use of the frame rule and small axioms and thus will still have mention all the state that is used. Work is progressing on a treatment of concurrency [BCOP05] and there is a relationship with the permissions based work of Boyland [Boy03].

[3]We are not allowed to weaken on $*$ as we would with $\wedge$ due to the semantics of predicate satisfaction. L' could be weakened to e.g. true in the postcondition but the precondition will still have to mention L.

Chapter 7

Conclusions

In this thesis, we have developed a language independent model which describes effects for both predicates and expressions. The model provides a set of requirements for programming languages, predicates and effects which enable judgement of non-interference properties. The application of effects to predicates is novel, and we show how calculation of effects for predicates might work. Our treatment of effects for programming languages is intended to be compatible with the properties of extant effects systems, as well as providing a framework for the development of new systems. The paper 'Multiple Ownership'[CDNS07] (to appear at OOPSLA 2007) has used the approach of this thesis to show soundness and non-interference properties of a novel ownership-based language with effects.

We have developed ODE a version of Ownership Domains with an effect system based on `Joe`. We have shown that the effects of ODE can give greater precision than `Joe`. By proving that ODE is an instance of our model of effects we not only show that ODE can detect non-interference properties but also give some some assurance of the validity of the model.

There are a number of areas for improvement. Our model is a first attempt and as such does not deal with more complicated features of programming languages such as concurrency.

As mentioned in Section 3.6.5 the model's treatment of predicates precludes some desirable idioms such as quantification. This is probably the main weakness of the model. However as explained in Section 3.6.5, we still expect that the non-interference property stated in Theorem 3.5.1 can hold for languages and effects systems which do not fit the model. Whether we have simply not yet found the right abstraction or whether trying to separate the properties of predicates and effects is not possible, remains an open question.

The part of the model concerned with non-interference of expressions is more successful though not unconditionally so. Developing ODE to be an instance of the system required some unimportant but annoying modifications to be made. The non-compliance of the projection function for effects (since it requires the program) suggests either a modification to the model or to the presentation of ODE. Further experience with other languages might suggest which is more appropriate. In order for ODE to satisfy LS5, some of the effects rules needed, otherwise unnecessary, strengthening. This highlights the more liberal nature of effects compared to types. In a system such as ODE even mal-formed expressions can be given a valid effect such as `world.under`. This is not a major concern although it does suggest improvements can be made so that the model fits more 'natural' language definitions.

Proving that ODE satisfies L4 turned out to be a more significant challenge than we had anticipated but in the process developed a style of proof we believe could be generally applicable. L4 was a late introduction to the model after we realised a counterexample while attempting to prove Theorem 3.5.2. As such we may have over-egged the pudding somewhat, although we still feel L4 captures an intuitive yet complex property, without needing to directly reference any language features. Other adjustments were made to the model during development, as we found it to be under or over-specified. This process continues as we explore instances of the model and experience feeds back into our understanding of effects.

Despite these criticisms, the model succeeds in a number of ways. Our experience with our own formulation of `Joe` suggests that proving that ODE conformed to the model was no harder than proving the standard language and effects properties. In fact, we feel the structure of the model made the proofs, particularly of LS4 easier. Results such as L3 and LL2 were used directly in the proof of LS4 as well as in the meta-results. Working within the model suggested a route for proving soundness and provided tools to assist.

ODE successfully added effects to Ownership Domains. The resulting system was able to give more precise effects than `Joe` could in the same situation. Our introduction of the `siblings` effect allowed us to calculate effects for programs using iterators as described in [AC04]. The pattern of a public domain providing objects with access to private state is a fundamental pattern of ownership domains as noted by [SPH06]. As such we believe we can provide useful effects for useful ownership domains programs.

7.1 Further Work

A number of avenues are open for further investigation. There is a range of improvements and refinements that can be made to the model of effects. Since the aim of the model is to describe a minimal set of conditions for a language with effects, we should investigate whether we can weaken its requirements. In particular, an alternative to L4 that is easier to prove for model instances is desirable. Other improvements would require a more substantial change to the model. A version of the model which uses equivalence of heaps and values rather than equality would allow us to consider the definition of non-interference discussed in Section 3.6.4. Such a version should still allow us to prove that effects can judge non-interference with our current definition. The logical conclusion is a model which considers languages with concurrency. This would offer the maximum flexibility and we would expect our results for sequential execution to hold in such a model. We would need to consider small-step semantics, non-terminating executions and scheduling of threads.

To further validate the model, more instances should be explored - both through the development of new effects systems and the checking of existing systems for compliance. The language FX would be a particularly good candidate, as it represents a different language paradigm (functional) as opposed to the object-oriented instance of ODE, and would demonstrate the generality of the model.

We believe the field of encapsulation and ownership is rich with opportunities to investigate effects. Recent work on increasing the flexibility of ownership domains [SPH06] and ownership types [LP06a] prompts the question of whether it is still possible to calculate useful effects for these systems.

Our consideration of effects for predicates offers a large number of further research opportunities. The treatment in this thesis suggests a general discipline but requires more rigorous investigation. Improving the model to allow more varied predicates (as discussed in Section 3.6.5) is of primary importance. The challenge is to maintain some separation of the semantics of predicates and the properties of effects, whilst increasing the flexibility of the definition. A fuller investigation of the use of effects in program verification is needed. The integration of effects and our suggested Rule of Constancy into a language and its Hoare logic would fully validate our claims as well as providing a platform for properly evaluating the usefulness of the approach.

Bibliography

[AC04] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*. Springer, June 2004.

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 311–330, New York, NY, USA, 2002. ACM Press.

[Alm97] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*. Springer, June 1997.

[ANN99] Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. *Type and Effect Systems: Behaviours for Concurrency.* Imperial College Press, 1999.

[AY05] Alexander Ahern and Nobuko Yoshida. Formalising java rmi with explicit code mobility. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 403–422, New York, NY, USA, 2005. ACM Press.

[BCD$^+$05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, New York, NY, USA, 2005. ACM Press.

[BLS03] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–223. ACM Press, 2003.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.

[Boy03] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[BP03] Gavin M. Bierman and Matthew J. Parkinson. Effects and effect inference for a core java calculus. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[BPP03] G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge, Computer Laboratory, April 2003.

[BR01] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 56–69, New York, NY, USA, 2001. ACM Press.

[BR05] John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 283–295, New York, NY, USA, 2005. ACM Press.

[CC89] Patrick Cousot and Radhia Cousot. A language independent proof of the soundness and completeness of generalized hoare logic. *Information and Computation*, 80(2):165–191, 1989.

[CD02] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 292–310. ACM Press, 2002.

[CDNS07] Nick Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple ownership. In *OOPSLA '07: Proceedings of the 22st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, New York, NY, USA, 2007. ACM Press. to appear.

[Cla01] Dave Clarke. *Object Ownership and Containment.* PhD thesis, University of New South Wales, Australia, 2001.

[CNP01] David G. Clarke, James Noble, and John Potter. Simple ownership types for object containment. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 53–76, London, UK, 2001. Springer-Verlag.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 48–64. ACM Press, 1998.

[DDDCG01] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle : Dynamic object re-classification. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2001.

[DE97] Sophia Drossopoulou and Susan Eisenbach. Java is type safe - probably. In *ECOOP'97 — Object-Oriented Programming, 11th European Conference*, pages 389–418, 1997.

[DEW99] Sophia Drossopoulou, Susan Eisenbach, and David Wragg. A Fragment Calculus - towards a Model of Separate Compilation, Linking and Binary Compatibility. In *14th Symposium on Logic in Computer Science (LICS'99)*, pages 147–156. IEEE, July 1999.

[DM05] Werner Dietl and Peter Müller. Universes: Lightweight ownership for jml. *Journal of Object Technology*, 4(8):5–32, 2005.

[Fel91] Matthias Felleisen. On the expressive power of programming languages. In *ESOP '90: Selected papers from the symposium on 3rd European symposium on programming*, pages 35–75, Amsterdam, The Netherlands, The Netherlands, 1991. Elsevier North-Holland, Inc.

[GB99] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99 — Object-Oriented Programming, 13th European Conference*, number 1628 in Lecture Notes in Computer Science, pages 205–229, Berlin, Heidelberg, New York, 1999. Springer.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJLS87] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. Fx-87 reference manual. Technical Report 407, M.I.T. Laboratory for Computer Science, September 1987.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[GM82] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, 1982.

[HK03] Rob Hunter and Shriram Krishnamurthi. A model of garbage collection for OO languages. In *Tenth International Workshop on Foundations of Object-Oriented Languages (FOOL10)*, 2003.

[Hog91] John Hogg. Islands: aliasing protection in object-oriented languages. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285. ACM Press, 1991.

[Hu05] Liyang Hu. Inferring effect annotations. Master's thesis, Imperial College London, September 2005.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object Oriented Programming*, October 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.

[KA05] Neel Krishnaswami and Jonathan Aldrich. Permission-based ownership: Encapsulating state in higher-order typed languages. In *Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, New York, NY, USA, 2005. ACM Press.

[Lei98] K. Rustan M. Leino. Data groups: specifying the modification of extended state. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 144–153. ACM Press, 1998.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57. ACM Press, 1988.

[LM04] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[LP06a] Yi Lu and John Potter. On ownership and accessibility. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 99–123. Springer, 2006.

[LP06b] Yi Lu and John Potter. Protecting representation with effect encapsulation. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 359–371, New York, NY, USA, 2006. ACM Press.

[LPC$^+$02] G. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. Jml reference manual, 2002.

[LPHZ02] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 246–257. ACM Press, 2002.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[MPH99] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, volume 263 of *Technical Report*. Fernuniversit"at Hagen, 1999. Available from `http://softech.informatik.uni-kl.de/en/publications/univ263.html`.

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[Mül01] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*. Springer, 2001.

[Par05] Matthew J. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge Computer Laboratory, November 2005.

[PB05] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–258, New York, NY, USA, 2005. ACM Press.

[PNCB06] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic java. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, New York, NY, USA, 2006. ACM Press.

[Rey78] John C. Reynolds. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 39–46. ACM Press, 1978.

[Rey81] John C. Reynolds. *The Craft of Programming*. Prentice-Hall International Series in Computer Science. Prentice Hall International, 1981.

[Rey02a] John C. Reynolds. Separation logic: A logic for shared mutable data structures. *Logic In Computer Science*, 00:55, 2002.

[Rey02b] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[SD03] Matthew Smith and Sophia Drossopoulou. Cheaper reasoning with ownership types. In *International Workshop on Aliasing and Ownership in object-oriented programming*, ECOOP 2003, Darmstadt, Germany, 2003.

[Smi05] Matthew Smith. Towards an effects system for ownership domains. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2005)*, Glasgow, Scotland, July 2005.

[SPH06] Jan Schäfer and Arnd Poetzsch-Heffter. Simple loose ownership domains - tr. Technical Report 348/06, Department of Computer Science, University of Kaiserslautern, March 2006.

[Str00] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Syk06] Daniel Sykes. Inferring ownership types. Master's thesis, Imperial College London, September 2006.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, July 1992.

[Wri92] Andrew K. Wright. Typing references by effect inference. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582, pages 473–491. Springer-Verlag, New York, N.Y., 1992.

[YK03] Kwok Cheung Yeung and Paul H. J. Kelly. Optimising java rmi programs by communication restructuring. In Markus Endler and Douglas C. Schmidt, editors, *Middleware*, volume 2672 of *Lecture Notes in Computer Science*, pages 324–343. Springer, 2003.

[ZNV04] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time java. In *RTSS: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 241–251. IEEE Computer Society, 2004.