Technical Report

UCAM-CL-TR-654
ISSN 1476-2986

Number 654

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Local reasoning for Java

Matthew J. Parkinson

November 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted August 2005 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Abstract

This thesis develops the local reasoning approach of separation logic for common forms of modularity such as abstract datatypes and objects. In particular, this thesis focuses on the modularity found in the Java programming language.

We begin by developing a formal semantics for a core imperative subset of Java, *Middleweight Java* (MJ), and then adapt separation logic to reason about this subset. However, a naïve adaption of separation logic is unable to reason about encapsulation or inheritance: it provides no support for modularity.

First, we address the issue of encapsulation with the novel concept of an *abstract predicate*, which is the logical analogue of an abstract datatype. We demonstrate how this method can encapsulate state, and provide a mechanism for *ownership transfer*: the ability to transfer state safely between a module and its client. We also show how abstract predicates can be used to express the calling protocol of a class.

However, the encapsulation provided by abstract predicates is too restrictive for some applications. In particular, it cannot reason about multiple datatypes that have shared read-access to state, for example list iterators. To compensate, we alter the underlying model to allow the logic to express properties about read-only references to state. Additionally, we provide a model that allows both sharing and disjointness to be expressed directly in the logic.

Finally, we address the second modularity issue: inheritance. We do this by extending the concept of abstract predicates to *abstract predicate families*. This extension allows a predicate to have multiple definitions that are indexed by class, which allows subclasses to have a different internal representation while remaining behavioural subtypes. We demonstrate the usefulness of this concept by verifying a use of the visitor design pattern.

Acknowledgments

I should like to begin by thanking my supervisors, Gavin Bierman and Andrew Pitts. Their careful stewardship has guided me through this thesis and developed my understanding of research immensely. I would also like to thank my colleagues in the Computer Laboratory for providing a supportive and friendly environment to work in. Especially Alisdair Wren for his attempts to teach me grammar, and for his assiduous proof reading abilities.

The *East London Massive* (Josh Berdine, Richard Bornat, Cristiano Calcagno, Dino Distefano, Ivana, Mijajlović, and Peter O'Hearn) also deserve many thanks for providing exciting discussions. Particular thanks to Peter O'Hearn for his comments that have help focus my research.

I should like to thank the following people for feedback and discussions: Moritz Becker, Nick Benton, Sophia Drossoupolou, Manuel Fähndrich, Philippa Gardner, Tony Hoare, Andrew Kennedy, Alan Lawrence, Ronald Middlekoop, David Naumann, Uday Reddy, John Reynolds, Chris Ross, Claudio Russo, Peter Sewell, Mark Shinwell, Matthew Smith, Sam Staton, Gareth Stoyle, Hayo Thielecke, David Walker, Hongseok Yang and Nobuko Yoshida. I should also like to thank all the people whom I have forgotten to thank, as well as the anonymous referees for POPL 2005 and WOOD 2003.

Finally, I should like to thank my family and friends for their love and friendship.

Contents

1

Introduction

Software is unreliable. Currently, operating systems are susceptible to viruses, and commercial software packages are sold with disclaimers not guarantees: software licences do not say the program will function as intended or that it will not damage other data. Instead, they say that the software is not sold for any purpose, it might damage your data and that if it does, there is no entitlement to compensation. In many ways the situation is getting worse as software products grow ever larger. However, the programmer has several weapons of varying power to improve software reliability, in particular: types, testing and verification.

Types Types allow the compiler to guarantee certain errors do not occur in a program; for example, multiplying an integer by a string. Each operation, or function, is specified to work with certain types of value and the compiler guarantees that the program does not perform operations on the wrong type of data. However, types fall short of eliminating all errors. Consider the following:

```
int [] x = new int[30];
for(int n = 1; n <= 30; n++)
   x[n] = 1;
```

This program fragment creates an array of 30 elements and assigns 1 to elements 1 to 30. However, in Java arrays are numbered from 0: the array is defined from 0 to 29, not 1 to 30. Therefore, on the last iteration of the loop, the assignment will fail. Fortunately, Java provides dynamic checks for accesses outside the range of an array, so it will throw an exception. The problem is worse for programming languages without dynamic checks, like C, as they have unpredictable behaviour in the error cases.

Standard type systems do not eliminate this type of error. These properties can be checked with dependent types [87] or static analyses [29], but there are many problems where type checking or static analysis is undecidable or infeasible.

Testing Testing can be used to check more complicated properties. The program code is run many times against different inputs to see if anything "goes wrong". The inputs are selected to attempt to cover all the possible executions, and then the program's outputs are compared with the expected results.

To help automate testing and localise errors, *run-time assertions* are used to find these "wrong" states: for example, Microsoft Office has a quarter of a million run-time assertions [44]. Consider the following:

```
if(debug && x.length < 30) throw new Error();
for(int n = 0; n <30 ; n++)
   x[n] = 1;
```

The `if` statement checks that the array is of an appropriate length before proceeding and, if not, throws a descriptive error. This helps track the location of the bug: the code fails quickly. The key weakness of testing is that it is not possible to exhaustively test code: testing cannot guarantee to remove all errors as we cannot consider all possible inputs.

Program verification Finally we turn to program verification: the "holy grail" of software reliability. Rather than checking that a property holds for a number of inputs, program verification allows one to formally prove, for all possible runs of the program, that the property will hold. This allows us to make strong guarantees about the software. Floyd [39] and Hoare [45] pioneered the use of logic for program verification. Hoare's logic provides pre- and post-conditions for each command, written $\{P\}\texttt{C}\{Q\}$, which is read "if property P holds of the start state and `C` terminates, then the final state will satisfy Q." This is called *partial correctness*; total correctness additionally guarantees the program will terminate. Hoare provided axioms and rules to derive pre- and post-conditions for a given program, for example:

$$\{P[E/x]\}x := E\{P\} \qquad \frac{\{P\}\texttt{C1}\{Q\} \quad \{Q\}\texttt{C2}\{R\}}{\{P\}\texttt{C1;C2}\{R\}}$$

With rules of this form properties about a program's final values can be shown. For example one can prove an implementation of Euclid's algorithm actually gives the greatest common divisor of its two inputs.

Despite its many advantages program verification is rarely used in practice. One key reason for this lack of use is that current program verification techniques do not scale. References and pointers are a particular impediment for scalability as they allow apparently unrelated pieces of code to affect each other's behaviour (for a detailed background of this problem see §1.1.2).

However, recently, O'Hearn, Reynolds and Yang [65] have developed an approach called *local reasoning*. Local reasoning deals with pointers and references, and has the potential to scale, but thus far it has only been applied to C-like languages with no support for modularity. This thesis builds a logic for Java by extending local reasoning to account for the modularity found in Java, particularly *encapsulation* and *inheritance*. Encapsulation is where the internal representation of an object is hidden from its clients, leaving the programmer free to change their internal representation without requiring changes to the clients. Inheritance allows a class to be extended with new features. This extended class can then be used in place of the original class—this is a defining characteristic of object-oriented programming languages.

The rest of the introduction begins by discussing the relevant background material. In §1.2 we summarize the content and contribution of this thesis. We conclude the introduction by discussing and comparing related methods for reasoning about Java.

1.1 Background

We now provide a summary of the relevant background work. We assume the reader is familiar with Java, first-order logic and Hoare logic. There are two key difficulties when reasoning about object-oriented programs: reference aliasing and dynamic dispatch. We begin by discussing dynamic dispatch and a method for reasoning about it known as *behavioural subtyping*. We then present the issues associated with reference aliasing and some approaches for reasoning about aliases.

1.1.1 Dynamic dispatch

In object-oriented languages like Java, the static type does not determine the precise class of the runtime values. If a variable has static type `C`, then the runtime values could be an object of class `C` or any subtype of `C`. This definition allows any instance of a subtype to be used in place of its supertype without causing errors.[1] For a class `D` to be a subclass of `C`, `D` must have all the methods and fields of `C` and they must have the same types. We will refer to the type of a variable or field as the *static type*, and the type of the object it actually refers to as the *dynamic type*. For soundness, the dynamic type of a variable or field is always a subtype of the static type (see §2.4).

Method calls in Java are determined by the dynamic type of the receiver not by its static type. This is known as *dynamic dispatch*. Consider the following:

```
C x; x = new D(); x.m();
```

This will not call `C`'s `m` method, but instead will call `D`'s `m` method. Dynamic dispatch may seem complicated, but it can be particularly useful. Consider the following:

```
Shape [] shapes;
...
for(Shape shape : shapes) {
  shape.draw();
}
```

We omit the definition of `Shape` and its subclasses `Circle` and `Triangle`. This code uses an array of `Shape` objects and applies `draw()` to each one using Java 5's new `for` each construct. Due to dynamic dispatch, the `draw` method of the dynamic type will be invoked, e.g. the `draw` methods for a `Circle` or a `Triangle` class. The programmer does not have to rewrite this code if new shapes are created.

Now consider reasoning about programs in the presence of dynamic dispatch. Consider the following code fragment:

```
C m1(C x) {
  x.m2()
  return x;
}
```

Assume `C.m2()` has a specification $\{P_{\texttt{C}}\}\_\{Q_{\texttt{C}}\}$. We could give `m1` the specification $\{P_{\texttt{C}} \wedge \texttt{x} : \texttt{C}\}\_\{Q_{\texttt{C}}\}$, but this prevents a call to the method `m1` where the type of `x` is a subtype of `C`. Assume `C` has a single subtype `D`, which has the specification $\{P_{\texttt{D}}\}\_\{Q_{\texttt{D}}\}$ for method `m2`. We could validate the method with the following specification:

$$\{\texttt{x} \neq \texttt{null} \wedge P_{\texttt{C}} \wedge P_{\texttt{D}}\}\_\{Q_{\texttt{C}} \vee Q_{\texttt{D}}\}$$

What happens if we introduce a new class? Reverifying `m1` for every new class is clearly impractical for any realistic program. For example, the `equal()` method in the `java.lang.Object` is overridden approximately 300 times in the Java 1.4 libraries. We cannot reason about 300 classes for a single call; we need to reduce the complexity.

In the programmer's informal reasoning, the programmer makes a simplifying assumption that the methods of each subclass behave, at some level, the same as the supertype's method. If we are to reason *formally* about dynamic dispatch, we need to capture this assumption. In particular, we need to consider a new subtype relation that is appropiate for dynamic dispatch: *behavioural subtyping*. In the rest of this subsection we will describe different definitions of behavioural subtyping, and in §3.5 we present a new notion of behavioural subtyping.

[1] In Java the ideas of subtype, and subclass are conflated. Likewise in this thesis we will not separate the concepts.

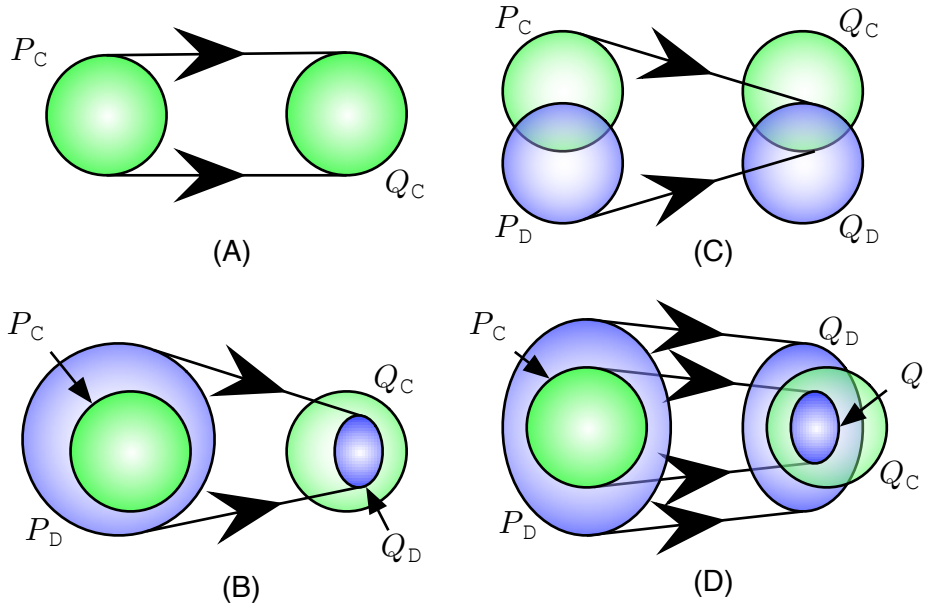

Figure 1.1: Visualization of behavioural specifications and subtyping.

Before we can define behavioural subtyping, we have to define behavioural types. America [3] considered the behaviour of a class to be specified by its methods' specifications, and the class's invariants. A method's specification is really a set of start states,[2] P_C, called its pre-condition and a set of end states, Q_C, called its post-condition, such that if the method starts in a state in P_C and the method terminates, then the terminating state will be in Q_C. We illustrate the idea of a specification in Figure 1.1 (A). This shows the states in P_C are mapped to the states in Q_C. The arrows are used to emphasis the mapping we are considering. A class's invariant is a property of a class instance that is true after every method call. Liskov and Wing [55] extend behavioural types to also consider a history property: a relationship between any state and a later state that the class or method must always preserve; for example a counter only goes up.

For a class to be a behavioural subtype of its superclass, it must preserve all of the parent's invariants, and its methods' specifications must be compatible. Compatibility means that the parent method's pre-condition must imply the subclass method's pre-condition and the subclass method's post-condition must imply the parent method's post-condition. The set of states A implies the set of states B, if and only if every state in A is also in B. We illustrate this in Figure 1.1 (B): P_C is the parent's pre-condition, P_D is the subclass's pre-condition, Q_C is the parent's post-condition and Q_D is the subclass's post-condition. In the diagram we represent the implications by containment of one region within another: P_C implies P_D as P_D contains P_C, likewise Q_D implies Q_C. Hence, we can see the mapping in (B) is a behavioural subtype of the mapping in (A).

To allow the internal representation to be altered, America [3] uses a transfer function from the internal values of the subclass to the parent's internal values. This function can allow the internal field names to be altered in the subtype. This function is similar to Hoare's data abstraction function [46] that maps from the abstract values of a datatype to their actual representation.

[2] We can interpret a formula as the set of states for which the formula is true.

Note: In this thesis we only use properties on method specifications to define behavioural subtyping. Invariants and transfer functions can be captured by the abstraction mechanism we add to the logic.

To reduce the burden of discovering and verifying compatible method specifications Dhara and Leavens [30, 31] proposed *specification inheritance*, which allows one to automatically generate specifications that ensure a class is a behavioural subtype of its parent. For each method, we must show that it satisfies all of its supertypes' specifications. For example if `C` is the only supertype of `D`, and `C`'s method `m` has specification $\{P_{\texttt{C}}\}\_\{Q_{\texttt{C}}\}$ then `D`'s specification is of the form $\{P_{\texttt{C}} \vee P_{\texttt{D}}\}\_\{Q_{\texttt{C}} \wedge Q_{\texttt{D}}\}$. It is easy to see this satisfies the requirements on method specifications. We illustrate specification inheritance in Figure 1.1 (C). Again we can see this mapping is a behavioural subtype of (A). This approach generates specifications that are behavioural subtypes even when the underlying specifications are not. Findler and Felleisen [37] show that generating run-time tests from these forms of specification can produce misleading error messages.

Poetzsch-Heffter and Müller [72] have proposed a different form of specification inheritance: each method must satisfy every specification of its supertypes' corresponding methods, for example if `D` has a method `m` with body $\overline{s}$ and has a single supertype `C` with specification $\{P_{\texttt{C}}\}\_\{Q_{\texttt{C}}\}$, then we must prove both $\vdash \{P_{\texttt{D}}\}\overline{s}\{Q_{\texttt{D}}\}$ *and* $\vdash \{P_{\texttt{C}}\}\overline{s}\{Q_{\texttt{C}}\}$. Hence every method body requires multiple verifications.

This is more general than Dhara and Leavens's specification inheritance as it allows $\overline{s}$ to terminate in a state not satisfying $Q_{\texttt{C}}$ as long as it does not start in a state satisfying $P_{\texttt{C}}$. In Figure 1.1 (D) we illustrate a method that is compatible at the behavioural level, but does not satisfy the standard pair of implications [3, 55]. We consider `D`'s method to satisfy the intersection of two specifications: one maps $P_{\texttt{C}}$ to Q and the other maps $P_{\texttt{D}}$ to $Q_{\texttt{D}}$. $Q_{\texttt{D}}$ does not have to be contained in $Q_{\texttt{C}}$, but Q must be contained in $Q_{\texttt{C}}$. This generalises Dhara and Leavens specification inheritance, and moves beyond the standard pair of implications.

In §3.5 we define a new notion *specification compatibility* that captures Poetzsch-Heffter and Müller's notion of specification inheritance without the need for multiple verifications of each method.

1.1.2 Aliasing

We have so far discussed methods for reasoning about dynamic dispatch. The second key stumbling block to reasoning about object-oriented languages, like Java, is reference aliasing. Consider the following code:

```
...
x.f = 3;
y.f = 4;
ASSERT(x.f != y.f)
```

The assertion might initially seem to hold, but in languages with references, or pointers, multiple names can refer to the same entity. This is *aliasing*. The assertion only holds if `x` and `y` reference different objects.

Aliasing makes reasoning about programs considerably harder. It breaks Hoare's original assignment rule [45]: $\{P[E/l]\}$ `l = E;` $\{P\}$. Consider an unsound use of this rule:

$$\{x.f = 3 \wedge 4 = 4 \wedge x = y\} \texttt{ y.f = 4;} \{x.f = 3 \wedge y.f = 4 \wedge x = y\}$$

This is an instance of the assignment rule because $x.f = 4 \wedge y.f = 4 \wedge x = y[4/y.f]$ equals $x.f = 3 \wedge 4 = 4 \wedge x = y$. For any sensible semantics the post-condition is *false* as it implies $3 = 4$.

One solution to the problem of assignment with aliasing is Morris's component substitution [59]. He modifies substitution to operate on components.

$$x.a[E/y.a] \quad \stackrel{\text{def}}{=} \quad \text{if } x = y \text{ then } E \text{ else } x.a$$

Every time we make a possibly aliased assignment, we insert a conditional test for the cases where it was aliased and where it was not. Alternatively one can see the components as an array, for example seeing $x.a$ as an array access $a[x]$. This then allows the Hoare logic rules for array assignment to be used rather than introducing the conditional test. These methods explicitly demonstrate the global nature of the store. Assignment to a particular component can potentially affect any other assertion about that component.

In Hoare logic without aliased state one can write local specifications, and then infer global specifications using the Hoare logic rule of invariance [4], or the specification logic rule of constancy [77]:

$$\frac{\{P\}\overline{s}\{Q\}}{\{P \wedge R\}\overline{s}\{Q \wedge R\}} \text{ provided } \overline{s} \text{ does not modify any free variables of R}$$

Without aliasing the side-condition can be verified by a simple syntactic check of $\overline{s}$. When aliasing is present the check is no longer simply syntactic and more complex conditions about aliasing and interference must be used.

This problem impacts on behavioural types. The simplest solution is to require methods to be annotated with the details of how they alter the state. For example [43] the annotation "`modifies this.f1, x.f2;`" specifies that the method only alters the receiver's `f1` field, and `x`'s `f2` field, no other state is modified. However, in the presence of behavioural subtyping these clauses are too precise: subtypes usually modify more state than their parents modify (they are, after all, extensions), but for soundness it can only modify the same or less. This is known as the *extended state problem*.

This problem was addressed by Leino [54]. He proposed the use of *data groups*, which allows the `modifies` clauses to be given in terms of abstract groups of fields. The membership of a group is then specified by the class. Hence it is possible for a subclass to modify more state than its parent, but it cannot modify data from any more data groups.

Whilst data groups solve some problems, they cannot directly deal with recursive data structures as the shape of the heap affects the values that are modified. Consider a method that modifies a list. What is the `modifies` clause? For a three element list it might be `x.tl,x.tl.tl, x.tl.tl.tl`, but clearly this approach does not scale. (One might imagine providing a regular expression for the `modifies` clause, but this is perhaps too complicated.)

To extend `modifies` clauses to recursive data structures, some form of ownership has been proposed [32, 60, 61, 79]. An object or field is said to own other objects or fields. If a field is specified as modified, then that field and all of the other state it owns could be modified. We will briefly describe some type systems that enforce ownership properties but refer the interested reader to Clarke's thesis [22] for a comprehensive survey of different type systems for reasoning about ownership and aliasing.

Ownership types [15, 22, 23, 24] allow the programmer to enforce one object's ownership of another. In Figure 1.2, we demonstrate a typical ownership diagram: the two node objects are owned by the list object, and the list and data objects are owned by a distinguished "world" object. The objects owned by a particular object are viewed as its representation: ownership types prevent external references into this representation, and hence enforce encapsulation of its representation. Universes [32] take a similar approach to ownership types. They only prevent the writing to the representation rather than any access. This allows universes to handle list

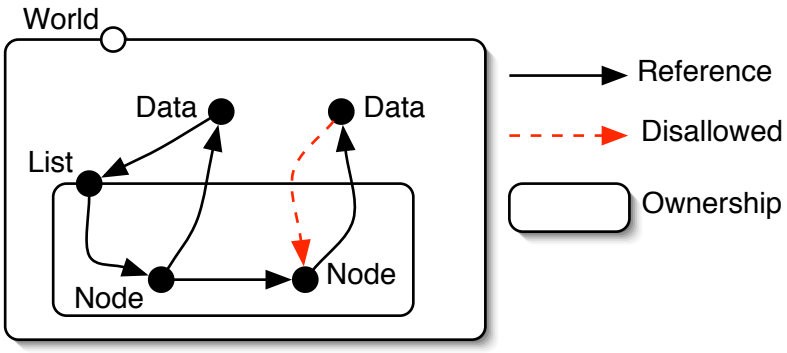

Figure 1.2: Example of ownership types [23]

iterators that are difficult to handle in ownership types. However, universes are less expressive in representing nested ownership.

So far we have reviewed some of the difficulties with reasoning about aliased state and described briefly some solutions. Now we will discuss *local reasoning*, which comes from observations of Burstall [20]. He noted that if you explicitly consider disjointness in the logic then you can simplify your reasoning. He presented a series of examples on lists and trees, and presented a predicate to assert two lists were acyclic and had disjoint domains. Bornat [12] has shown how to extend this style of reasoning to a more general recursive datatype setting.

Based on Burstall's observations about disjointness, Reynolds [76], and Ishtiaq and O'Hearn [48] took a different approach. Rather than adding a predicate for disjointness, they added a new logical connective for disjointness: a second form of conjunction $P * Q$, which means both P and Q hold, but in disjoint parts of the heap. Ishtiaq and O'Hearn [48] showed this new connective was the multiplicative conjunction from the "Logic of Bunched Implications" by O'Hearn and Pym [64].

This programming logic is now known as *separation logic*. It has a subtly different semantics from standard Hoare logic as the triples are *tight specifications*: that is for $\{P\}\overline{s}\{Q\}$ to hold, P must describe all the heap $\overline{s}$ needs during execution. This leads to the *frame rule*:

$$\frac{\{P\}\overline{s}\{Q\}}{\{P * R\}\overline{s}\{Q * R\}} \text{ provided } \overline{s} \text{ does not modify the free variables of } R.$$[3]

Although this rule has a side-condition, it is simple and syntactic as the issues with aliasing are dealt with using the $*$ connective. There is no need for type systems to check uniqueness or ownership. Additionally, if one precludes global variables then methods will always have an empty modifies set, removing the need for `modifies` clauses: the specifications contain all the information about the state that is modified.

The frame rule allows *local reasoning*: the code can be specified based solely on the state it uses. Clients can then extend the specification to the global setting that accounts for all of their additional state. This kind of reasoning is essential for open programs: programs where the client cannot verify their requirements against the implementation, but must simply rely on the specification provided.

[3] Bornat [13] has shown how to treat the framing of variables with $*$, hence removing the side-condition.

1.2 Content and contribution

We will now provide an overview of the thesis's content and highlight its contribution.

Chapter 2 We begin the thesis by identifying Middleweight Java (MJ), a core imperative fragment of Java. We formalize the type system for this fragment and give a novel operational semantics, which we prove is type sound.
Contribution: The key contributions of this chapter are the development of the semantics and type system and the proof of soundness.

Chapter 3 This chapter extends local reasoning to the Java programming language and serves as an introduction to separation logic. A rule is given for dynamic dispatch based on a new formulation of behavioural subtyping named *specification inheritance*. It describes how this formulation is more flexible than standard behavioural subtyping. We illustrate the logic's use with a simple tree copy program and prove the logic is sound with respect to the semantics in Chapter 2. We conclude by showing that the logic allows local reasoning, but does not provide modular reasoning in the sense of encapsulation or inheritance.
Contribution: This chapter has two key contributions: a separation logic for Java and a new definition of behavioural subtyping named *specification compatibility*.

Chapter 4 In this chapter we address the issues of encapsulation and abstract datatypes, by developing a novel concept of an *abstract predicate*. Abstract predicates are the logical equivalent of an abstract datatype. We show the encapsulation they provide allows *ownership transfer* [66]: the ability for state to be safely transferred between the client and the module, or class. In particular, we show how to reason about the object pool pattern [42]. We prove the extension is sound and conclude the chapter with a discussion of related work, and a demonstration of how to reason about a C-style memory manager.
Contribution: The key contribution of this chapter is the concept of an *abstract predicate*. Additionally, we demonstrate that abstract predicates provide more flexible modularity than the hypothetical frame rule [66].

Chapter 5 This chapter shows that the encapsulation provided by abstract predicates in separation logic is too strict to allow many common programming idioms, including list iterators. We present a new model of separation logic that allows read sharing. In the standard presentation of separation logic, only a single datatype (or, in the concurrent setting, a thread) can have access to a location in memory. In this chapter, we extend separation logic to allow multiple datatypes to own read references to the same location and demonstrate this work with a list iterator example. However, the initial presentation does not allow the disjointness of datastructures to be properly expressed. We present a model of named permissions and show that this model is capable of representing disjointness in datatypes while allowing read sharing.
Contribution: This chapter has two key contributions: demonstrating the use of read-only permissions with abstract predicates and presenting a model of separation logic that can express both read sharing and disjointness.

Chapter 6 This chapter highlights the issues of reasoning about inheritance in separation logic. We extend the concept of an abstract predicate to an *abstract predicate family*. The extension allows a predicate to have one of many different definitions based on its argument's type. We demonstrate the utility of abstract predicate families with two examples: Cell/Recell [1] a simple

example of subtyping; and the visitor design pattern [40]. We prove this extension is sound, and conclude by discussing future work.
Contribution: The key contribution of this chapter is the *abstract predicate family* and the demonstration of its utility when reasoning about object-oriented programs.

Chapter 7 This chapter concludes the thesis.

1.3 Related work

To conclude this chapter, we discuss other logics for reasoning about object-oriented languages, and compare them to the logic developed in this thesis. We will not consider object-based languages [1] and associated logics [2] for two reasons: firstly, we want our verification methods to be close to programmers, who generally use class-based languages; and secondly, the class hierarchy actually imposes a structure which helps with specification. Reus [74] has given a denotational comparison between specifications for class- and object-based languages and shows that the domains are simpler for class-based languages.

There are several Hoare-like logics embedded into theorem provers. Von Oheimb and Nipkow [84] embed a logic into Isabelle. The assertion language is part of Isabelle, and the assertions use an explicit state parameter. They do not support any form of encapsulation, inheritance or frame-like property. Poetzsch-Heffter and Müller [71] present a different logic for reasoning about Java-like languages. Again, they use an explicit store in the logic. However, they give axioms for manipulating the store [72]. Müller's thesis [60] addresses the issues of modularity by adding abstract fields with dependency information and universe types [32]. He provides `modifies` clauses to provide a frame property. The `modifies` clauses are given in terms of the abstract fields and the fields they depend on. The universe types [32] allow object invariants to depend on other objects.

The syntax of Müller's assertion language is closely related to Leavens *et al.*'s JML [53], which attempts to unify many approaches to tool support by giving them a common syntax. There are now a reasonable collection of tools for JML [19]: some compile dynamic checks into the code; some provide test cases; and some do static verification. JML does not provide any help with dealing with aliasing. Müller *et al.* [61] apply the ideas from Müller's thesis on integrating universes to JML to provide a modular frame property, but this is not part of the language yet. Many of the tools use their own methods, and syntax, for dealing with aliasing. One notable example of a JML tool is ESC/Java [29]. Interestingly it is not designed to be sound or complete, but simply to find common errors.

In a similar vein to JML, Spec$^\sharp$ of Barnett, Leino, and Schulte [6] allows assertions to be statically verified. Any assertions that cannot be statically verified will be converted into run-time checks. Many difficult features, such as invariants, are modelled using additional logical fields: fields only required for the proof. These allow the invariant to be broken while the field is false. The invariant is, in effect, $\texttt{this}.inv \Rightarrow I$: while the field is true, I must hold, otherwise it may or may not hold. More complex schemes can be used where the invariant is a stack of invariants corresponding to each of the superclasses' invariants [5]. This work encodes the ideas of object ownership into fields that represent object ownership. Recently, there has been a proposal to extend this methodology to a concurrent setting [49].

A different, automated method has been taken by Fähndrich and DeLine [36]. They revisit some work on compiler detected errors known as typestates [81], and have developed an object-oriented extension. They use typestates as extended invariants: an invariant that has many different, visible states. Typestates can be viewed as abstract predicates on an object's state, in a very similar way to the work developed in Chapters 4 and 6. Fähndrich and DeLine use

typestates to check for errors in calling protocols. Their logic is considerable less expressive than the logic developed in this thesis, but their aim was automatic checking, not correctness proofs. They use the logic for an automatic protocol checker Fugue [28].

There have been a few works based on the component substitution ideas of Morris [59]. In particular, De Boer and Pierik [26, 27, 69, 70] have extended this to an object-oriented setting and introduce a form of substitution for dynamic allocation of objects, $[\texttt{new}/u]$. Recently Berger, Honda, and Yoshida [7] have also developed a logic based on Morris's work [59]. They add modal operators about locations, which are used to provide structural reasoning about observations. The work differs fundamentally from separation logic as it does not have explicit concepts of resource. Their logic can, however, deal with higher order functions.

Finally, Middelkoop's recent Master's thesis [56, 57] also adapts separation logic to reason about Java. However, his work takes many different design decision to this thesis, and does not attempt to solve any of the issues surrounding modularity.

The logics outlined above either do not provide support for modularity and framing, or if they do, then the extensions are separate from the logic. For example, the logical rules might use information from the type systems for ownership or uniqueness. However, this tight integration can make it hard to express some dependencies. Consider the following code:

```
void m() {
   if(x == null) {
      <code modifying (1)>
   } else {
      <code modifying (2)>
   }
}
```

Given a standard `modifies` clause we cannot express the dependency between the state and what is modified: the method modifies the union of (1) and (2). To then get an accurate specification we would than have to add assertions that say (1) is unchanged when `x` is not `null`, and (2) is not changed otherwise. However, we have lost much of the advantage of `modifies` clauses as the specifications must provide additional framing properties.

By using separation logic we automatically get notions of framing and ownership *in the logic*. This provides a single unified logical view of these concepts rather than having to extend our language with many different ideas. We believe this leads to a clearer logic and to simpler proof techniques.

2

Middleweight Java

In this chapter we define Middleweight Java, MJ, our proposal for an imperative core calculus for Java. MJ contains the core object-oriented and imperative features of Java, in particular block structured local state, mutable fields, inheritance and method overiding. MJ does not contain interfaces, super calls, static fields, reflection, concurrency and inner classes.

It is important to note that MJ is an entirely valid subset of Java: all MJ programs are literally executable Java programs. An alternative would be to allow extra-language features; for example, Classic Java uses annotations and let bindings which are not valid Java syntax in the operational semantics [38]. This is important as we want programmers to reason directly about this code, without having to consider a separate language.

This chapter is structured as follows. We present MJ's syntax in §2.1, its type system in §2.2 and its single-step operational semantics in §2.3. In §2.4 we provide definitions for typing configurations, and in §2.5 we present a proof of correctness of the type system, and conclude in §2.6 with a discussion of related Java semantics.

2.1 Syntax

The syntax for MJ programs is given in Figure 2.1. An MJ program is a collection of class definitions plus a sequence of statements, $\overline{s}$, to be evaluated. This sequence corresponds to the body of the `main` method in a Java program [41, §12.1.4].

For example, Figure 2.2 presents some typical MJ class definitions. This code defines two classes: `Cell` which is a subclass of the `Object` class, and `Recell` which is a subclass of `Cell`. The `Cell` class has one field, `contents`. The constructor simply assigns to the field the value of the parameter. The `Cell` class defines a method `set`, which sets the field to a given value.

`Recell` objects inherit the `contents` field from their superclass, and also have another field, `undo`. The constructor first calls the superclass's constructor (which will assign the `contents` field) and then sets the `undo` field to the `null` object reference. The `Recell` class definition overrides the `set` method of its superclass.

As with Featherweight Java [47], FJ, we insist on a certain amount of syntactic regularity in class definitions, although this is really just to make the definitions compact. We insist that all class definitions (1) include a supertype (we assume a distinguished class `Object`); (2) include a constructor (for simplicity we only allow a single constructor per class); (3) have a call to `super` as the first statement in a constructor method; (4) have a `return` as the last statement of every method definition except for `void` methods and constructor definitions (this

Program

π ::= $cd_1 \ldots cd_n; \overline{s}$

Class definition

cd ::= `class` C `extends` $C\{\overline{fd}\ \ cnd\ \ \overline{md}\}$

Field definition

fd ::= $C\ f;$

Constructor definition

cnd ::= $C(\overline{C}\ \overline{x})\{$`super`$(\overline{e});\ \overline{s}\}$

Method definition

md ::= $\tau\ m(\overline{C}\ \overline{x})\{\overline{s}\}$

Return type

τ ::= $C \mid$ `void`

Expression

e	::=	x	Variable
	$\mid$	`null`	Null
	$\mid$	$e.f$	Field access
	$\mid$	$(C)e$	Cast
	$\mid$	pe	Promotable expression

Promotable expression

pe	::=	$e.m(\overline{e})$	Method invocation
	$\mid$	`new` $C(\overline{e})$	Object creation

Statement

s	::=	;	No-op
	$\mid$	pe;	Promoted expression
	$\mid$	`if` $(e == e)\{\overline{s}\}$ `else` $\{\overline{s}\}$	Conditional
	$\mid$	$e.f = e$;	Field assignment
	$\mid$	$C\ x$;	Local variable declaration
	$\mid$	$x = e$;	Variable assignment
	$\mid$	`return` e;	Return
	$\mid$	$\{\overline{s}\}$	Block

Figure 2.1: Syntax for MJ programs

```
class Cell extends Object{
  Object contents;
  Cell (Object start){
    super();
    this.contents = start;
  }
  void set(Object update){
    this.contents = update;
  }
}
```

```
class Recell extends Cell{
  Object undo;
  Recell (Object start){
    super(start);
    this.undo = null;
  }
  void set(Object update){
    this.undo = this.contents;
    this.contents = update;
  }}
```

Figure 2.2: Example MJ code: Cell and Recell

constraint is enforced by the type system); and (5) write out field accesses explicitly, even when the receiver is `this`.

In what follows, again for compactness, we assume that MJ programs are well-formed, that is we insist that (1) they do not have duplicate definitions for classes, fields and methods (for simplicity we do not allow overloaded methods—as overloading is determined statically, overloaded methods can be simulated faithfully in MJ); (2) fields are not redefined in subclasses (we do not allow shadowing of fields); and (3) there are no cyclic class definitions (for example `A extends B` and `B extends A`). We do not formalise these straightforward conditions.

Returning to the definition in Figure 2.1: a class definition contains a collection of field and method definitions, and a single constructor definition. A field is defined by a type and a name. Methods are defined as a return type, a method name, an ordered list of arguments, where an argument is a variable type and name, and a body. A constructor is defined by the class name, an ordered list of arguments and a body. There are a number of well-formedness conditions for a collection of class definitions which are formalised in the next section.

The rest of Figure 2.1 defines MJ expressions and statements. We assume a number of metavariables: f ranges over field names, m over method names, and x over variables. We assume that the set of variables includes a distinguished variable, `this`, which is not permitted to occur as the name of an argument to a method or on the left of an assignment. In what follows, we shall find it convenient to write $\overline{e}$ to denote the possibly empty sequence $e_1, \ldots, e_n$ (and similarly for $\overline{C}, \overline{x}$, etc.). We write $\overline{s}$ to denote the sequence $s_1 \ldots s_n$ with no commas (and similarly for $\overline{fd}$ and $\overline{md}$). We abbreviate operations on pairs of sequences in the obvious way, thus for example we write $\overline{C}\,\overline{x}$ for the sequence $C_1\,x_1, \ldots, C_n\,x_n$ where n is the length of $\overline{C}$ and $\overline{x}$.

The reader will note MJ has two classes of expressions: the class of 'promotable expressions', pe, defines expressions that can be can be promoted to statements by postfixing a semicolon '`;`'; the other, e, defines the other expression forms. This slightly awkward division is imposed upon us by our desire to make MJ a valid fragment of Java. Java [41, §14.8] only allows particular expression forms to be promoted to statements by postfixing a semicolon. This leads to some rather strange syntactic surprises: for example, `x++;` is a valid statement, but `(x++);` is not!

MJ includes the essential imperative features of Java. Thus we have fields, which can be both accessed and assigned to, as well as variables, which can be locally declared and assigned. As with Java, MJ supports block-structure; consider the following valid MJ code fragment.

```
if(var1 == var2) { ; }
else {
   Object temp;
   temp = var1;
   var1 = var2;
   var2 = temp;
}
```

This code compares two variables, `var1` and `var2`. If they are not equal then it creates a new locally scoped variable, `temp`, and uses this to swap the values of the two variables. At the end of the block, `temp` will no longer be in scope and will be removed from the variable stack.

2.2 Types

As with FJ, for simplicity, MJ does not have primitive types, sometimes called base types. Thus all well-typed expressions are of a class type, C. All well-typed statements are of type `void`, except for the statement form `return` e; which has the type of e, i.e. a class type. We use τ to range over valid statement types. The type of a method is a pair, written $\overline{C} \to \tau$, where $\overline{C}$ is a sequence of argument class types and τ is the return type (if a method does not return anything, its return type is `void`). We use μ to range over method types.

Expression types

C a valid class name, including a distinguished class `Object`

Statement types

τ ::= `void` | C

Method types

μ ::= $C_1, \ldots, C_n \to \tau$

Java, and MJ, class definitions contain both typing information and code. This typing information is extracted and used to typecheck the code. Thus before presenting the typechecking rules we need to specify how this typing information is induced by MJ code.

This typing information consists of two parts: The subclassing relation, and a class table (which stores the types associated with classes). First let us consider the subclassing relation. Recall that in MJ we restrict class declarations so that they must give the name of class that they are extending, even if this class is the `Object` class. We forbid the `Object` class being defined in the program.

A well-formed program, π, then induces an immediate subclassing relation, which we write $\prec_1$.[1] We define the *subclassing relation*, $\prec$, as the reflexive, transitive closure of this immediate subclassing relation. This can be defined formally as follows.

TR-IMMEDIATE	$\dfrac{\texttt{class } C_1 \texttt{ extends } C_2\{\ldots\} \in \pi}{C_1 \prec_1 C_2}$
TR-TRANSITIVE	$\dfrac{C_1 \prec C_2 \quad C_2 \prec C_3}{C_1 \prec C_3}$
TR-EXTENDS	$\dfrac{C_1 \prec_1 C_2}{C_1 \prec C_2}$
TR-REFLEXIVE	$C \prec C$

A well-formed program π also induces a class table, δ_π. As the program is fixed, we will simply write this as δ. A class table, δ, is actually a triple, $(\delta_{\mathrm{M}}, \delta_{\mathrm{C}}, \delta_{\mathrm{F}})$, which provides typing information about the methods, constructors, and fields, respectively. δ_{M} is a partial map from a class name to a partial map from a method name to that method's type. Thus $\delta_{\mathrm{M}}(C)(m)$ is intended to denote the type of method m in class C. δ_{C} is a partial map from a class name to the type of that class's constructor's arguments. δ_{F} is a partial map from a class name to a map from a field name to a type. Thus $\delta_{\mathrm{F}}(C)(f)$ is intended to denote the type of f in class C. The details of how a well-formed program π induces a class table δ are given below.

Notation: $\delta_{\mathrm{M}}^{\Delta}$ and $\delta_{\mathrm{F}}^{\Delta}$ define the types of the methods and fields that each class defines, respectively. δ_{M} and δ_{F} are the types of the methods and fields a class defines and inherits, respectively.

[1] This relation is also a partial function that uniquely defines all non-`Object` classes' immediate superclass.

Immediate definitions

$$\begin{array}{lcll} \delta_{\mathrm{C}}(C) & \stackrel{\text{def}}{=} & \overline{C} & \text{where } cnd = C(\overline{C}\overline{x})\{\ldots\} \\ \delta^{\Delta}_{\mathrm{M}}(C) & \stackrel{\text{def}}{=} & \{m \mapsto (\overline{C} \to C'') \mid C''m(\overline{C}\overline{x})\{\ldots\} \in \overline{md}\} & \\ \delta^{\Delta}_{\mathrm{F}}(C) & \stackrel{\text{def}}{=} & \{f \mapsto C'' \mid C''f; \in \overline{fd}\} & \\ & & & \text{where } \texttt{class}\ \ C\ \texttt{extends}\ C'\{\overline{fd}\ cnd\ \overline{md}\} \in \pi \end{array}$$

Inherited definitions

$$\begin{array}{lcl} \delta_{\mathrm{M}}(C)(m) & \stackrel{\text{def}}{=} & \begin{cases} \delta^{\Delta}_{\mathrm{M}}(C)(m) & \delta^{\Delta}_{\mathrm{M}}(C)(m) \text{ defined} \\ \delta_{\mathrm{M}}(C')(m) & \text{otherwise} \end{cases} \\ \delta_{\mathrm{F}}(C)(f) & \stackrel{\text{def}}{=} & \begin{cases} \delta^{\Delta}_{\mathrm{F}}(C)(m) & \delta^{\Delta}_{\mathrm{F}}(C)(m) \text{ defined} \\ \delta_{\mathrm{F}}(C')(m) & \text{otherwise} \end{cases} \\ & & \text{where } C \prec_1 C' \end{array}$$

Note: The constructors are not inherited.

Until now we have assumed that the class definitions are well-formed. Now let us define formally well-formedness of class definitions. First we will find it useful to define when a type (either a method type or statement type) is well-formed with respect to a class table. This is written as a judgement $\delta \vdash \mu$ ok and $\delta \vdash \tau$ ok which mean that μ and τ are a valid type given the class table δ, respectively. (We define $dom(\delta)$ to be the domain of δ_{F}, δ_{M} or δ_{C}, which are all equal.)

T-CTYPE
$$\frac{C \in dom(\delta)}{\delta \vdash C\ \mathsf{ok}}$$

T-VTYPE
$$\delta \vdash \texttt{void}\ \mathsf{ok}$$

T-MTYPE
$$\frac{\delta \vdash C_1\ \mathsf{ok} \quad \ldots \quad \delta \vdash C_n\ \mathsf{ok} \quad \delta \vdash \tau\ \mathsf{ok}}{\delta \vdash C_1, \ldots, C_n \to \tau\ \mathsf{ok}}$$

Now we define formally the judgement for well-formed class tables, which is written $\vdash \delta$ ok. This essentially checks two things: firstly that all types are valid given the classes defined in the class table, and secondly that if a method is overridden then it is done so at the *same* type. The judgements are given as follows

$$\text{T-FIELDSOK}\quad \frac{\forall f \in dom(\delta_{\mathrm{F}}^{\Delta}(C)).(\quad \delta \vdash \delta_{\mathrm{F}}^{\Delta}(C)(f)\ \mathsf{ok} \quad C \prec_1 C' \quad \delta_{\mathrm{F}}(C')(f)\ \text{undefined} \quad)}{\delta \vdash_C \delta_{\mathrm{F}}\ \mathsf{ok}}$$

$$\text{T-CONSOK}\quad \frac{\forall C' \in \delta_{\mathrm{C}}(C).\quad \delta \vdash C'\ \mathsf{ok}}{\delta \vdash_C \delta_{\mathrm{C}}\ \mathsf{ok}}$$

$$\text{T-METHOK}\quad \frac{\delta \vdash \mu\ \mathsf{ok} \quad \delta_m(C)(m) = \mu \quad C \prec_1 C' \quad m \in dom(\delta_m(C')) \Rightarrow \delta_m(C')(m) = \mu}{\delta \vdash C.m\ \mathsf{ok}}$$

$$\text{T-METHSOK}\quad \frac{\forall m \in dom(\delta_{\mathrm{M}}(C)).\quad \delta \vdash C.m\ \mathsf{ok}}{\delta \vdash_C \delta_{\mathrm{M}}\ \mathsf{ok}}$$

$$\text{T-CLASSOK}\quad \frac{\forall C \in dom(\delta).(\ \delta \vdash_C \delta_{\mathrm{F}}\ \mathsf{ok} \quad \delta \vdash_C \delta_{\mathrm{M}}\ \mathsf{ok} \quad \delta \vdash_C \delta_{\mathrm{C}}\ \mathsf{ok} \quad C \prec_1 C' \quad \delta \vdash C'\ \mathsf{ok}\ )}{\vdash \delta\ \mathsf{ok}}$$

A typing environment, γ, is a map from program variables to expression types. We write $\gamma, x\colon C$ to denote the map γ disjointly extended so that x maps to C. We write $\delta \vdash \gamma\ \mathsf{ok}$ to mean that the types in the image of the typing environment are well-formed with respect to the class table, δ.

We can now define the typing rules for expressions and promotable expressions. A typing judgement for a given MJ expression, e, is written $\delta; \gamma \vdash e\colon C$ where δ is a class table and γ is a typing environment. These rules are given in Figure 2.3. Our treatment of casting is the same as in FJ [47]—thus we include a case for 'stupid' casting, where the target class is completely unrelated to the subject class. This is needed to handle the case where an expression without stupid casts may reduce to one containing a stupid cast. Consider the following expression, taken from [47], where classes `A` and `B` are both defined to be subclasses of the `Object` class, but are otherwise unrelated.

```
(A)(Object)new B()
```

According to the Java Language Specification [41, §15.16], this is well-typed, but consider its operational behaviour: a `B` object is created and is dynamically upcast to `Object` (which has no dynamic effect). At this point we wish to downcast the `B` object to `A`—a 'stupid' cast! Thus if we wish to maintain a subject reduction theorem (that the result of a single step reduction of a well-typed program is also a well-typed program) we need to include the TE-STUPIDCAST rule. For the same reason, we also require two rules for typing an `if` statement. Java [41, §15.21.2] enforces statically that when comparing objects, one object should be a subclass of the other. However this is not preserved by the dynamics. Consider again the unrelated classes `A` and `B`. The following code fragment is well-typed but at runtime will end up comparing objects of unrelated classes.

```
if ((Object)new A() == (Object)new B()) { ... } else {...};
```

Definition 2.2.1. A *valid* Java/MJ program is one that does not have occurrences of TE-STUPIDCAST or TS-STUPIDIF in its typing derivation.

A typing judgement for a sequence of statements, $\overline{s}$, is written $\delta; \gamma \vdash \overline{s}\colon \tau$ where δ is a class table and γ is a typing environment. We begin by presenting rules for a singleton sequence. As we noted earlier, statements in Java can have a non-`void` type (see rule TS-RETURN). The rules for forming these typing judgements are given in Figure 2.4.

TE-VAR
$$\frac{\delta \vdash \gamma \ \mathsf{ok} \quad \vdash \delta \ \mathsf{ok}}{\delta; \gamma, x\colon C \vdash x\colon C}$$

TE-NULL
$$\frac{\delta \vdash C \ \mathsf{ok} \quad \delta \vdash \gamma \ \mathsf{ok} \quad \vdash \delta \ \mathsf{ok}}{\delta; \gamma \vdash \texttt{null}\colon C}$$

TE-FIELDACCESS
$$\frac{\delta; \gamma \vdash e\colon C_2 \quad \delta_{\mathrm{F}}(C_2)(f) = C_1}{\delta; \gamma \vdash e.f\colon C_1}$$

TE-UPCAST
$$\frac{\delta; \gamma \vdash e\colon C_2 \quad \delta \vdash C_1 \ \mathsf{ok}}{\delta; \gamma \vdash (C_1)e\colon C_1}$$
provided $C_2 \prec C_1$

TE-DOWNCAST
$$\frac{\delta; \gamma \vdash e\colon C_2 \quad \delta \vdash C_1 \ \mathsf{ok}}{\delta; \gamma \vdash (C_1)e\colon C_1}$$
provided $C_1 \prec C_2$

TE-STUPIDCAST
$$\frac{\delta; \gamma \vdash e\colon C_2 \quad \delta \vdash C_1 \ \mathsf{ok}}{\delta; \gamma \vdash (C_1)e\colon C_1}$$
provided $C_2 \not\prec C_1 \ \wedge \ C_1 \not\prec C_2$

TE-METHOD
$$\frac{\delta; \gamma \vdash e\colon C' \quad \delta; \gamma \vdash e_1\colon C_1 \quad \ldots \quad \delta; \gamma \vdash e_n\colon C_n}{\delta; \gamma \vdash e.m(e_1, \ldots, e_n)\colon C}$$
provided $\delta_{\mathrm{M}}(C')(m) = C'_1, \ldots, C'_n \to C \ \wedge \ C_1 \prec C'_1 \ \ldots \ C_n \prec C'_n$

TE-NEW
$$\frac{\delta; \gamma \vdash e_1\colon C'_1 \quad \ldots \quad \delta; \gamma \vdash e_n\colon C'_n}{\delta; \gamma \vdash \texttt{new}\ C(e_1, \ldots, e_n)\colon C}$$
provided $\delta_{\mathrm{C}}(C) = C_1, \ldots, C_n \ \wedge \ C'_1 \prec C_1 \ \ldots \ C'_n \prec C_n$

Figure 2.3: Typing rules for expressions

Java allows variable declarations to occur at any point in a block. To handle this we introduce two typing rules for sequencing: TS-INTRO for the case where the first statement in the sequence is a variable declaration, and TS-SEQ for the other cases. An alternative approach would be to force each statement to return the new typing environment. We feel that our presentation is simpler.

Finally in Figure 2.4 we define the typing of the `super` call in the constructor of a class. A call to the empty constructor in a class that directly extends `Object` is always valid, and otherwise it must be a valid call to the constructor of the parent class. The expressions evaluated before this call are not allowed to reference `this` [41, §8.8.5.1]. This is because it is not a validly initialised object until the parent's constructor has been called.

Note: We could remove the subtyping from the individual rules and add a single subtype rule. However this would allow implicit casting where Java does not.

Before we give the typing rules involving methods we first define some useful auxiliary functions for accessing the bodies of constructors and methods.

TS-NoOp
$$\frac{\vdash \delta \text{ ok} \quad \delta \vdash \gamma \text{ ok}}{\delta;\gamma \vdash ; : \texttt{void}}$$

TS-PE
$$\frac{\delta;\gamma \vdash pe : \tau}{\delta;\gamma \vdash pe; : \texttt{void}}$$

TS-If
$$\frac{\delta;\gamma \vdash \overline{s_1} : \texttt{void} \quad \delta;\gamma \vdash \overline{s_2} : \texttt{void} \quad \delta;\gamma \vdash e_1 : C' \quad \delta;\gamma \vdash e_2 : C''}{\delta;\gamma \vdash \texttt{if}\ (e_1 == e_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\} : \texttt{void}}$$
provided $C'' \prec C' \vee C' \prec C''$

TS-StupidIf
$$\frac{\delta;\gamma \vdash \overline{s_1} : \texttt{void} \quad \delta;\gamma \vdash \overline{s_2} : \texttt{void} \quad \delta;\gamma \vdash e_1 : C' \quad \delta;\gamma \vdash e_2 : C''}{\delta;\gamma \vdash \texttt{if}\ (e_1 == e_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\} : \texttt{void}}$$
provided $C'' \not\prec C' \ \wedge\ C' \not\prec C''$

TS-FieldWrite
$$\frac{\delta;\gamma \vdash e_1 : C_1 \quad \delta;\gamma \vdash e_2 : C_2 \quad \delta_{\text{F}}(C_1)(f) = C_3}{\delta;\gamma \vdash e_1.f = e_2; : \texttt{void}}$$
provided $C_2 \prec C_3$

TS-VarWrite
$$\frac{\delta;\gamma \vdash x : C \quad \delta;\gamma \vdash e : C'}{\delta;\gamma \vdash x = e; : \texttt{void}}$$
provided $C' \prec C \ \wedge\ x \neq \texttt{this}$

TS-Return
$$\frac{\delta;\gamma \vdash e : C}{\delta;\gamma \vdash \texttt{return}\ e; : C}$$

TS-Block
$$\frac{\delta;\gamma \vdash s_1 \ldots s_n : \texttt{void}}{\delta;\gamma \vdash \{s_1 \ldots s_n\} : \texttt{void}}$$

TS-Intro
$$\frac{\delta;\gamma, x : C \vdash s_1 \ldots s_n : \tau}{\delta;\gamma \vdash C\ x; s_1 \ldots s_n : \tau}$$
provided $x \notin dom(\gamma)$

TS-Seq
$$\frac{\delta;\gamma \vdash s_1 : \texttt{void} \quad \delta;\gamma \vdash s_2 \ldots s_n : \tau}{\delta;\gamma \vdash s_1\ s_2 \ldots s_n : \tau}$$
provided $s_1 \neq C\ x;$

T-CObject
$$\frac{C \prec_1 \texttt{Object}}{\delta;\gamma, \texttt{this} : C \vdash \texttt{super}() : \texttt{void}}$$

T-CSuper
$$\frac{\delta;\gamma' \vdash \overline{e} : \overline{C'}}{\delta;\gamma \vdash \texttt{super}(\overline{e}); : \texttt{void}}$$
provided $\gamma(\texttt{this}) = C \ \wedge\ \gamma = \gamma' \uplus \{\texttt{this} : C\} \ \wedge\ \delta_{\text{C}}(C') = \overline{C}$
$\wedge\ C \prec_1 C' \quad \wedge\ \overline{C'} \prec \overline{C}$

Figure 2.4: Typing rules for statements

T-MDEFN
$$\frac{\delta; \{\texttt{this} : C, \overline{x} : \overline{C}\} \vdash \overline{s} : \tau}{\delta \vdash mbody(C, m)\ \mathsf{ok}} \quad \text{provided } mbody(C,m) = (\overline{x}, \overline{s}) \ \wedge\ \delta_{\mathrm{M}}(C)(m) = (\overline{C} \to \tau)$$

T-MBODYS
$$\frac{\forall m \in \delta_{\mathrm{M}}(C).\quad \delta \vdash mbody(C,m)\ \mathsf{ok}}{\delta \vdash C\ \mathsf{mok}}$$

T-CDEFN
$$\frac{\delta; \texttt{this} : C, \overline{x} : \overline{C} \vdash \texttt{super}(\overline{e}); : \texttt{void} \quad \delta; \texttt{this} : C, \overline{x} : \overline{C} \vdash \overline{s} : \texttt{void}}{\delta \vdash C\ \mathsf{cok}} \quad \text{provided } \delta_{\mathrm{C}}(C) = \overline{C} \ \wedge\ cnbody(C) = (\overline{x}, \texttt{super}(\overline{e});\ \overline{s})$$

T-PROGDEF
$$\frac{\forall C \in \delta.(\quad \delta \vdash C\ \mathsf{cok} \quad \delta \vdash C\ \mathsf{mok}\quad)}{\delta \vdash \pi\ \mathsf{ok}}$$

Figure 2.5: Typing MJ programs

Method Body
$$mbody(C,m) \stackrel{\text{def}}{=} \begin{cases} (\overline{x}, \overline{s}) & md_i = C''m(\overline{C}\overline{x})\{\overline{s}\} \\ mbody(C', m) & m \notin md_1 \ldots md_n \end{cases}$$
where $\texttt{class}\ \ C\ \texttt{extends}\ C'\{\overline{fd}\ cnd\ md_1 \ldots md_n\} \in \pi$

Constructor Body
$$cnbody(C) \stackrel{\text{def}}{=} (\overline{x}, \overline{s})$$
where $\texttt{class}\ \ C\ \texttt{extends}\ C'\{\overline{fd}\ C(\overline{C''}\ \overline{x})\{\overline{s}\}\ \overline{md}\} \in \pi$

We can now formalise the notion of a program being well-typed with respect to its class table. This is denoted by the judgement $\delta \vdash \pi\ \ \mathsf{ok}$. Informally this involves checking that each method body and constructor body is well-typed and that the type deduced matches that contained in δ. We introduce two new judgement forms: $\delta \vdash C\ \mathsf{mok}$ denotes that the methods of class C are well-typed, and $\delta \vdash C\ \mathsf{cok}$ denotes that the constructor method of class C is well-typed. The rules for forming these judgements are given in Figure 2.5.

2.3 Operational semantics

We define the operational semantics of MJ in terms of transitions between *configurations*. A configuration is a four-tuple, containing the following information:

1. **Heap**: A pair of finite partial functions, the first maps object identifiers, *oids*, to class names, and the second maps pairs of oids and fields to values (oids and null).[2]
2. **Variable Stack**: This essentially maps variable names to *oids*. To handle static block-structured scoping it is implemented as a list of lists of partial functions from variables to values. (This is explained in more detail later.) We use $\circ$ to denote stack cons.
3. **Term**: The program (a closed frame) to be evaluated.
4. **Frame stack**: This is essentially the program context in which the term is currently being evaluated.

This is defined formally in Figure 2.6. CF is a closed frame (i.e. with no hole) and OF is an open frame (i.e. requires an expression to be substituted in for the hole). Additionally we have

[2]This is different to the original design of MJ, in [9]. The changes are to streamline the presentation in the later chapters, see §3.2.

Configuration
$config ::= (H, VS, CF, FS)$
$\quad | \quad (H, VS, EF, FS)$
Frame stack
$FS ::= F \circ FS \mid []$
Frame
$F ::= CF \mid OF$
Closed frame
$CF ::= \overline{s} \mid \texttt{return}\ e; \mid \{\ \}$
$\quad | \quad e \mid \texttt{super}(\overline{e});$
Open frame
$OF ::= \texttt{if}\ (\bullet == e)\{\overline{s}\}\ \texttt{else}\ \{\overline{s}\}$
$\quad | \quad \texttt{if}\ (v == \bullet)\{\overline{s}\}\ \texttt{else}\ \{\overline{s}\}$
$\quad | \quad \bullet.f \mid \bullet.f = e;$
$\quad | \quad \mid v.f = \bullet; \mid (C)\bullet$
$\quad | \quad v.m(\overline{v}, \bullet, \overline{e}) \mid \bullet.m(\overline{e})$
$\quad | \quad \texttt{new}\ C(\overline{v}, \bullet, \overline{e})$
$\quad | \quad \texttt{super}(\overline{v}, \bullet, \overline{e});$
$\quad | \quad x = \bullet; \mid \texttt{return}\ \bullet;$

Exception frame
$EF ::= \textbf{NPE} \mid \textbf{CCE}$
Values
$v ::= \texttt{null} \mid o$
Variable stack
$VS ::= MS \circ VS \mid []$
Method scope
$MS ::= BS \circ MS \mid []$
Block scope
$BS ::=$ is a finite partial function from variables to pairs of expression types and values
Heap
$H ::= (H_v, H_t)$
$H_t ::=$ is a finite partial function from *oids* to class names
$H_v ::=$ is a finite partial function from *oids* paired with field names to values

Figure 2.6: Syntax for the MJ operational semantics

frames that correspond to exceptional states, described in §2.3.2. We use $H(o)$ as a shorthand for applying the second function in H to o, and $H(o, f)$ for applying the first to (o, f). We use the obvious shorthands for dom and function update, for example $o \in dom(H)$ means o is in the domain of H's first function.

In the following example we demonstrate how the variable scopes correctly model the block structure scoping of Java. The left hand side gives the source code and the right the contents of the variable stack.

Note: We could use only a single level of list, but this complicates other issues, as we can define a variable to have different types in two different non-nested scopes. Also we could collapse the third and fourth elements of a configuration and consider only non-empty stacks: we would simply be replacing a comma with a cons.

```
1: B m(A a) {                 ← VS
2:   B r;                     ← ({a ↦ (A, v1)} ∘ []) ∘ VS
3:   if(this.x == a){         ← ({r ↦ (B, null), a ↦ (A, v1)} ∘ []) ∘ VS
4:     r = a;
5:   } else {                 ← ({} ∘ {r ↦ (B, null), a ↦ (A, v1)} ∘ []) ∘ VS
6:     A t;                   ← ({t ↦ (A, null)} ∘ {r ↦ (B, null), a ↦ (A, v1)} ∘ []) ∘ VS
7:     t = a.m1();            ← ({t ↦ (A, v3} ∘ {r ↦ (B, null), a ↦ (A, v2)} ∘ []) ∘ VS
8:     r = a.m2(t,t);         ← ({t ↦ (A, v3} ∘ {r ↦ (B, v4), a ↦ (A, v2)} ∘ []) ∘ VS
9:   }                        ← ({r ↦ (B, v4), a ↦ (A, v2)} ∘ []) ∘ VS
10:  return r;
11: }                         ← VS
```

Before calling this method, let us assume we have a variable scope, VS. A method call should not affect any variables in the current scopes, so we create a new method scope, $\{\texttt{a} \mapsto (\texttt{A}, v_1)\} \circ []$, on entry to the method. This scope consists of a single block scope that points the argument `a` at the value v_1 with a type annotation of `A`. On line 2 the block scope is extended to contain the new variable `r`. On line 5 we assumed that `this.x` $\neq$ `a` and enter into a new block. This has the effect to adding a new empty block scope, $\{\}$, into the current method scope. On line 6 this new scope is extended to contain the variable `t`. Notice how it is added to the current

top scope, as was the variable `r`. Updates can, however, occur to block scopes anywhere in the current method scope. This can be seen on line 8 where `r` is updated in the enclosing scope. On line 9 the block scope is disposed of, and hence `t` cannot be accessed by the statement on line 10.

We find it useful to define two operations on a variable stack, VS, in addition to the usual list operations. The first, $VS(x)$, evaluates a variable, x, in the top method scope, MS (in the top list of block scopes). This is a partial function and is only defined if the variable name is in the scope. The second, $VS[x \mapsto v]$, updates the top method scope, MS, with the value v for the variable x. Again this is a partial function and is undefined if the variable is not in the scope.

$$
\begin{array}{rcl}
((BS \circ MS) \circ VS)(x) & \stackrel{\text{def}}{=} & \begin{cases} BS(x) & x \in dom(BS) \\ (MS \circ VS)(x) & \text{otherwise} \end{cases} \\
([] \circ VS)(x) & \stackrel{\text{def}}{=} & [](x) \stackrel{\text{def}}{=} \text{ undefined} \\
((BS \circ MS) \circ VS)[x \mapsto v] & \stackrel{\text{def}}{=} & \begin{cases} (BS[x \mapsto (v, C)] \circ MS) \circ VS & BS(x) = (\_, C) \\ BS \circ ((MS \circ VS)[x \mapsto v]) & \text{otherwise} \end{cases} \\
([] \circ VS)[x \mapsto v] & \stackrel{\text{def}}{=} & []\,[x \mapsto v] \stackrel{\text{def}}{=} \text{ undefined}
\end{array}
$$

where we treat

$$
\begin{array}{rcl}
BS \circ (MS \circ VS) & = & (BS \circ MS) \circ VS \\
BS \circ \text{undefined} & = & \text{undefined}
\end{array}
$$

2.3.1 Reductions

This section defines the transition rules that correspond to meaningful computation steps. In spite of the complexity of MJ, there are only seventeen rules, one for each syntactic constructor.

We begin by giving the transition rules for accessing, assigning values to, and declaring variables. Notice that the side condition in the E-VARWRITE rule ensures that we can only write to variables declared in the current method scope. The E-VARINTRO rule follows Java's restriction that a variable declaration *cannot* hide an earlier declaration within the current method scope.[3] (Note also how the rule defines the binding for the new variable in the current *block* scope.)

E-VARACCESS
$$(H, VS, x, FS) \rightarrow (H, VS, v, FS) \quad \text{provided } VS(x) = v$$

E-VARWRITE
$$(H, VS, x = v;, FS) \rightarrow (H, VS', ;, FS) \quad \text{provided } VS' = VS[x \mapsto v]$$

E-VARINTRO
$$(H, (BS \circ MS) \circ VS, C\ x;, FS) \rightarrow (H, (BS' \circ MS) \circ VS, ;, FS) \quad \text{provided } x \notin dom(BS \circ MS) \wedge\ BS' = BS[x \mapsto (\texttt{null}, C)]$$

Now we consider the rules for constructing and removing scopes. The first rule E-BLOCKINTRO introduces a new block scope, and leaves a 'marker' token, $\{\}$, on the frame stack. The second E-BLOCKELIM removes the token and the top block scope. The final rule E-RETURN leaves the scope of a method, by removing the top scope, MS.

[3] This sort of variable hiding is, in contrast, common in functional languages such as SML.

E-BLOCKINTRO	$(H, MS \circ VS, \{\overline{s}\}, FS) \rightarrow (H, (\{\} \circ MS) \circ VS, \overline{s}, (\{\ \}) \circ FS)$
E-BLOCKELIM	$(H, (BS \circ MS) \circ VS, \{\ \}, FS) \rightarrow (H, MS \circ VS, ;, FS)$
E-RETURN	$(H, MS \circ VS, \texttt{return}\ v;, FS) \rightarrow (H, VS, v, FS)$

Next we give the transition rules for the conditional expression. One should note that the resulting term of the transition is a block.

E-IF1	$(H, VS, (\texttt{if}\ (v_1 == v_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};), FS) \rightarrow (H, VS, \{\overline{s_1}\}, FS)$ provided $v_1 = v_2$
E-IF2	$(H, VS, (\texttt{if}\ (v_1 == v_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};), FS) \rightarrow (H, VS, \{\overline{s_2}\}, FS)$ provided $v_1 \neq v_2$

Next we consider the rules dealing with objects. First let us define the transition rule dealing with field access and assignment, as they are reasonably straightforward.

E-FIELDACCESS	$(H, VS, o.f, FS) \rightarrow (H, VS, v, FS)$ provided $(o, f) \in dom(H)\ \wedge\ H(o, f) = v$
E-FIELDWRITE	$(H, VS, o.f = v;, FS) \rightarrow (H', VS, ;, FS)$ provided $(o, f) \in dom(H)\ \wedge\ H' = H[(o, f) \mapsto v]$

Now we consider the rules dealing with casting of objects. Rule E-CAST simply ensures that the cast is valid (if it is not, the program should enter an error state—this is covered in §2.3.2). Rule E-NULLCAST simply ignores any cast of a `null` object reference.

E-CAST	$(H, VS, ((C_1)o), FS) \rightarrow (H, VS, o, FS)$ provided $H(o) = C_2\ \wedge\ C_2 \prec C_1$
E-NULLCAST	$(H, VS, ((C)\texttt{null}), FS) \rightarrow (H, VS, \texttt{null}, FS)$

Let us consider the transition rule involving the creation of objects. The E-NEW rule creates a fresh *oid*, o, and extends the heap with the new heap object consisting of the class C and sets all the fields to `null`. As we are executing a new method (the constructor), a new method scope is created and added on to the variable stack. This method scope initially consists of just one block scope, that consists of bindings for the method parameters, and also a binding for the `this` identifier. The method body $\overline{s}$ is then the next term to be executed, but importantly the continuation `return` o; is placed on the frame stack. This is because the result of this statement is the *oid* of the object, and the method scope is removed.

E-NEW	$(H, VS, \texttt{new}\ C(\overline{v}), FS) \rightarrow (H', (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$ provided $H' = H[o \mapsto C][(o, \overline{f}) \mapsto \texttt{null}]\ \wedge\ \delta_{\mathrm{C}}(C) = \overline{C}\ \wedge\ cnbody(C) = (\overline{x}, \overline{s})\ \wedge\ o \notin dom(H)\ \wedge\ BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}$

Next we consider the transition rule for the `super` statement, which occurs inside constructor methods. This rule is basically the same as the previous, without the construction of the new object in the heap. It inspects the stack to determine the type of `this`, and hence which constuctor body the `super` call refers to.

E-SUPER $(H, VS, \texttt{super}(\overline{v}), FS) \rightarrow (H, (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$
provided $VS(\texttt{this}) = (o, C) \ \wedge\ C \prec_1 C'$
$\wedge\ BS = \{\texttt{this} \mapsto (o, C'), \overline{x} \mapsto (\overline{v}, \overline{C})\}\ \wedge\ \delta_{\mathrm{C}}(C) = \overline{C}$
$\wedge\ cnbody(C') = (\overline{x}, \overline{s})$

Next we can give the transition rule for method invocation. Invocation is relatively straightforward: note how a new method scope is created, consisting of just the bindings for the method parameters and the `this` identifier. The reader should note the method body is selected by the dynamic, or run-time, type of the object, not its static type. We require two rules as a method returning a `void` type requires an addition to the stack to clear the new method scope once the method has completed. Recall that in E-METHOD rule, the last statement in the sequence $\overline{s}$ will be a `return` if the method is well typed.

E-METHOD $(H, VS, o.m(\overline{v}), FS) \rightarrow (H, (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ e;) \circ FS)$
provided $mbody(C, m) = (\overline{x}, \overline{s}\ \texttt{return}\ e;) \ \wedge\ H(o) = C$
$\wedge\ BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}\ \wedge\ \delta_{\mathrm{M}}(C)(m) = \overline{C} \rightarrow C'$

E-METHODVOID $(H, VS, o.m(\overline{v}), FS) \rightarrow (H, (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$
provided $H(o) = C \ \wedge\ \delta_{\mathrm{M}}(C)(m) = \overline{C} \rightarrow \texttt{void}$
$\wedge\ mbody(C, m) = (\overline{x}, \overline{s})\ \wedge\ BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}$

Finally we have two reductions rules for fully reduced terms. The first rule deals with completed statements, and the second for evaluated expressions.

E-SKIP $(H, VS, ;, F \circ FS) \rightarrow (H, VS, F, FS)$

E-SUB $(H, VS, v, F \circ FS) \rightarrow (H, VS, F[v], FS)$

To assist the reader, all the reduction rules are repeated in full in Figure 2.7. There are a number of other reduction rules, which simply decompose terms. These rules essentially embody the order of evaluation, and are given in Figure 2.8.

2.3.2 Error states

A number of expressions will lead us to a predictable error state. These are errors that are allowed at run-time as they are dynamically checked for by the Java Virtual Machine. Java's type system is not capable of removing these errors statically. The two errors that can be generated, in MJ, are `NullPointerException`, written here **NPE** [41, §15.11.1,§15.12.4.4], and `ClassCastException` [41, §5.5], **CCE**.

E-NULLFIELD $(H, VS, \texttt{null}.f, FS) \rightarrow (H, VS, \textbf{NPE}, FS)$

E-NULLWRITE $(H, VS, \texttt{null}.f = v, FS) \rightarrow (H, VS, \textbf{NPE}, FS)$

E-NULLMETHOD $(H, VS, \texttt{null}.m(v_1, \ldots, v_n), FS) \rightarrow (H, VS, \textbf{NPE}, FS)$

E-INVCAST $(H, VS, (C)o, FS) \rightarrow (H, VS, \textbf{CCE}, FS)$
provided $H(o) = (C', \mathbb{F}) \ \wedge\ C' \not\prec C$

Definition 2.3.1 (Terminal configuration)**.** A configuration is said to be *completed* if it is of the form $(H, VS, v, [])$ or $(H, VS, ;, [])$, and is an *exception* if it is of the form (H, VS, EF, FS). A configuration is said to be *terminal* if it is completed or an exception. We call a configuration *stuck* if it cannot reduce, that is $config$ is stuck iff $\neg\exists config'.config \rightarrow config'$.

E-VARACCESS $(H, VS, x, FS) \to (H, VS, v, FS)$
provided $VS(x) = v$

E-VARWRITE $(H, VS, x = v;, FS) \to (H, VS', ;, FS)$
provided $VS' = VS[x \mapsto v]$

E-VARINTRO $(H, (BS \circ MS) \circ VS, C\ x;, FS) \to (H, (BS' \circ MS) \circ VS, ;, FS)$
provided $x \notin dom(BS \circ MS) \ \wedge\ BS' = BS[x \mapsto (\texttt{null}, C)]$

E-BLOCKINTRO $(H, MS \circ VS, \{\overline{s}\}, FS) \to (H, (\{\} \circ MS) \circ VS, \overline{s}, (\{\ \}) \circ FS)$

E-BLOCKELIM $(H, (BS \circ MS) \circ VS, \{\ \}, FS) \to (H, MS \circ VS, ;, FS)$

E-RETURN $(H, MS \circ VS, \texttt{return}\ v;, FS) \to (H, VS, v, FS)$

E-IF1 $(H, VS, (\texttt{if}\ (v_1 == v_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};), FS) \to (H, VS, \{\overline{s_1}\}, FS)$
provided $v_1 = v_2$

E-IF2 $(H, VS, (\texttt{if}\ (v_1 == v_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};), FS) \to (H, VS, \{\overline{s_2}\}, FS)$
provided $v_1 \neq v_2$

E-FIELDACCESS $(H, VS, o.f, FS) \to (H, VS, v, FS)$
provided $(o, f) \in dom(H) \ \wedge\ H(o, f) = v$

E-FIELDWRITE $(H, VS, o.f = v;, FS) \to (H', VS, ;, FS)$
provided $(o, f) \in dom(H) \ \wedge\ H' = H[(o, f) \mapsto v]$

E-CAST $(H, VS, ((C_1)o), FS) \to (H, VS, o, FS)$
provided $H(o) = C_2 \ \wedge\ C_2 \prec C_1$

E-NULLCAST $(H, VS, ((C)\texttt{null}), FS) \to (H, VS, \texttt{null}, FS)$

E-NEW $(H, VS, \texttt{new}\ C(\overline{v}), FS) \to (H', (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$
provided $H' = H[o \mapsto C][(o, \overline{f}) \mapsto \texttt{null}] \ \wedge\ \delta_{\mathrm{C}}(C) = \overline{C}$
$\wedge\ cnbody(C) = (\overline{x}, \overline{s}) \ \wedge\ o \notin dom(H)$
$\wedge\ BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}$

E-SUPER $(H, VS, \texttt{super}(\overline{v}), FS) \to (H, (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$
provided $VS(\texttt{this}) = (o, C) \ \wedge\ C \prec_1 C'$
$\wedge\ BS = \{\texttt{this} \mapsto (o, C'), \overline{x} \mapsto (\overline{v}, \overline{C})\} \ \wedge\ \delta_{\mathrm{C}}(C) = \overline{C}$
$\wedge\ cnbody(C') = (\overline{x}, \overline{s})$

E-METHOD $(H, VS, o.m(\overline{v}), FS) \to (H, (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ e;) \circ FS)$
provided $mbody(C, m) = (\overline{x}, \overline{s}\ \texttt{return}\ e;) \ \wedge\ H(o) = C$
$\wedge\ BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\} \ \wedge\ \delta_{\mathrm{M}}(C)(m) = \overline{C} \to C'$

E-METHODVOID $(H, VS, o.m(\overline{v}), FS) \to (H, (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$
provided $H(o) = C \ \wedge\ \delta_{\mathrm{M}}(C)(m) = \overline{C} \to \texttt{void}$
$\wedge\ mbody(C, m) = (\overline{x}, \overline{s}) \ \wedge\ BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}$

E-SKIP $(H, VS, ;, F \circ FS) \to (H, VS, F, FS)$

E-SUB $(H, VS, v, F \circ FS) \to (H, VS, F[v], FS)$

Figure 2.7: MJ reduction rules

EC-SEQ	$(H, VS, s_1\ s_2 \ldots s_n, FS) \rightarrow (H, VS, s_1, (s_2 \ldots s_n) \circ FS)$
EC-RETURN	$(H, MS \circ VS, \texttt{return}\ e;, FS) \rightarrow (H, MS \circ VS, e, (\texttt{return}\ \bullet;) \circ FS)$
EC-EXPSTATE	$(H, VS, e;, FS) \rightarrow (H, VS, e, FS)$
EC-IF1	$(H, VS, \texttt{if}\ (e_1 == e_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};, FS) \rightarrow (H, VS, e_1, (\texttt{if}\ (\bullet == e_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};) \circ FS)$
EC-IF2	$(H, VS, \texttt{if}\ (v_1 == e_2)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};, FS) \rightarrow (H, VS, e_2, (\texttt{if}\ (v_1 == \bullet)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};) \circ FS)$
EC-FIELDACCESS	$(H, VS, e.f, FS) \rightarrow (H, VS, e, (\bullet.f) \circ FS)$
EC-CAST	$(H, VS, (C)e, FS) \rightarrow (H, VS, e, ((C)\bullet) \circ FS)$
EC-FIELDWRITE1	$(H, VS, e_1.f = e_2;, FS) \rightarrow (H, VS, e_1, (\bullet.f = e_2;) \circ FS)$
EC-FIELDWRITE2	$(H, VS, v_1.f = e_2;, FS) \rightarrow (H, VS, e_2, (v_1.f = \bullet;) \circ FS)$
EC-VARWRITE	$(H, VS, x = e;, FS) \rightarrow (H, VS, e, (x = \bullet;) \circ FS)$
EC-NEW	$(H, VS, \texttt{new}\ C(\overline{v}, e, \overline{e}), FS) \rightarrow (H, VS, e, (\texttt{new}\ C(\overline{v}, \bullet, \overline{e})) \circ FS)$
EC-SUPER	$(H, VS, \texttt{super}(\overline{v}, e, \overline{e}), FS) \rightarrow (H, VS, e, (\texttt{super}(\overline{v}, \bullet, \overline{e})) \circ FS)$
EC-METHOD1	$(H, VS, e.m(\overline{e}), FS) \rightarrow (H, e, (\bullet.m(\overline{e})) \circ FS)$
EC-METHOD2	$(H, VS, v.m(\overline{v}, e, \overline{e}), FS) \rightarrow (H, VS, e, (v.m(\overline{v}, \bullet, \overline{e})) \circ FS)$

Figure 2.8: MJ syntax decomposition reduction rules

2.3.3 Example execution

To help the reader understand our operational semantics, in this section we will consider a simple code fragment and see how the operational semantics captures its dynamic behaviour.

Consider the following MJ code whose effect is to swap the contents of two variables `var1` and `var2`, using a temporary variable `t`.

```
if(var1 == var2) {
  ;
} else {
  Object t;
  t = var1;
  var1 = var2;
  var2 = t;
}
```

We consider its execution in a configuration where MS maps `var1` to a value, say $v1$, and `var2` to a value, say $v2$.

$$
\begin{array}{ll}
 & (H, MS \circ VS, \texttt{if}\ (\texttt{var1} == \texttt{var2})\{;\}\ \texttt{else}\ \{\ldots\}, FS) \\
\rightarrow & (H, MS \circ VS, \texttt{var1}, (\texttt{if}\ (\bullet == \texttt{var2})\{;\}\ \texttt{else}\ \{\ldots\}) \circ FS) \\
\rightarrow & (H, MS \circ VS, v1, (\texttt{if}\ (\bullet == \texttt{var2})\{;\}\ \texttt{else}\ \{\ldots\}) \circ FS) \\
\rightarrow & (H, MS \circ VS, (\texttt{if}\ (v1 == \texttt{var2})\{;\}\ \texttt{else}\ \{\ldots\}), FS) \\
\rightarrow & (H, MS \circ VS, \texttt{var2}, (\texttt{if}\ (v1 == \bullet)\{;\}\ \texttt{else}\ \{\ldots\}) \circ FS) \\
\rightarrow & (H, MS \circ VS, v2, (\texttt{if}\ (v1 == \bullet)\{;\}\ \texttt{else}\ \{\ldots\}) \circ FS) \\
\rightarrow & (H, MS \circ VS, (\texttt{if}\ (v1 == v2)\{;\}\ \texttt{else}\ \{\ldots\}), FS)
\end{array}
$$

At this point there are two possibilities: we will consider the case where $v1 \neq v2$.

Note: For compactness, we use `O` for `Object`.

$\rightarrow (H, (\{\}) \circ MS \circ VS, \circ\ \texttt{t}; \ldots, (\{\}) \circ FS)$
$\rightarrow (H, (\{\}) \circ MS \circ VS, \circ\ \texttt{t}; , (\texttt{t = var1}; \ldots) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (\texttt{null}, \circ\}) \circ MS \circ VS, ; , (\texttt{t = var1}; \ldots) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (\texttt{null}, \circ)\}) \circ MS \circ VS, (\texttt{t = var1}; \ldots), (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (\texttt{null}, \circ)\}) \circ MS \circ VS, \texttt{t = var1}; , (\texttt{var1 = var2}; \ldots) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (\texttt{null}, \circ)\}) \circ MS \circ VS, \texttt{var1}, (\texttt{t} = \bullet;) \circ (\texttt{var1 = var2}; \ldots) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (\texttt{null}, \circ)\}) \circ MS \circ VS, v1, (\texttt{t} = \bullet;) \circ (\texttt{var1 = var2}; \ldots) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (\texttt{null}, \circ)\}) \circ MS \circ VS, \texttt{t} = v1; , (\texttt{var1 = var2}; \ldots) \circ (\{\}) \circ FS)$

At this point we update the variable stack; note how the update does not change the type.

$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS \circ VS, ; , (\texttt{var1 = var2}; \ldots) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS \circ VS, \texttt{var1 = var2}; \ldots, (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS \circ VS, \texttt{var1 = var2}; , (\texttt{var2 = t};) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS \circ VS, \texttt{var2}, \texttt{var1} = \bullet; \circ (\texttt{var2 = t};) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS \circ VS, v2, \texttt{var1} = \bullet; \circ (\texttt{var2 = t};) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS \circ VS, \texttt{var1} = v2; , (\texttt{var2 = t};) \circ (\{\}) \circ FS)$

Let MS' be a variable scope which is the same as MS except that `var1` is mapped to $v2$ instead of $v1$.

$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS' \circ VS, ; , (\texttt{var2 = t};) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS' \circ VS, \texttt{var2 = t}; , (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS' \circ VS, \texttt{t}, (\texttt{var2} = \bullet;) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS' \circ VS, v1, (\texttt{var2} = \bullet;) \circ (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS' \circ VS, \texttt{var2} = v1; , (\{\}) \circ FS)$

Let MS'' be a variable scope which is the same as MS' except that `var2` is mapped to $v1$.
$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS'' \circ VS, ; , (\{\}) \circ FS)$
$\rightarrow (H, (\{\texttt{t} \mapsto (v1, \circ)\}) \circ MS'' \circ VS, \{\}, FS)$
$\rightarrow (H, MS'' \circ VS, ; , FS)$

At this point the execution of the `if` statement has been completed and its temporary variable, `temp` has been removed from the scope. The variable stack has had the values of `var1` and `var2` correctly swapped.

2.4 Well-typed configuration

To prove type soundness for MJ, we need to extend our typing rules from expressions and statements to configurations. We write $\delta \vdash (H, VS, CF, FS) : \tau$ to mean (H, VS, CF, FS) is well-typed with respect to δ and will result in a value of type τ (or a valid error state). We break this into three properties, written: $\delta \vdash H$ ok, $\delta, H \vdash VS$ ok and $\delta, H, VS \vdash CF \circ FS : \texttt{void} \rightarrow \tau$.

The first property, written $\delta \vdash H$ ok, ensures that every field points to a valid object or `null`, and all the types mentioned in the heap are in δ.

TH-OBJECTTYPED
$$\frac{H(o) = C \quad C \prec \tau \quad \delta \vdash C \text{ ok}}{\delta, H \vdash o : \tau}$$

TH-NULLTYPED
$$\frac{\delta \vdash C \text{ ok}}{\delta, H \vdash \texttt{null} : C}$$

TH-OBJECTOK
$$\frac{H(o) = (C) \quad \forall f \in dom(\delta_{\text{F}}(C)). \quad \delta, H \vdash H(o, f) : \delta_{\text{F}}(C)(f)}{\delta, H \vdash o \text{ ok}}$$

TH-HEAPOK
$$\frac{\forall o \in dom(H). \quad \delta, H \vdash o \text{ ok}}{\delta \vdash H \text{ ok}}$$

The second property, written $\delta, H \vdash VS$ ok, constrains every variable to be either a valid object identifier, or `null`.

$$\text{TV-EMPTY} \quad \delta, H \vdash [] \text{ ok}$$

$$\text{TV-MSEMPTY} \quad \frac{\delta, H \vdash VS \text{ ok}}{\delta, H \vdash [] \circ VS \text{ ok}}$$

$$\text{TV-STACK} \quad \frac{\delta, H \vdash MS \circ VS \text{ ok} \quad \forall (v, C) \in cod(BS).\ \delta, H \vdash v : C}{\delta, H \vdash (BS \circ MS) \circ VS \text{ ok}}$$

We generalise the final property to type arbitrary framestacks, written $\delta, H, VS \vdash FS : \tau \rightarrow \tau'$. It types each frame in the context formed from the heap and variable stack. This requires us to define a collapsing of the context to form a typing environment. We must extend the typing environment to map, in addition to variables, object identifiers, o, and holes, $\bullet$, to values. The collapsing function is defined as follows.

$$\begin{array}{rcl} heaptype((H_t, H_v)) & \stackrel{\text{def}}{=} & H_t \\ stacktype((BS \circ MS)) & \stackrel{\text{def}}{=} & \{x \mapsto C \mid (x \mapsto C, v) \in BS\} \uplus stacktype(MS) \\ stacktype([] \circ VS) & \stackrel{\text{def}}{=} & \emptyset \\ context(H, MS \circ VS) & \stackrel{\text{def}}{=} & heaptype(H) \uplus stacktype(MS) \end{array}$$

The syntax of expressions in framestacks is extended to contain both object identifiers and holes. Hence we require two additional expression typing rules.

$$\text{TE-OID} \quad \frac{o : C \in \gamma \quad \delta \vdash \gamma \text{ ok} \quad \vdash \delta \text{ ok}}{\delta; \gamma \vdash o : C}$$

$$\text{TE-HOLE} \quad \frac{\bullet : C \in \gamma \quad \delta \vdash \gamma \text{ ok} \quad \vdash \delta \text{ ok}}{\delta; \gamma \vdash \bullet : C}$$

We can now define frame stack typing as shown in Figure 2.9. We have the obvious typing for an empty stack. We require special typing rules for the frames that alter variable scoping: TF-STACKBLOCK, TF-STACKMETHOD, TF-STACKMETHOD2 and TF-STACKINTRO. We also require a rule, TF-SEQUENCE, for unrolling sequences because the sequence can contain items to alter scoping. We then require two rules for typing the rest of the frames; one for frames that require an argument, called TF-STACKOPEN, and one for frames that do not, called TF-STACKCLOSED.

2.5 Type soundness

Our main technical contribution in this chapter is a proof of type soundness of the MJ type system. In order to prove correctness we first prove two useful propositions in the style of Wright and Felleisen[86]. The first proposition states that any well typed non-terminal configuration can make a reduction step.

Proposition 2.5.1 (Progress)**.** *If $config$ is not terminal and $\delta \vdash config : \tau$ then $\exists config'.config \rightarrow config'$.*

Proof. By case analysis of CF. By considering all well typed configurations, we can show how each term can reduce.

TF-STACKEMPTY
$$\delta, H, (BS \circ []) \circ [] \vdash [] : \tau \to \tau$$

TF-ERROR
$$\delta, H, VS \vdash EF \circ FS : \tau \to \tau'$$

TF-STACKBLOCK
$$\frac{\delta, H, MS \circ VS \vdash FS : \texttt{void} \to \tau}{\delta, H, (BS \circ MS) \circ VS \vdash (\{\}) \circ FS : \tau' \to \tau}$$

TF-STACKMETHOD
$$\frac{\delta, H, VS \vdash FS : \tau \to \tau'}{\delta, H, (BS \circ []) \circ VS \vdash (\texttt{return}\ \bullet;) \circ FS : \tau \to \tau'}$$

TF-STACKMETHOD2
$$\frac{\delta; context(H, MS \circ VS) \vdash e : \tau \quad H, VS \vdash FS : \tau \to \tau'}{\delta, H, MS \circ VS \vdash (\texttt{return}\ e;) \circ FS : \tau'' \to \tau'}$$

TF-STACKINTRO
$$\frac{\delta, H, (BS[x \mapsto (\texttt{null}, C)] \circ MS) \circ VS \vdash FS : \texttt{void} \to \tau}{H, (BS \circ MS) \circ VS \vdash (C\ x;) \circ FS : \tau' \to \tau}$$
provided $x \notin dom(BS \circ MS)$

TF-SEQUENCE
$$\frac{\delta, H, VS \vdash (s_1) \circ (s_2 \ldots s_n) \circ FS : \tau \to \tau'}{\delta, H, VS \vdash (s_1 s_2 \ldots s_n) \circ FS : \tau \to \tau'}$$

TF-STACKOPEN
$$\frac{\delta; context(H, VS), \bullet : C \vdash OF : \tau \quad H, VS \vdash FS : \tau \to \tau'}{\delta, H, VS \vdash OF \circ FS : C \to \tau'}$$
provided $OF \neq (\texttt{return}\ \bullet;)$

TF-STACKCLOSED
$$\frac{\delta; context(H, VS) \vdash CF : \tau \quad H, VS \vdash FS : \tau \to \tau'}{\delta, H, VS \vdash CF \circ FS : \tau'' \to \tau'}$$
provided $CF \neq (\texttt{return}\ e;) \ \wedge\ CF \neq (\{\}) \ \wedge\ CF \neq s_1 \ldots s_n \wedge n > 1 \ \wedge\ CF \neq C\ x$

Figure 2.9: Frame stack typing

Case: $CF = (\textbf{return}\ e;)$ As it is well typed we know $VS = MS \circ VS'$. This has two possible cases of reducing. Either $e = v$ and can reduce by E-RETURN or $e \neq v$ and can reduce by EC-RETURN.

Case: $CF = (\{\})$ As this is well typed, we know $VS = (BS \circ MS) \circ VS'$ and hence this can reduce by E-BLOCK.

Case: $CF = C\ x;$ As it is well typed, we know $VS = (BS \circ MS) \circ VS'$ and also $x \notin context(VS)$. From the definition of context we can see that this gives $x \notin dom(BS \circ MS)$ and hence it can reduce by E-VARINTRO.

Case: $CF = x$ We know that x must be in $context(H, VS)$, and as x cannot be in H it must come from VS and more precisely in MS, where $VS = MS \circ VS'$, hence it can reduce by E-VARACCESS.

Case: $CF = o$ Either $FS \neq []$ and reduces by E-SUB, or $FS = []$ which is a terminal framestack.

Case: $CF = \textbf{null}$ Either $FS \neq []$ and reduces by E-SUB, or $FS = []$ which is a terminal framestack.

Case: $CF = e.f$ This can be broken into three cases. Either $e = \texttt{null}$ and reduces by E-NULLFIELD; $e = o$ and reduces by E-FIELDACCESS; or $e \neq v$ and reduces by EC-FIELDACCESS.

Case: $CF = e'.m(e_1, \ldots, e_n)$ If $e' \neq v$ then EC-METHOD1 will apply. If any of $e_1, \ldots, e_n$ are not values then EC-METHOD2 will be applied. Otherwise we have the case where $CF = v'.m(v_1, \ldots, v_n)$ in which case E-METHOD applies if $v' = o$ or E-NULLMETHOD if $v' = \texttt{null}$.

Case: $CF = \textbf{new}\ C(e_1, \ldots, e_n)$ reduces by EC-NEW when $e_i \neq v$ or E-NEW otherwise.

Case: $CF = (C)e$ Either $e \neq v$ and reduces by EC-CAST; $e = o$ and reduces by either E-CAST or

E-INVCAST depending on the type of o; or $e = \texttt{null}$ and reduces by E-NULLCAST.

Case: $CF =;$ Either $FS \neq []$ and reduces by E-SKIP; or $FS = []$ which is a terminal state.

Case: $CF = s_1 \ldots s_n$ reduces by EC-SEQ.

Case: $CF = \texttt{if}\ (e_1 \texttt{==} e_2)\{\overline{s}_1\}\ \texttt{else}\ \{\overline{s}_2\}$. Either $e_1 \neq v_1$ and reduces by EC-IF1; $e_2 \neq v_2 \wedge e_1 = v_1$ and reduces by EC-IF2; or $e_1 = v_1 \wedge e_2 = v_2$ can reduce by either E-IF1 and E-IF2 depending on the test .

Case: $CF = x \texttt{ = } e$ Either $e = v$ and can reduce using E-VARWRITE if x is in the environment. The typing rule tells us that $context(H, VS) \vdash x : C$, hence it is in the environment. Otherwise $e \neq v$ which can reduce by EC-VARWRITE.

Case: $CF = e.f \texttt{ = } e'$ Either $e \neq v$ and reduces using EC-FIELDWRITE1; $e = v \wedge e' \neq v'$ reduces using EC-FIELDWRITE2; $e = o \wedge e' = v'$ and reduces using E-FIELDWRITE; or $e = \texttt{null} \wedge e' = v'$ reduces using E-NULLWRITE.

Case: $CF = e;$ reduces by EC-EXPSTATE.

Case: $CF = \texttt{super}(e_1, \ldots, e_n)$ If all the expressions are values then this can reduce by E-SUPER, otherwise it can reduce by EC-SUPER.

□

Next we find it useful to prove the following few lemmas. The first states that subtyping on frame stacks is *covariant*.

Lemma 2.5.2 (Covariant subtyping of frame stack)**.** *if* $\delta, H, VS \vdash FS : \tau_1 \rightarrow \tau_2$ *and* $\tau_3 \prec \tau_1$ *then* $\exists \tau_4.H, VS \vdash FS : \tau_3 \rightarrow \tau_4$ *and* $\tau_4 \prec \tau_2$.

Proof. By induction on the size of FS. The base case, $FS = []$, holds trivially as TF-STACKEMPTY is covariant. For the inductive step we must show that if FS is covariant, then so is $F \circ FS$. If F is closed then trivially holds as it ignores the argument. We only need consider open frames. First consider `return` $\bullet$; as this affects the typing environment. Holds because it is covariantly typed if the remainder of the frame stack is covariant. The remaining open frames require us to prove:

$$\delta; \gamma, \bullet : \tau \vdash OF : \tau_1 \wedge \tau_2 \prec \tau \quad \Rightarrow \quad \exists \tau_3.\delta; \gamma, \bullet : \tau_2 \vdash OF : \tau_3 \wedge \tau_3 \prec \tau_1$$

We proceed by case analysis on OF.

Case: $OF = \texttt{if}\ (\bullet \texttt{ == } e)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};$ Holds, because e, $\overline{s_1}$ and $\overline{s_2}$ do not contain $\bullet$ they are unaffected by the change in type and using TS-IF or TS-STUPIDIF the statement is well-typed if τ_2 is a valid type.

Case: $OF = \texttt{if}\ (v \texttt{ == } \bullet)\{\overline{s_1}\}\ \texttt{else}\ \{\overline{s_2}\};$ Similar to previous case.

Case: $OF = \bullet.f$ Holds as $\tau_2 \prec \tau$ and field types cannot be overridden, hence we know $\delta_{\mathrm{F}}(\tau_2)(f) = \delta_{\mathrm{F}}(\tau)(f)$.

Case: $OF = \bullet.f \texttt{ = } e;$ Similar to previous case.

Case: $OF = \bullet.m(e_1, \ldots, e_n)$ Similar to previous case, except we use the fact method types cannot be overridden.

Case: $OF = x \texttt{ = } \bullet;$ We know from the assumptions that

$$\frac{\overline{\delta; \gamma, \bullet : \tau \vdash x : C} \qquad \overline{\delta; \gamma, \bullet : \tau \vdash \bullet : \tau} \qquad \overline{\tau \prec C}}{\delta; \gamma, \bullet : \tau \vdash x = \bullet; : \texttt{void}}$$

As the sub-typing relation is transitive and $\tau_2 \prec \tau$, we can see this is well typed for $\bullet : \tau_2$, and the result type is the same.

Case: $OF = v.f \texttt{ = } \bullet;$ Similar to previous case.

Case: $OF = v.m(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n)$ Similar to previous case.

Case: $OF = \texttt{new}\ C(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n)$ Similar to previous case.

Case: $OF = \texttt{super}(v_1, \ldots, v_{i-1}, \bullet, e_{i+1}, \ldots, e_n)$ Similar to previous case.

Case: $OF = (C)\bullet$ The result type of this frame does not depend on the argument type. The three possible typing rules for this case combined to only require $\bullet$ is typeable. Hence this case is trivial.

□

Next we prove that adding a new heap object preserves the typing.

Lemma 2.5.3 (Heap extension preserves typing)**.** *If $\delta, H, VS \vdash FS : \tau \to \tau'$, $o \notin dom(H)$ and $\delta \vdash C$ and $\delta, H \vdash VS$ ok then (1) $\delta, H[o \mapsto C][o, \overline{f} \mapsto \texttt{null}], VS \vdash FS : \tau \to \tau'$ and (2) $\delta, H[o \mapsto C][o, \overline{f} \mapsto \texttt{null}] \vdash VS$ ok.*

Proof. First we prove (1) by induction on the length of FS. All the rules for typing the frame stack follow by induction, except for TF-OPENFRAME, TF-CLOSEDFRAME and TF-METHOD2. To prove these it suffices to prove

$$\delta; \gamma \vdash F : \tau \quad \wedge \quad \delta \vdash C \quad \Rightarrow \quad \delta; \gamma, o : C \vdash F : \tau$$

where $o \notin dom(\gamma)$.

We can see $\delta \vdash \gamma$ ok $\quad \wedge \quad \delta \vdash C \quad \Rightarrow \quad \delta \vdash \gamma, o : C$ ok from the definition of γ ok. We can use this to prove all the axioms of the typing judgement. The rules follow directly by induction except for TS-INTRO. For this rule, we must prove that x and o are different. This is true as they are from different syntactic categories.

To prove (2) we induct over the structure of VS. We require the following

$$\delta, H \vdash v : \tau \Rightarrow \delta, H[o \mapsto C][o, \overline{f} \mapsto \texttt{null}] \vdash v : \tau$$

which holds directly from the definitions. □

We require two lemmas for dealing with typing sequences when they are added to the frame stack.

Lemma 2.5.4 (Block)**.** *If $\delta, H, MS \circ VS \vdash FS : \texttt{void} \to \tau$ and $\delta; context(H, (BS \circ MS) \circ VS) \vdash \overline{s} : \texttt{void}$, then $\delta, H, (BS \circ MS) \circ VS \vdash \overline{s} \circ \{\} \circ FS : \texttt{void} \to \tau$*

Lemma 2.5.5 (Return)**.** *If $\delta, H, VS \vdash FS : \tau \to \tau'$ and $\delta; context(H, MS \circ VS) \vdash s_1 \ldots s_n : \tau$, then $\delta, H, MS \circ VS \vdash s_1 \circ \ldots \circ s_n \circ FS : \texttt{void} \to \tau'$ where s_n is of the form $\texttt{return}\ e$;*

The two lemmas have very similar proofs. We present the proof for the first lemma.

Proof. We prove this by induction on the size of $\overline{s}$.

Base case: Assume $\delta, H, MS \circ VS \vdash FS : \texttt{void} \to \tau$ and $\delta; context(H, (BS \circ MS) \circ VS) \vdash s : \texttt{void}$ by TF-CLOSEDFRAME and TF-BLOCK we can derive $\delta, H, (BS \circ MS) \circ VS \vdash s \circ \{\} \circ FS : \texttt{void} \to \tau$ as required.

Inductive case: Assume property holds for $\overline{s}$ and show it holds for $s_1\overline{s}$. Assume:

$$\delta, H, MS \circ VS \vdash FS : \texttt{void} \to \tau \tag{2.1}$$

$$\delta; context(H, (BS' \circ MS) \circ VS) \vdash s_1\overline{s} \tag{2.2}$$

Prove:

$$\delta, H, (BS' \circ MS) \circ VS \vdash s_1\overline{s} \circ \{\} \circ FS : \texttt{void} \to \tau \tag{2.3}$$

We proceed by a case analysis of s_1.

Case: $s_1 = Cx$; From (2.2) we can deduce $\delta; context(H, (BS'[x \mapsto (C, \texttt{null})] \circ MS) \circ VS) \vdash \overline{s} : \texttt{void}$. Using this and specialising the the inductive hypothesis allows us to deduce $\delta, H, (BS'[x \mapsto (C, \texttt{null})] \circ MS) \circ VS \vdash \overline{s} \circ \{\} \circ FS : \texttt{void} \to \tau$. This is exactly what we require to prove (2.3), which completes this case.

Case: $s_1 \neq Cx$; From (2.2) we can deduce

$$\delta; context(H, (BS' \circ MS) \circ VS) \vdash \overline{s} : \texttt{void} \tag{2.4}$$
$$\delta; context(H, (BS' \circ MS) \circ VS) \vdash s_1 : \texttt{void} \tag{2.5}$$

Specialising the inductive hypothesis and using (2.4) gives us $\delta, H, (BS' \circ MS) \circ VS \vdash \overline{s} \circ \{\} \circ FS : \texttt{void} \to \tau$, which combined with (2.5) gives (2.3), and completes this case.

□

We can now prove the second important proposition of our reduction system, which states that if a configuration can make a transition, then the resulting configuration is of the appropriate type. (This is sometimes referred to as the "subject reduction" theorem.)

Proposition 2.5.6 (Type preservation)**.** *If $\delta \vdash config : \tau$ and $config \to config'$ then $\exists \tau'.\ \ \delta \vdash config' : \tau'$ where $\tau' \prec \tau$.*

Proof. By case analysis on the $\to$ relation. The decomposition rules, in Figure 2.8, and E-SKIP follow by trivially restructuring the proof trees. The remaining rules we prove as follows. Assume

a. $\delta \vdash H$ ok
b. $\delta, H \vdash VS$ ok
c. $\delta, H, VS \vdash CF \circ FS : \texttt{void} \to \tau$
d. $(H, VS, CF, FS) \to (H', VS', CF', FS')$

We must prove the following

1. $\delta \vdash H'$ ok
2. $\delta, H' \vdash VS'$ ok
3. $\delta, H', VS' \vdash CF' \circ FS' : \texttt{void} \to \tau' \wedge \tau' \prec \tau$

Case: E-SUB If $FS = CF \circ FS'$ then holds trivially.
Otherwise $FS = OF \circ FS'$.

1. Holds by assumptions as heap is unchanged.
2. Holds by assumptions as stack is unchanged.
3. Holds due to

$$\delta; context(H, VS), \bullet : \tau' \vdash OF : \tau'' \ \wedge\ \delta; context(H, VS) \vdash v : \tau' \Rightarrow \delta; context(H, VS) \vdash OF[v/\bullet] : \tau''$$

which is shown by induction on OF.

Case: E-RETURN

1. Holds as heap is unchanged.
2. Holds as new stack is a sub stack of the old one.
3. Holds as returned value only depends on the heap.

Case: E-VARACCESS Let $F = x$

1. Holds as heap is unchanged
2. Holds as stack is unchanged
3. As stack is well typed, if x evaluates to v, then v's type is a subtype of x's static type. Using covariant lemma rest of framestack is well-typed.

Case: E-VARWRITE Let $F = x = v;$

1. Holds as heap is unchanged.
2. For F to be well-typed v must be a sub-type of x's static type. So updated stack is well-formed.
3. Holds as new frame stack is contained in the old one, with a skip prepended.

Case: E-VARINTRO

1. Holds as heap is unchanged.
2. New entry in stack is well-typed, so new stack is well-typed.
3. New framestack is typed by rearranging the old ones proof.

Case: E-IF1

1. Holds as heap is unchanged.
2. Holds as stack is unchanged.
3. New framestack is typed, because all the elements were typed by the previous one.

Case: E-IF2 Identical to previous case.

Case: E-BLOCKINTRO

1. Holds as heap is unchanged.
2. Adding an empty block to a well-typed stack results in a well-typed stack.
3. Holds as consequence of lemma 2.5.4.

Case: E-CAST

1. Holds as heap is unchanged.
2. Holds as stack is unchanged.
3. Holds directly by covariant lemma.

Case: E-FIELDWRITE Let $F = o.f = v;$

1. As F is well typed we know that v is a subtype of the field's type the heap update leaves it well typed.
2. Holds as the stack in unchanged.
3. New framestack is typed, because all the elements were typed by the previous one, and the typing information in the heap is unchanged.

Case: E-FIELDACCESS Let $F = o.f$

1. Holds as heap is unchanged.
2. Holds as stack is unchanged.
3. As heap is well-typed, we know $H(H(o, f)) \prec \delta_{\mathrm{F}}(H(o), f)$. Hence the new stack is typed by the covariant lemma.

Case: E-METHOD Let $F = o.m(\overline{v})$

1. Holds as the heap is unchanged.
2. As frame stack was well-typed if $\overline{v}$ has type $\overline{C}$ then $\overline{C} \prec \overline{C'}$ and $\delta_{\mathrm{F}}(H(o), C) = \overline{C'} \to C$. Therefore new scope is well-typed.

3. As the new scope gives the same scope as the method was type-checked against, we can use lemma 2.5.5 and the covariant lemma to show the new framestack is well-typed. Covariant lemma is required as the `return` e; could be a more specific type than the method signature gives.

Case: E-METHODVOID Same as previous case, with a trivial alteration for the `return` .
Case: E-NEW

1. The heap is extended with an object. This extension has all the required fields, and their values are all null. So the new heap is well-typed.
2. Follows by similar argument to previous case, with the heap extension lemma 2.5.3.
3. Follows by similar argument to previous case, but covariant lemma is not required, and heap extension preserves typing is require 2.5.3.

Case: E-SUPER 1. Holds as heap is unchanged.
2. Follows by similar argument to previous case.
3. Follows by similar argument to previous case.

□

We can now combine the propositions 2.5.1 and 2.5.6 to prove the type safety of the MJ type system.

Theorem 2.5.7 (Type safety)**.** *If* $config : \tau$ *and* $config \rightarrow^* config'$ *then* $config' : \tau'$ *where* $\tau' \prec \tau$ *and either* $config'$ *is terminal or not stuck.*

2.6 Related Work

There have been many works on formalizing subsets of Java. Our work is closely related to, and motivated by, Featherweight Java [47]. We have upheld the same philosophy, by keeping MJ a valid subset of Java. However FJ lacks many key features we wish to reason about. It has no concept of state or object identity, which is essential for developing usable specification logics. Our work has extended FJ to contain what we feel are the important imperative features of Java.

Another related calculus is Classic Java [38], which embodies many of the language features of MJ. However, Classic Java is not a valid subset of Java, as it uses type annotations and also uses let-binding to model both sequencing and locally scoped variables. Hence several features of Java, such as its block structured mutable state and unusual syntactic distinction of promotable expressions are not modelled directly. However, Felleisen *et al.*'s motivation is different to ours: they are interested in extending Java with mixins, rather than reasoning about features of Java.

Eisenbach, Drossopoulou, *et al.* have developed type soundness proofs for various subsets of Java [33, 34]. They consider a larger subset of Java than MJ as they model exceptions and arrays, however they do not model block structured scoping. Our aim was to provide an imperative core subset of Java amenable to formal proof, rather than to prove soundness of a large fragment. Syme [82] has formalised Eisenbach, Drossopoulou, *et al.*'s semantics in Isabelle.

There have been other fragments of Java formalised in theorem provers. Nipkow and Oheimb [62] formalize a subset of Java in the Isabelle theorem prover. More recently, Nipkow and Klein have formalized a larger subset of Java, named Jinja [51]. They also model the bytecode and compiler in the theorem prover.

3

A logic for Java

In this chapter we present a programming logic for reasoning about Java. Although the logic described in this chapter is unable to reason about forms of abstraction commonly found in object-oriented languages (namely encapsulation and inheritance) it forms a basis for the logics in the rest of the thesis. Additionally it serves as an introduction to separation logic.

We choose to build our logic on top of the separation logic by O'Hearn, Reynolds and others [76, 48, 65, 75], as it addresses the problems of aliasing (see §1.1.2 for details). Separation logic extends Hoare logic with spatial connectives, which allow assertions to define separation between parts of the heap. This separation provides the key feature of separation logic—*local reasoning*—specifications need only mention the state that they access. This local reasoning approach has proved itself successful when considering some real-world algorithms, including the Schorr-Waite graph marking algorithm [88] and a copying garbage collector [11].

Until now, all the work on separation logic has focused on simple, imperative, low-level languages. In this chapter we present an extension to separation logic to allow reasoning about Java-like languages, that is languages with features that are prevalent in the real-world including encapsulation and inheritance. It is known that separation logic solves the problems caused by aliasing. However a naïve adaptation of this approach falls short of reasoning about encapsulation and inheritance. Solutions to these issues will be presented in Chapters 4 and 6.

The rest of the chapter is structured as follows. First we present a simplification of MJ that is well suited to separation logic. In §3.2 we describe the storage model we will reason about, and then, in §3.3, we give the syntax and semantics of separation logic assertions. In §3.4 we present the Hoare logic rules to reason about Java, and then in §3.5, we present an additional rule to reason about dynamically dispatched method calls. We then give an example of copying a tree in §3.6, and present a full semantic account of the logic and its soundness in §3.7. We conclude, in §3.8, by discussing encapsulation and inheritance.

3.1 Inner MJ

In order to allow clean and elegant separation logic rules, we make a few simplifications to MJ, and call this subset *Inner MJ*. The first and most important simplification is that statements are not allowed multiple indirections: a statement can only perform a single read or write of a field. Additionally, for compactness, we remove `void` methods. The simplifications to the syntax are presented in Figure 3.1; the original syntax is given on page 20.

Method definition
$md ::= C\ m(C_1\ x_1,\ \ldots,\ C_n\ x_n)\{\overline{s}\ \texttt{return}\ x;\}$
Expressions
$e ::= x \mid \texttt{null}$
Statements
$s ::= x = y.f; \mid x = (C)y; \mid x = \texttt{new}\ C();$
$\mid\ x.f = e; \mid x = y.m(\overline{e}); \mid C\ x;$
$\mid\ \{\overline{s}\} \mid ; \mid \texttt{if}\,(e == e)\{\overline{s}\}\,\texttt{else}\,\{\overline{s}\}$

Figure 3.1: Syntax of Inner MJ

Additionally, we make a semantic restriction on method and constructor bodies that they do not modify their parameters. This reduces the complexity of reasoning about method and constructor calls. This may at first sound restrictive but simply assigning each parameter to a new local variable allows modification. This restriction removes the need to consider the call by value arguments to a method.

Although we have simplified the syntax, we have not removed any computational power. A simple transformation from full MJ to Inner MJ is given in Figure 3.2. We write $(\!|e|\!)_z$ for the transformation on expressions to statements. This replaces e by a series of small statements which evaluate the expression. The final statement stores the result in z. We then define the transformation on statements $(\!|s|\!)$. This evaluates each expression, stores its result in a variable, and then evaluates the statement for these variables. The transformation on a program simply transforms all the method and constructor bodies. In Lemma 3.7.17 on page 61 we prove this translation preserves program semantics.

3.2 Storage model

Recent work on separation logic has allowed reasoning about pointer datastructures with address arithmetic [65, 75]. This includes complicated doubly linked lists where only the XOR of the forwards and backwards pointers is stored in the heap. This reasoning is powerful and interesting, but it does not represent the programmer's view of memory in Java: Java does *not* provide address arithmetic. Obviously, this model can be used to represent Java's memory by compiling field locations away, but representing the compilation in the logic seems an unnecessary burden on the proof. It would be analogous to reasoning about a high-level language by considering the assembly language to which it is compiled. The original work on separation logic [76, 48] considered the heap to be a partial finite function from locations to unlabelled records:

$$H : \mathsf{Locations} \rightharpoonup_{\mathit{fin}} \mathsf{Values}^{+}$$

where Locations correspond to oids from the previous chapter, and Values are oids and null.

Note: In particular many models consider a simplification to just pairs, $\mathsf{Values} \times \mathsf{Values}$, rather than unlabelled records, Values^{+}

In order to represent objects in this model, we can replace the tuple by a partial finite function from field names to values:

$$H : \mathsf{Locations} \rightharpoonup_{\mathit{fin}} (\mathsf{Fields} \rightharpoonup_{\mathit{fin}} \mathsf{Values})$$

This model is similar to those used in object-oriented semantics [9, 33, 38]. It is useful to be able to split an object into its different aspects, e.g. a `ColourPoint` class has a location and a

Expressions

$$
\begin{array}{rcl}
(\!|x|\!)_z & = & z = x; \\
(\!|\texttt{null}|\!)_z & = & z = \texttt{null}; \\
(\!|e.f|\!)_z & = & \{D\ y; (\!|e|\!)_y z = y.f; \} \\
(\!|(C)e|\!)_z & = & \{D\ y; (\!|e|\!)_y z = (C)y; \} \\
(\!|e.m(e_1,\ldots,e_n)|\!)_z & = & \{D\ y; D_1\ y_1; \ldots; D_n\ y_n; (\!|e|\!)_y (\!|e_1|\!)_{y_1} \ldots (\!|e_n|\!)_{y_n} \\
 & & \qquad z = y.m(y_1,\ldots,y_n); \} \\
(\!|\texttt{new}\ \ C(e_1,\ldots,e_n)|\!)_z & = & \{D_1\ y_1; \ldots; D_n\ y_n; (\!|e_1|\!)_{y_1} \ldots (\!|e_n|\!)_{y_n} \\
 & & \qquad z = \texttt{new}\ \ C(y_1,\ldots,y_n); \}
\end{array}
$$

Statements

$$
\begin{array}{rcl}
(\!|;|\!) & = & ; \\
(\!|e;|\!) & = & \{D\ y; (\!|e|\!)_y\} \\
(\!|\texttt{if}\,(e_1 == e_2)\{\overline{s}_1\}\,\texttt{else}\,\{\overline{s}_2\}|\!) & = & \{D_1\ y_1; D_2\ y_2; (\!|e_1|\!)_{y_1} (\!|e_2|\!)_{y_2} \\
 & & \qquad \texttt{if}\,(y_1 == y_2)\{(\!|\overline{s}_1|\!)\}\,\texttt{else}\,\{(\!|\overline{s}_2|\!)\}\} \\
(\!|e_1.f = e_2|\!) & = & \{D_1\ y_1; D_2\ y_2; (\!|e_1|\!)_{y_1} (\!|e_2|\!)_{y_2} y_1.f = y_2; \} \\
(\!|x = e;|\!) & = & (\!|e|\!)_x \\
(\!|\texttt{return}\,e;|\!) & = & D\ y; (\!|e|\!)_y \texttt{return}\,y; \\
(\!|\{\overline{s}\}|\!) & = & \{(\!|\overline{s}|\!)\} \\
(\!|C\ x;|\!) & = & C\ x; \\
(\!|s\ \overline{s}|\!) & = & (\!|s|\!)(\!|\overline{s}|\!) \\
(\!|\epsilon|\!) & = & \epsilon
\end{array}
$$

provided the static types of $e, e_1, \ldots, e_n$ are $D, D_1, \ldots, D_n$ respectively, and $y, y_1, \ldots, y_n$ are always fresh variables.

Figure 3.2: Transformation of expressions and statements

colour. We want to consider methods that modify the location property without having to reason about the colour property, hence it is important to be able to split an object. Later we will see that this simplifies the logic, as it allows the rules to be given with respect to just the fields used by a command. The frame rule can then be used to extend the specifications to the rest of the object, as well as to other objects. However, in this model it is difficult to define the composition of heaps such that an oid can be in both heaps as long as the two heaps have different fields for that oid.

By uncurrying the partial finite functions, we get a natural definition that allows splitting at the field level.

$$H : \mathsf{Locations} \times \mathsf{Fields} \rightharpoonup_{fin} \mathsf{Values}$$

Finally, we store each object's type. This could be done using a special field. In order to allow sharing of the immutable type information we distinguish it as in the definition below.[1]

[1] Java does not allow the explicit disposal of an object and an object can never change type, i.e. the type of an object is immutable, so it does not matter who knows it. If one allows disposal of objects then sharing this information would be somewhat dangerous, and methods described in Chapter 5 should be employed for such sharing.

Definition 3.2.1 (Heap)**.** A heap, H, is a pair of finite partial functions: the first stores the fields, H_v, and the second stores the types of objects, H_t.

$$H : \mathcal{H} \stackrel{\text{def}}{=} (\mathsf{Locations} \times \mathsf{Fields} \rightharpoonup_{fin} \mathsf{Values}) \times (\mathsf{Locations} \rightharpoonup_{fin} \mathsf{Classes})$$

We write $H \perp H'$ to mean two heaps are disjoint and compatible:

$$(H'_v, H'_t) \perp (H''_v, H''_t) \quad \stackrel{\text{def}}{=} \quad dom(H'_v) \perp dom(H''_v) \quad \wedge \quad H'_t = H''_t$$

We write $H * H$ for the composition of two heaps:

$$(H'_v, H'_t) * (H''_v, H''_t) \stackrel{\text{def}}{=} \begin{cases} (H'_v \circ H''_v, H'_t) & (H'_v, H'_t) \cap (H''_v, H''_t) = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $\circ$ is union of disjoint partial functions. Finally we define an order on heaps $\subseteq$

$$H_1 \subseteq H_2 \quad \stackrel{\text{def}}{=} \quad \exists H_3 . H_1 * H_3 = H_2$$

Note: This definition is the same as in the previous chapter. Classes is the set of all class names.

Again we use the same shorthands for accessing the first and second components of the heap: $(H_v, H_t)(o) = H_t(o)$ and $(H_v, H_t)(o, f) = H_v(o, f)$. We use the obvious shorthands for dom and function update that can always be disambiguated by context, i.e. $(o, f) \in dom(H)$ checks that (o, f) is in the domain of the first component of H.

A stack is defined as in the previous chapter. We also use ghost[2] variables that do not interfere with program execution. Ghost variables are used to express protocols of method calls, so they require a global scope. We cannot simply use the program stack as this cannot define a variable across method calls: it only contains local variables. So we define a ghost stack that maps ghost variables, e.g. X, Y, etc. to values.

A logical state is a triple of the form (VS, H, I) where VS is a program variable stack, H is a heap and I is a ghost variable stack.

3.3 Assertion language

The key difference between the assertion languages of separation logic and Hoare logic is the introduction of the spatial, or multiplicative, conjunction *. The spatial conjunction $P * Q$ means the heap can be split into two disjoint parts in which P and Q hold respectively.

There are two flavours of separation logic: classical and intuitionistic [48]. In classical separation logic the spatial conjunction does not admit weakening or contraction, while in intuitionistic separation logic it admits weakening, but not contraction. Also unlike classical separation logic, intuitionistic separation logic does not have an empty heap predicate: $true$ is the unit of both $*$ and $\wedge$.

In this thesis we will use the intuitionistic version. This is because Java is a garbage collected language, hence it does not have an explicit dispose primitive. State is deallocated when it is inaccessible. As intuitionistic separation logic admits weakening, the following holds $P * Q \Rightarrow Q$. This allows state to be ignored, hence we do not need to explicitly dispose state. If the language was not garbage collected, then intuitionistic separation logic would allow memory leaks.

[2] These variables are sometimes called auxiliary or logical variables.

$$
\begin{array}{ll}
VS,H,I \models \mathit{false} & \stackrel{\text{def}}{=} \text{never} \\
VS,H,I \models \mathit{true} & \stackrel{\text{def}}{=} \text{always} \\
VS,H,I \models P \wedge Q & \stackrel{\text{def}}{=} VS,H,I \models P \text{ and } VS,H,I \models Q \\
VS,H,I \models P \vee Q & \stackrel{\text{def}}{=} VS,H,I \models P \text{ or } VS,H,I \models Q \\
VS,H,I \models P \Rightarrow Q & \stackrel{\text{def}}{=} \forall H' \supseteq H. \text{ if } VS,H',I \models P \text{ then } VS,H',I \models Q \\
VS,H,I \models \forall x.P & \stackrel{\text{def}}{=} \forall v \in \textsf{Values}.\ VS[x \mapsto v],H,I \models P \\
VS,H,I \models \exists x.P & \stackrel{\text{def}}{=} \exists v \in \textsf{Values}.\ VS[x \mapsto v],H,I \models P \\
VS,H,I \models e.f \mapsto e' & \stackrel{\text{def}}{=} H([\![e]\!]_{VS,I}, f) = [\![e']\!]_{VS,I} \\
VS,H,I \models e : C & \stackrel{\text{def}}{=} H([\![e]\!]_{VS,I}) = C \\
VS,H,I \models e = e' & \stackrel{\text{def}}{=} [\![e]\!]_{VS,I} = [\![e']\!]_{VS,I} \\
VS,H,I \models P * Q & \stackrel{\text{def}}{=} \exists H_1, H_2.\ H_1 * H_2 = H, \quad VS,H_1,I \models P \text{ and } VS,H_2,I \models Q \\
VS,H,I \models P \mathrel{-\!\!*} Q & \stackrel{\text{def}}{=} \forall H_1. \text{ if } H_1 \perp H \text{ and } VS,H_1,I \models P \text{ then } VS,H_1 * H,I \models Q
\end{array}
$$

where $[\![X]\!]_{VS,I} = I(X)$, $[\![\texttt{null}]\!]_{VS,I} = \texttt{null}$ and $[\![x]\!]_{VS,I} = VS(x)$.

Figure 3.3: Semantics of the assertion language

However, Reynolds noted that intuitionistic separation logic does not correctly model garbage collected languages [76] as assertions can be made about inaccessible state. Consider the following code fragment:

```
x = new C(); x = null;
```

The constructed object is not accessible. In the programming logic, given the usual semantics [48], it is possible to prove it exists in the heap after the assignment of `null`. If we model the garbage collector operationally, then we can prove the object may have been garbage collected. Therefore our assertion language would be unsound. Calcagno and O'Hearn [21] present a solution to this problem where they alter the semantics of the existential quantifier to correctly model garbage collected languages. Fortunately, we can avoid these complications as we do not model the garbage collector operationally, and hence it is sound to remain in "vanilla" intuitionistic separation logic.

Formulae in our assertion language are given by the following grammar

$$
\begin{array}{rl}
P,Q ::= & \mathit{true} \mid \mathit{false} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x.P \mid \exists x.P \\
& \mid P * Q \mid P \mathrel{-\!\!*} Q \mid e.f \mapsto e' \mid v : C \mid e = e
\end{array}
$$

Expressions, e, are extended from Figure 3.1 to allow ghost variables as well.

For intuitionistic logic we require all assertions to satisfy the monotonicity, or persistency, constraint.

Definition 3.3.1 (Monotonicity constraint)**.** P satisfies the monotonicity constraint if whenever $VS,H,I \models P$ and $H \subseteq H'$, then $VS,H',I \models P$

The monotonic subsets of the powerset of states form a Heyting algebra. We provide the formal semantics of formulae in Figure 3.3. The intuitionistic connectives $(\vee, \wedge, \Rightarrow)$ and quantifiers $(\forall, \exists)$ are interpreted in the usual way [48]. In addition to the intuitionistic connectives we have the new spatial connectives $*$ and $\mathrel{-\!\!*}$, along with the predicates $e = e'$, $e : C$ and $e.f \mapsto e'$. Taking these in reverse order: the predicate $e.f \mapsto e'$ consists of all the tuples (VS,H,I) where the heap, *H*, consists of at least the single mapping from the location given by the meaning of e and the field f to the value given by the meaning of e'. $e : C$ is true if the evaluation of e gives

$$\begin{aligned}
\text{mods}(x = \ldots) &\stackrel{\text{def}}{=} \{x\} \\
\text{mods}(x.f = e;) &\stackrel{\text{def}}{=} \text{mods}(;) \stackrel{\text{def}}{=} \{\} \\
\text{mods}(C\ x; \overline{s}) &\stackrel{\text{def}}{=} \text{mods}(\overline{s}) \setminus x \\
\text{mods}(s\overline{s}) &\stackrel{\text{def}}{=} \text{mods}(s) \cup \text{mods}(\overline{s}) \\
\text{mods}(\{\overline{s}\}) &\stackrel{\text{def}}{=} \text{mods}(\overline{s}) \\
\text{mods}(\texttt{if}\,(e_1 == e_2)\{\overline{s}_1\}\,\texttt{else}\,\{\overline{s}_2\}) &\stackrel{\text{def}}{=} \text{mods}(\overline{s}_1) \cup \text{mods}(\overline{s}_2)
\end{aligned}$$

Figure 3.4: Modifies property for Inner MJ

a value of dynamic type[3] C. $e = e'$ is true in any state where the expressions evaluate to the same value. The spatial conjunction $P * Q$ means the heap can be split into two disjoint parts in which P and Q hold respectively. Heaps of more than one element are specified by using the $*$ connective to join smaller heaps. Finally, the adjoint to $*$, written $P \mathrel{-\!\!*} Q$, means that if the heap can be extended with any disjoint heap satisfying P then the resultant heap will satisfy Q.

The essence of "local reasoning" is that, to understand how a piece of code works, it should only be necessary to reason about the memory the code actually accesses (its so-called "footprint"). Ordinarily, aliasing precludes such a principle but the separation enforced by the $*$ connective allows this intuition to be captured formally by the following rule.

L-FRAME

$$\frac{\vdash \{P\}\overline{s}\{Q\}}{\vdash \{P * R\}\overline{s}\{Q * R\}}$$

provided $\overline{s}$ does not modify the free variables of R: $\text{mods}(\overline{s}) \cap FV(R) = \emptyset$

The side-condition is required because $*$ only describes the separation of heap locations and not variables, without this restriction $\overline{s}$ could modify stack variables in R altering R's validity; see [13] for more details. We give the definition of mods(_) in Figure 3.4.

By using this rule, a local specification concerning only the variables and parts of the heap that are used by $\overline{s}$ can be arbitrarily extended as long as the extension's free variables are not modified by $\overline{s}$. Thus, from a local specification we can infer a global specification that is appropriate to the larger footprint of an enclosing program. Consider a function that operates on a list:

$$\{list(i)\}m(i)\{list(i)\}$$

This specification says that if provided a list, i, then if `m` returns it will leave i still satisfying $list$. Implicitly, anything not mentioned is unchanged by the call. Hence we can derive a specification

$$\{list(i) * P\}m(i)\{list(i) * P\}$$

where P represents the other state we might know about. We have gone from a local specification describing what we use, to a global specification that additionally describes the context: this means we do not need to provide a different specification for each call site.

3.4 Java rules

Now we provide the logic for reasoning about Java. A judgement in our assertion language is written as follows:

$$\Gamma \vdash \{P\}\overline{s}\{Q\}$$

[3] We use dynamic type to mean the class the object was created with, and static type for the type of an expression denoting that object. For example `C x = new D();` The static type of `x` is `C` but the dynamic type of `x` is `D`.

This is read: the statement, $\overline{s}$, satisfies the specification $\{P\}\_\{Q\}$, given the method and constructor hypotheses, Γ. These hypotheses are given by the following grammar:

$$\begin{array}{rcl} \Gamma & := & \epsilon \mid \{P\}\eta(\overline{x})\{Q\}, \Gamma \\ \eta & := & C.m \mid C \end{array}$$

However, when it simplifies the presentation, we will treat Γ as a partial function from method and defining class name, $C.m$, and constructor names, C, to specifications. For the hypotheses, Γ, to be well-formed each method, $C.m$, and constructor, C, should appear at most once; and each specification's free program variables should be contained in the method's arguments and ret. We will only consider well-formed Γ.

First we present the rules for field manipulation. We give small axioms in the style of O'Hearn, Reynolds, and Yang [65]. Field write requires the heap to contain at least the single field being written to, and the post-condition specifies it has the updated value. Implicitly the rule specifies no other fields are modified, and hence it can be extended by the frame rule to talk about additional state. The field access rule simply requires the state to be known and sets the variable equal to the contents. The ghost variables X and Y are used to allow x and y to be syntactically the same variable without needing a case split or substitution in the rule definition.

L-FWRITE	$\Gamma \vdash \{x.f \mapsto \_\}x.f = y; \{x.f \mapsto y\}$
L-FREAD	$\Gamma \vdash \{X = x \wedge X.f \mapsto Y\}y = x.f; \{X.f \mapsto Y \wedge y = Y\}$

Note: These rules highlight our choice to have fields assertions as the primitive assertion about the heap. The rules are given in terms of just the fields they use, and the frame rule dictates what happens to the other values. If the primitive assertions described entire objects, we would need to give the rules in terms of the whole object and say how the other fields were affected by this assignment.

Next we present the rules associated with types. The first ensures that a cast will complete successfully: the object identifier being cast must be a subtype of the target type or null. The L-STYPE rule allows static typing information to be exploited in the proof: if we can prove that it meets the specification for all subtypes and `null`, then it meets the specification without any typing information.

L-DOWNCAST
$$\Gamma \vdash \{P[x/y] \wedge (x : C \vee x = \texttt{null})\}y = (D)x\{P\}$$
provided $C \prec D$.

L-STYPE
$$\frac{\Gamma \vdash \{P \wedge x = null\}\overline{s}\{Q\} \quad \forall C \prec D.\, \Gamma \vdash \{P \wedge x : C\}\overline{s}\{Q\}}{\Gamma \vdash \{P\}\overline{s}\{Q\}}$$
provided x has static type D.

Note: The L-STYPE proof rule is not local: it requires knowledge of all the classes that exist. Hence it will not lead to open proofs, i.e. proofs open to the extension of the class hierarchy. Later we will use this rule with a restriction on the class hierarchy to derive the rule for dynamic dispatch.

Next we give the rules for reasoning about methods. These are essentially the same as the standard function rules [75]. The method call rule states that if the pre-condition of $C.m$ holds and the receiver is of type C, then the post-condition will hold after the call. The L-RETURN rule allows the method's return value to be matched with the call's return variable using the special

variable ret. This command is logically equivalent to $ret = v;$: we give this rule to match the Java syntax.

L-CALL	$\{P\}C.m(\overline{x})\{Q\}, \Gamma \vdash \{P \wedge \texttt{this} : C\}ret = \texttt{this}.m(\overline{x}); \{Q\}$
L-RETURN	$\Gamma \vdash \{P[v/ret]\}\texttt{return}\, v; \{P\}$

The arguments, receiver and return variable of a method, and constructor, can all be specialized to arbitrary variables using the variable substitution rule, L-VARSUBST. The rule is similar to that given by Reynolds [75]. The first side-condition is the same as given by Reynolds, and prevents the substitution creating assignment to an expression. The second is because we make a distinction between ghost variables and program variables, and hence ghost variables cannot appear in the program text. The third restriction is because `return` implicitly uses a variable ret, but ret is not syntactically free in $\texttt{return}\, x;$.

L-VARSUBST
$$\frac{\Gamma \vdash \{P\}\overline{s}\{Q\}}{\Gamma \vdash \{\theta(P)\}\theta(\overline{s})\{\theta(Q)\}}$$
provided $\theta = [e_1, \ldots, e_n/x_1, \ldots, x_n]$;
- if x_i is modified by $\overline{s}$, then e_i is just a variable not free in any other e_j;
- if x_i is in the free variables of $\overline{s}$, then e_i is restricted to program expressions, expressions not containing ghost variables; and
- if $\overline{s}$ contains `return` then $\theta(ret) = ret$.

Note: We allow θ to alter `this`, as it simplifies the L-CALL rule.

Next, we give the rules for reasoning about constructors. Before we give the rules we define the shorthand $new(C, x)$ as meaning $x.f_1 \mapsto \texttt{null} * \ldots * \; x.f_n \mapsto \texttt{null}$ where $dom(\delta_{\mathrm{F}}(C)) = f_1, \ldots, f_n$. The L-NEW rule is similar to the L-CALL rule with the additional post-condition giving the object's type. The first statement in a constructor definition is always a `super` call. We validate a `super` call in the same way as `new`, but the post-condition about the object's type is not added, and the pre-condition requires the fields to exist (and be initialized to `null`).

L-NEW	$\Gamma, \{P\}C(\overline{x})\{Q\} \vdash \{P\}\texttt{this} = \texttt{new}\; C(\overline{x}); \{Q \wedge \texttt{this} : C\}$
L-SUPER	$\Gamma, \{P\}D(\overline{x})\{Q\} \vdash \{P * new(D, \texttt{this})\}\texttt{super}(\overline{x}); \{Q\}$ provided the `super` call is in class C constructor and $C \prec_1 D$.

Now we give the rules for introducing assumptions about definitions of methods and constructors, given in Figure 3.5. We define a new judgement $\Gamma_1 \Vdash \Gamma_2$ to allow the introduction of mutually recursive methods. $\Gamma_1 \Vdash \Gamma_2$ means the bodies of the methods in Γ_2 can be verified assuming all the method calls satisify the specifications in Γ_1. The method introduction rule, L-DMETHOD, checks that the body, $\overline{s}$, meets the specification assuming it is invoked on the correct class. The constructor introduction rule, L-DCONSTR, verifies the body with both the pre-condition and the new fields for the object. It is not given the pre-condition $\texttt{this} : C$ as the constructor is also called by the `super` call. The remaining three rules are used to introduce and manipulate these definitions. These rules are similar in style to those used by von Oheimb [83] to reason about mutually recursive procedures.

Finally, the standard rules are presented in Figure 3.6. As we allow allocation of new variables at any point in a sequence, we require two sequencing rules. The first is for the case where the first command is not a variable allocation. The second is where a new variable is being created. This requires that the new variable does not get passed out of this sequence.

L-DMETHOD
$$\frac{\Gamma \vdash \{P \wedge \texttt{this} : C\}\overline{s}\{Q\}}{\Gamma \Vdash \{P\}C.m(\overline{x})\{Q\}}$$
provided $mbody(C, m) = (\overline{x}, \overline{s})$

L-DCONSTR
$$\frac{\Gamma \vdash \{new(C, \texttt{this}) * P\}\texttt{super}(\overline{e}); \overline{s}\{Q\}}{\Gamma \Vdash \{P\}C(\overline{x})\{Q\}}$$
provided $cnbody(C) = (\overline{x}, \texttt{super}(\overline{e}); \overline{s})$.

L-DSPLIT
$$\frac{\Gamma \Vdash \Gamma_1 \quad \Gamma \Vdash \Gamma_2}{\Gamma \Vdash \Gamma_1, \Gamma_2}$$

L-DWEAKEN
$$\frac{\Gamma \Vdash \Gamma_1}{\Gamma, \Gamma_2 \Vdash \Gamma_1}$$

L-DINTRO
$$\frac{\Gamma, \Gamma' \Vdash \Gamma' \quad \Gamma, \Gamma' \vdash \{P\}\overline{s}\{Q\}}{\Gamma \vdash \{P\}\overline{s}\{Q\}}$$

Figure 3.5: Definition rules

L-IF
$$\frac{\Gamma \vdash \{P \wedge x = y\}\overline{s}_1\{Q\} \quad \Gamma \vdash \{P \wedge \neg(x = y)\}\overline{s}_2\{Q\}}{\Gamma \vdash \{P\}\texttt{if}\,(x == y)\{\overline{s}_1\}\,\texttt{else}\,\{\overline{s}_2\}\{Q\}}$$

L-SKIP
$$\Gamma \vdash \{P\}; \{P\}$$

L-BLOCK
$$\frac{\Gamma \vdash \{P\}\{\overline{s}\}\{Q\}}{\Gamma \vdash \{P\}\overline{s}\{Q\}}$$

L-SEQ1
$$\frac{\Gamma \vdash \{P\}s_1\{R\} \quad \Gamma \vdash \{R\}s_2 ... s_n\{Q\} \quad s_1 \neq Cx;}{\Gamma \vdash \{P\}s_1 s_2 ... s_n\{Q\}}$$

L-SEQ2
$$\frac{\Gamma \vdash \{P\}s_1 ... s_n\{Q\} \quad x \notin FV(P) \cup FV(Q)}{\Gamma \vdash \{P\}C\ x; s_1 ... s_n\{Q\}}$$

L-CONSEQUENCE
$$\frac{P \Rightarrow P' \quad \Gamma \vdash \{P'\}C\{Q'\} \quad Q' \Rightarrow Q}{\Gamma \vdash \{P\}C\{Q\}}$$

L-VARELIM
$$\frac{\Gamma \vdash \{P\}\overline{s}\{Q\}}{\Gamma \vdash \{\exists x.P\}\overline{s}\{\exists x.Q\}}$$
provided x is not free in $\overline{s}$.

Figure 3.6: Standard Rules

The whole program is verified by

$$\{true\}\texttt{Object()}\{true\} \vdash \{P\}\overline{s}\{Q\}$$

Note: There is an implicit context of the program π.

3.5 Dynamic dispatch

The method call rule, L-CALL, defined earlier does not account for dynamic dispatch. As explained in §1.1.1, the usual approach to dealing with dynamic dispatch is to use behavioural subtyping. We present a generalised notion of behavioural subtyping: *specification compatibility*.

Definition 3.5.1 (Specification compatibility)**.** We define specification compatiblity, $\vdash \{P_D\}\_\{Q_D\} \Rightarrow \{P_C\}\_\{Q_C\}$, iff forall $\overline{s}$ if $\mathit{modifies}(\overline{s}) \subseteq \{ret\}$ then $\Gamma \vdash \{P_C\}\overline{s}\{Q_C\}$ is derivable from $\Gamma \vdash \{P_D\}\overline{s}\{Q_D\}$, that is a proof of $\Gamma \vdash \{P_C\}\overline{s}\{Q_C\}$ exists whose only assumption is $\Gamma \vdash \{P_D\}\overline{s}\{Q_D\}$.

The quantification over the statement sequence $\overline{s}$ in the definition restricts the derivation to the structural rules: L-CONSEQUENCE, L-FRAME, L-VARELIM and L-VARSUBST. The frame rule, L-FRAME, can only introduce formula that do not mention ret. If the derivation only uses the rule L-CONSEQUENCE, specification compatibility degenerates to standard behavioural subtyping.

From now on in this thesis we will restrict method and constructor environments to *well-behaved environments*, that is where subtypes must have compatible specifications with their supertypes' specification.

Definition 3.5.2 (Well-behaved environments)**.** For each $\{P_C\}C.m(\overline{x})\{Q_C\}$ in Γ, if $D \prec C$, then $\{P_D\}D.m(\overline{y})\{Q_D\}$ is in Γ and $\vdash \{P_D[\overline{x}/\overline{y}]\}\_\{Q_D[\overline{x}/\overline{y}]\} \Rightarrow \{P_C\}\_\{Q_C\}$

The substitution, $[\overline{x}/\overline{y}]$, in the definition is required to allow methods to use different argument names.

Specification compatibility generalises behaviour subtyping to allow variable manipulation. Consider the following specifications

$$\{\texttt{this}.f \mapsto \_\}\_\{\texttt{this}.f \mapsto \_\} \tag{3.1}$$
$$\{\texttt{this}.f \mapsto X\}\_\{\texttt{this}.f \mapsto X\} \tag{3.2}$$

If code expects the behaviour of (3.1) then clearly it would be fine with (3.2): it is a refinement. However, the standard behavioural subtyping implications would not allow this as $\texttt{this}.f \mapsto \_ \Rightarrow \texttt{this}.f \mapsto X$ does not hold. Hence we must add rules to allow variable manipulation in behavioural subtyping.

Not only does specification compatibility generalize behavioural subtyping in its manipulation of variables, but it also allows specification intersections. Consider the following pair of specifications

Method	Pre-condition	Post-condition
`C.m`	$P_{\texttt{C}}$	$Q_{\texttt{C}}$
`D.m`	$(P_{\texttt{C}} \wedge X = 1) \vee (P_{\texttt{D}} \wedge X = 2)$	$(Q_{\texttt{C}} \wedge X = 1) \vee (Q_{\texttt{D}} \wedge X = 2)$

Note: We use the ghost variable X to allow us to encode the intersection of the specification $\{P_{\texttt{C}}\}\_\{Q_{\texttt{C}}\}$ with $\{P_{\texttt{D}}\}\_\{Q_{\texttt{D}}\}$. We assume $P_{\texttt{C}}$ and $Q_{\texttt{C}}$ do not contain X.

If you are in a state satisfying P_C then you will end up in a state satisfying Q_C for $D.m$. This is not a standard behavioural subtype in the same way a function from $int \vee bool \to int \vee bool$ is not a subtype of $int \to int$. However, something that maps ints to ints *and* $bool$s to $bool$s is a subtype of $int \to int$. We need to be more expressive in how the output depends on the input: $\alpha \in \{int, bool\}.\alpha \to \alpha$, or $(int \to int) \wedge (bool \to bool)$. In our specifications the ghost variable X is used like the α to ensure that only the extra states of the pre-condition end in the extra states of the post-condition. This subtyping does satisfy specification compatibility:

$$\dfrac{\dfrac{\vdash \left\{ \begin{array}{r} (P_{\mathrm{C}} \wedge X = 1) \\ \vee\, (P_{\mathrm{D}} \wedge X = 2) \end{array} \right\} \_ \left\{ \begin{array}{r} (Q_{\mathrm{C}} \wedge X = 1) \\ \vee\, (Q_{\mathrm{D}} \wedge X = 2) \end{array} \right\}}{\vdash \left\{ X = 1 * \left(\begin{array}{r} (P_{\mathrm{C}} \wedge X = 1) \\ \vee\, (P_{\mathrm{D}} \wedge X = 2) \end{array} \right) \right\} \_ \left\{ \left(\begin{array}{r} (Q_{\mathrm{C}} \wedge X = 1) \\ \vee\, (Q_{\mathrm{D}} \wedge X = 2) \end{array} \right) * X = 1 \right\}}}{\dfrac{\dfrac{\vdash \{P_{\mathrm{C}} \wedge X = 1\} \_ \{Q_{\mathrm{C}} \wedge X = 1\}}{\vdash \{P_{\mathrm{C}} \wedge \_ = 1\} \_ \{Q_{\mathrm{C}} \wedge \_ = 1\}}}{\vdash \{P_{\mathrm{C}}\} \_ \{Q_{\mathrm{C}}\}}}$$

The key to the proof is the L-FRAME rule, which lets us preserve the value of the ghost variable across the call. In standard Hoare logic the invariance rule [4], or constancy rule [77], could be used. Specification compatibility allows us to model the specification inheritance used by Poetzsch-Heffter and Müller [72].

Given our assumption that all method and constructor environments are well-behaved (Definition 3.5.2), we can combine the static method call rule, L-CALL, with the static typing rule, L-STYPING, to derive the following useful rule concerning dynamic dispatch.

L-CALLDYN

$$\Gamma, \{P\}C.m(\overline{x})\{Q\} \vdash \{P \wedge \mathtt{this} \neq \mathtt{null}\} ret = \mathtt{this}.m(\overline{x})\{Q\}$$

provided `this` has static type C.

This derived rule allows us to reason about an unknown set of methods with a single specification. At link time we must check the methods are specification compatible.

Proposition 3.5.3 (Derivation of dynamic dispatch rule)**.** *Assuming the methods are specification compatible, then the dynamic dispatch method call rule* L-CALLDYN *is derivable.*

Proof. The following holds for any C that is a subtype of D as (1) is given by specification compatibility. Assume wlog Γ contains $\{P_C\}C.m\{Q_C\}$ and $\{P_D\}D.m\{Q_D\}$. The leaf is verified by the L-CALL rule.

$$\left.\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \{P_C \wedge \mathtt{this} : C\} ret = \mathtt{this}.m(\overline{x}); \{Q_C\}}}{\vdots\,(1)}}{\Gamma \vdash \{P_D \wedge \mathtt{this} : C[\overline{x}/\overline{y}]\} ret = \mathtt{this}.m(\overline{y})[\overline{x}/\overline{y}]\{Q_D[\overline{x}/\overline{y}]\}}}{\Gamma \vdash \{P_D \wedge \mathtt{this} \neq \mathtt{null} \wedge \mathtt{this} : C\} ret = \mathtt{this}.m(\overline{y})\{Q_D\}}\right\} (3.3)$$

Semantically, any statement with a precondition *false* can have any postcondition.

$$\left.\dfrac{\dfrac{}{\Gamma \vdash \{\mathtt{this} : D \wedge P_D\} ret = \mathtt{this}.m(\overline{y}); \{Q_D\}}}{\Gamma \vdash \{\mathit{false}\} ret = \mathtt{this}.m(\overline{y}); \{Q_D\}}}{\Gamma \vdash \{P \wedge \mathtt{this} = \mathtt{null} \wedge \mathtt{this} \neq \mathtt{null}\} ret = \mathtt{this}.m(\overline{y}); \{Q_D\}}\right\} (3.4)$$

Putting these together with the L-STYPE rule, we can derive the dynamic dispatch rule.

$$\dfrac{(3.3) \qquad (3.4)}{\Gamma \vdash \{P_D \wedge \mathtt{this} \neq \mathtt{null}\} ret = \mathtt{this}.m(\overline{y})\{Q_D\}}$$

```
class Tree extends Object {
  Tree l; Tree r;

  Tree(Tree l, Tree r) {
    super();
    this.l = l;
    this.r = r;
  }

  Tree clone() {
    Tree l,r,l2,r2;
    l = this.l;
    if(l != null) l2 = l.clone();
    else l2 = null;

    Tree r = this.r;
    if(r != null) r2 = r.clone();
    else r2 = null;

    Tree nt = new Tree(l2,r2);
    return nt;
  }
}
```

Figure 3.7: Tree example source code.

□

In Section 3.8, we will show that specification compatibility is too restrictive for many programs, and in Chapter 6 we will address this.

3.6 Example: Tree copy

Now we illustrate the use of this logic with a simple example of a tree class and a clone method. The source code is given in Figure 3.7.

Before we can specify the clone method, we extend our logic slightly to reason about trees, by adding the following predicate to the logic. In the next chapter we will formally define recursive predicates.

$$\mathit{tree}(i) \stackrel{\text{def}}{=} i = \texttt{null} \vee \exists j,k.\ i.\texttt{l} \mapsto j * i.\texttt{r} \mapsto k * \mathit{tree}(j) * \mathit{tree}(k)$$

Using this predicate, we can then specify the constructor and `clone` method as

Method	Pre-condition	Post-condition
`Tree(l,r)`	$\mathit{tree}(\texttt{l}) * \mathit{tree}(\texttt{r})$	$\mathit{tree}(\texttt{this})$
`Tree.clone()`	$\mathit{tree}(\texttt{this})$	$\mathit{tree}(\texttt{this}) * \mathit{tree}(\mathit{ret})$

Note: Although the specification does not prevent the `clone` method being called on a `null` objects, the L-CALL rule does.

The constructor is verified with its precondition and the fields defined by the class:

$\{\mathit{tree}(\texttt{l}) * \mathit{tree}(\texttt{r}) * \texttt{this.l} \mapsto \texttt{null} * \texttt{this.r} \mapsto \texttt{null}\}$
  $\{\mathit{true}\}$
  `super()`
  $\{\mathit{true}\}$
$\{\mathit{tree}(\texttt{l}) * \mathit{tree}(\texttt{r}) * \texttt{this.l} \mapsto \texttt{null} * \texttt{this.r} \mapsto \texttt{null}\}$

We verify the `super` call using the frame rule. The superclass is `Object`, and `Object`'s constructor's specification is $\{\mathit{true}\}\_\{\mathit{true}\}$.

$\{\mathit{tree}(\texttt{l}) * \mathit{tree}(\texttt{r}) * \texttt{this.l} \mapsto \texttt{null} * \texttt{this.r} \mapsto \texttt{null}\}$
  $\{\texttt{this.l} \mapsto \texttt{null}\}$
  `this.l = l;`
  $\{\texttt{this.l} \mapsto \texttt{l}\}$
$\{\mathit{tree}(\texttt{l}) * \mathit{tree}(\texttt{r}) * \texttt{this.l} \mapsto \texttt{l} * \texttt{this.r} \mapsto \texttt{null}\}$

We verify the assignment to the `l` field using the frame rule and the assignment rule.

$\{tree(\texttt{l}) * tree(\texttt{r}) * \texttt{this.l} \mapsto \texttt{l} * \texttt{this.r} \mapsto \texttt{null}\}$
$\quad \{\texttt{this.r} \mapsto \texttt{null}\}$
```
    this.r = r;
```
$\quad \{\texttt{this.r} \mapsto \texttt{r}\}$
$\{tree(\texttt{l}) * tree(\texttt{r}) * \texttt{this.l} \mapsto \texttt{l} * \texttt{this.r} \mapsto \texttt{r}\}$

and the `r` field in the same way.

$\{tree(\texttt{l}) * tree(\texttt{r}) * \texttt{this.l} \mapsto \texttt{l} * \texttt{this.r} \mapsto \texttt{r}\}$
$\{tree(\texttt{this})\}$

Finally, we use the rule of consequence to put the post-condition in as it appears in the specification.

We can verify the `clone` method as follows. First we verify the code for copying the left hand subtree.

$\{\texttt{this.l} \mapsto j * tree(j)\}$
$\quad \{\texttt{this.l} \mapsto j\}$
```
    l = this.l;
```
$\quad \{\texttt{this.l} \mapsto \texttt{l} \wedge \texttt{l} = j\}$
$\{\texttt{this.l} \mapsto \texttt{l} \wedge \texttt{l} = j * tree(j)\}$
$\{\texttt{this.l} \mapsto \texttt{l} * tree(\texttt{l})\}$
$\quad \{tree(\texttt{l})\}$
```
    if(l != null)
```
$\quad\quad \{tree(\texttt{l}) \wedge \texttt{l} \neq \texttt{null}\}$
```
      l2 = l.clone();
```
$\quad\quad \{tree(\texttt{l}) * tree(\texttt{l2})\}$
```
    else
```
$\quad\quad \{tree(\texttt{l}) \wedge \texttt{l} = \texttt{null}\}$
$\quad\quad \{tree(\texttt{l}) \wedge \texttt{l} = \texttt{null} \wedge \texttt{null} = \texttt{null}\}$
```
      l2 = null;
```
$\quad\quad \{tree(\texttt{l}) \wedge \texttt{l} = \texttt{null} \wedge \texttt{l2} = \texttt{null}\}$
$\quad\quad \{tree(\texttt{l}) * tree(\texttt{l2})\}$
$\quad \{tree(\texttt{l}) * tree(\texttt{l2})\}$
$\{\texttt{this.l} \mapsto \texttt{l} * tree(\texttt{l}) * tree(\texttt{l2})\}$

We can provide a similar proof for the right hand subtree. Using these proofs we can complete the proof as

$\{tree(\texttt{this}) \wedge \texttt{this} : \texttt{Tree}\}$
$\{\texttt{this.l} \mapsto j * \texttt{this.r} \mapsto k * tree(j) * tree(k)\}$
```
    Tree l2,r2,l,r;
```
$\{\texttt{this.l} \mapsto j * \texttt{this.r} \mapsto k * tree(j) * tree(k)\}$
$\quad \{\texttt{this.l} \mapsto j * tree(j)\}$
```
    l = this.l;
    if(l != null)
      l2 = l.clone();
    else
      l2 = null;
```
$\quad \{\texttt{this.l} \mapsto \texttt{l} * tree(\texttt{l}) * tree(\texttt{l2})\}$
$\{\texttt{this.l} \mapsto \texttt{l} * \texttt{this.r} \mapsto k * tree(\texttt{l}) * tree(k) * tree(\texttt{l2})\}$
$\quad \{\texttt{this.r} \mapsto k * tree(k)\}$
```
    r = this.r;
    if(r != null)
      r2 = r.clone();
```

```
    else
      r2 = null;
    {this.r↦r * tree(r) * tree(r2)}
{this.l↦l * this.r↦r * tree(l) * tree(r) * tree(l2) * tree(r2)}
{tree(this) * tree(l2) * tree(r2)}
    {tree(l2) * tree(r2)}
    Tree nt = new Tree(l2,r2);
    {tree(nt)}
{tree(this) * tree(nt)}
    return nt;
{tree(this) * tree(ret)}
```

Again the frame rule is used to allow the additional state to be preserved by the call, without any additional proof burden.

3.7 Semantics

Now we will relate the semantics of the previous chapter with the logic given in this chapter, and hence prove its soundness. The typing of configurations from the previous chapter would consider heaps containing objects with missing fields as invalid. However, the logic works explicitly with partially defined objects so we relax the conditions on a heap being well-formed, originally given on page 34. We no longer require that every field of an object exists, but still require it to point to the correct type if it does. This relaxed rule is as follows:

$$\text{TH-OBJECTOK} \quad \frac{\forall f \in dom(H(o,\_)).\quad \delta, H \vdash H(o,f) : \delta_{\mathrm{F}}(H(o))(f)}{\delta, H \vdash o \ \mathsf{ok}}$$

This weakening of the type system leads to two additional reduction rules

$$\text{E-FIELDACCESSFAIL} \quad (H, VS, o.f, FS) \rightarrow (H, VS, \mathbf{NSFE}, FS) \quad \text{provied } (o,f) \notin dom(H)$$

$$\text{E-FIELDWRITEFAIL} \quad (H, VS, o.f = v;, FS) \rightarrow (H, VS, \mathbf{NSFE}, FS) \quad \text{provided } (o,f) \notin dom(H)$$

These exceptional states correspond to the `NoSuchFieldException` in Java.

To simplify the proofs we want to consider $(H, VS, \overline{s}, [])$ as well-typed when $\overline{s}$ is a method body, i.e. ends in `return`. This is forbidden by the type rules in the previous chapter. We modify the case for the empty framestack to accept any variable scoping.

$$\text{TF-STACKEMPTY} \quad \delta, H, VS \vdash [] : \tau \rightarrow \tau$$

and relax the definition of a completed configuration:

$$compl(config) \stackrel{\text{def}}{=} ;\ \exists H, VS, \zeta, v.(H, VS, \zeta, []) = config \vee (H, VS, v, []) = config$$

where ζ ranges over `return` v; and ;.

The logic will be used to remove the runtime exceptions: **NSFE** and the null pointer and class cast exception from the previous chapter. We define safety, in the style of Yang [88], to mean a configuration cannot get to an exceptional state.

Definition 3.7.1 (Safety)**.** A configuration is safe iff it cannot reduce to a stuck state that is not completed configuration, i.e.

$$config\ \mathbf{safe} \stackrel{\text{def}}{=} \forall config'.(config \rightarrow^* config' \ \wedge \ stuck(config') \ \Rightarrow \ compl(config'))$$

Note: Safety is a stronger property than type safety enforced in the previous chapter: a safe program will not dereference null, or perform an invalid cast.

We then prove the standard properties, following Yang [88], of the operational semantics for the soundness of the frame rule. For compactness we use $(H,\, VS, F, FS) * H'$ as a shorthand for $(H * H',\, VS, F, FS)$.

Lemma 3.7.2 (Safety monotonicity)**.** *If* $config$ ***safe*** *then for any disjoint extension* H' *the extended configuration is safe:* $config * H'$ ***safe****.*

Proof. Assume $config * H'$ reduces to an error state. Then $config$ must also reduce, possibly earlier, to an error state. But we know $config$ **safe**, so the original assumption must be wrong. □

Lemma 3.7.3 (Heap locality)**.** *If* $(H_0,\, VS, \overline{s}, [])$ ***safe*** *and* $(H_0 * H_1,\, VS, \overline{s}, []) \rightarrow^* (H',\, VS', \zeta, [])$ *then* $\exists H_0'.H_0' * H_1 = H'$ *and* $(H_0, VS, \overline{s}, []) \rightarrow^* (H_0',\, VS', \zeta, [])$.

Proof. We prove

$$config \textbf{ safe} \;\wedge\; config * H_0 \rightarrow (H,\, VS, \overline{s}, FS) \Rightarrow \exists H_1.config \rightarrow (H_1,\, VS, \overline{s}, FS) \;\wedge\; H_1 * H_0 = H$$

by case analysis of the reduction relation. □

In what follows we find it useful to use a subset of the reduction relation that satisfies certain properties.

Definition 3.7.4 (Predicated reduction)**.** Assuming we have a reduction relation $config \rightarrow config$ and a predicate, ϕ, on a configuration, we define a restricted reduction relation where every configuration must satisfy a predicate as

$$\frac{\phi(config)}{config \xrightarrow{\phi}{}^* config} \qquad \frac{\phi(config'') \quad config \xrightarrow{\phi}{}^* config' \quad config' \rightarrow config''}{config \xrightarrow{\phi}{}^* config''}$$

We find it particularly useful to induct over the recursive depth of a computation.

Definition 3.7.5 (Recursive depth)**.** We define recursive depth, $rdepth$, as the number of `return` statements in a given framestack:

$$rdepth(FS) = \begin{cases} 0 & FS = [] \;\vee\; FS = \texttt{return}\, e; \circ [] \\ 1 + rdepth(FS') & FS = \texttt{return}\, e; \circ FS' \;\wedge\; FS' \neq [] \\ rdepth(FS') & FS = F \circ FS' \;\wedge\; F \neq \texttt{return}\, e; \end{cases}$$

We consider $\texttt{return}\, e; \circ []$ to have a zero recursive depth as the `return` will remain on the stack as there is no where to return to. We extend this to configurations in the obvious way, and define a predicate $rdepth_n$ that holds if the depth is less than or equal to n.

Lemma 3.7.6 (Bounded depth reductions contains all finite reductions)**.**
$config \rightarrow^* config'$ *iff there exists an* n *such that* $config \xrightarrow{rdepth_n}{}^* config'$.

Proof. No finite reduction sequence can generate an infinite depth frame stack: each reduction step can introduce at most one `return`. □

Rather than proving properties about the reduction relation in all possible contexts, we will show that the context does not affect the reduction. First we define a relation for when a computation does not affect its context.

Definition 3.7.7 (Stack contains predicate)**.** We define a predicate, $ends$, for checking a framestack ends in a particular framestack.

$$ends_{FS,VS} \stackrel{\text{def}}{=} \{(H, VS' \circ VS, CF, FS') \mid FS' = FS'' \circ FS \ \wedge \ VS' = MS \circ VS''\}$$

Lemma 3.7.8 (Context change)**.**

$$(H, MS \circ [], \overline{s}, []) \rightarrow^* (H', MS' \circ [], \zeta, []) \iff (H, MS \circ VS, \overline{s}, FS) \xrightarrow{ends_{FS,VS}}{}^*(H', MS' \circ VS, \zeta, FS)$$

Proof. We prove the stronger fact:

$$(H, VS \circ VS_1, CF, FS \circ FS_1) \xrightarrow{ends_{FS_1,VS_1}}{}^*(H', VS' \circ VS_1, CF', FS'' \circ FS_1) \Rightarrow (H, VS \circ VS_2, CF, FS \circ FS_2) \xrightarrow{ends_{FS_2,VS_2}}{}^*(H', VS' \circ VS_2, CF', FS'' \circ FS_2)$$

By induction on the length of the reduction. The base case holds trivially. The inductive case requires

$$(H, VS \circ VS_1, CF, FS \circ FS_1) \rightarrow (H', VS' \circ VS_1, CF', FS' \circ FS_1) \Rightarrow (H, VS \circ VS_2, CF, FS \circ FS_2) \rightarrow (H', VS' \circ VS_2, CF', FS' \circ FS_2)$$

This can be shown by case analysis of the rules. □

This lemma shows that if we consider the empty context, then that is equivalent to considering all computations that ignore their context.

Lemma 3.7.9 (Method reduction ignores context)**.**

$$(H, MS \circ VS, \overline{s}, FS) \rightarrow^* (H'', VS'', CF'', FS'') \wedge \neg ends_{FS,VS}(H'', VS'', CF'', FS'') \Rightarrow \exists H', MS', \zeta. \quad (H, MS \circ VS, \overline{s}, FS) \xrightarrow{ends_{FS,VS}}{}^*(H', MS' \circ VS, \zeta, FS)$$

Proof. We can ignore exceptional reduction as it is ruled out by the assumption.[4] Consider the last state in the reduction that satisfies the $ends$ predicate. The only reduction rules that can leave the $ends$ relation are: E-RETURN, E-SKIP and E-SUB, but these all operate on states of the form $(H', MS' \circ VS, \zeta, FS)$. □

By showing method reduction ignores its context we can see it is valid to only consider method reduction in the empty context.

Lemma 3.7.10 (Expression evaluation)**.** $[\![e]\!]_{VS,I}$ *respects the operational semantics,*

$$(H, VS, e, []) \rightarrow^* (H, VS, v, []) \iff v = [\![e]\!]_{VS,I}$$

Proof. Trivial as e can only be `null` or a variable. □

The rules given earlier used substitution to model assignment; next we link update and substitution semantically.

[4] If we consider exceptions to be caught, then this would not hold. We would need to extend ζ to exceptions.

Definition 3.7.11 (Substitution on a stack)**.** Let θ be a substitution of the form: $[e_1, \ldots, e_n/x_1, \ldots, x_n]$. We define a relation between stacks

$$VS, I \prec_\theta VS', I' \quad \stackrel{\text{def}}{=} \quad \forall e.\ [\![e]\!]_{VS,I} = v \Leftrightarrow [\![\theta(e)]\!]_{VS',I'} = v$$

It is important to note $VS[x \mapsto [\![e]\!]_{VS,I}] \prec_{[e/x]} VS$ holds, that is this relation contains the update on stacks under the syntactic substitution used in the logic.

Lemma 3.7.12 (Substitution preserves truth)**.** *If VS and I are related by θ to VS' and I' then $\theta(P)$ is satisfied by VS, H, I iff P is satisfied by VS', H, I'.*

Proof. We prove

$$VS, I \prec_\theta VS', I' \quad \Rightarrow \quad (VS, H, I \models P \ \Leftrightarrow \ VS', H, I' \models \theta(P))$$

by induction on the structure of P. Holds at leaves by definition of the relation. □

The following lemma is used to prove L-VARSUBST is sound.

Lemma 3.7.13 (Substitution preserves reduction)**.**

$$VS_1 \prec_\theta VS_2 \Rightarrow ((H,\ VS_1, \overline{s}, []) \ \textbf{\textit{safe}} \Leftrightarrow (H,\ VS_2, \theta(\overline{s}), []) \ \textbf{\textit{safe}}) \tag{3.5}$$

$$(H,\ VS_1, \overline{s}, []) \rightarrow^* (H',\ VS_1', \zeta, []) \ \wedge \ VS_1 \prec_\theta VS_2 \Rightarrow ((H,\ VS_2, \theta(\overline{s}), []) \rightarrow^* (H',\ VS_2', \zeta, []) \ \wedge \ VS_1' \prec_\theta VS_2') \tag{3.6}$$

$$(H,\ VS_2, \theta(\overline{s}), []) \rightarrow^* (H',\ VS_2', \zeta, []) \ \wedge \ VS_1 \prec_\theta VS_2 \Rightarrow ((H,\ VS_1, \overline{s}, []) \rightarrow^* (H',\ VS_1', \zeta, []) \ \wedge \ VS_1' \prec_\theta VS_2') \tag{3.7}$$

provided $\theta = [e_1, \ldots, e_n/x_1, \ldots, x_n]$;

- *if x_i is modified by $\overline{s}$, then e_i is just a variable not free in any other e_j;*
- *if x_i is in the free variables of $\overline{s}$, then e_i is restricted to program expressions, expressions not containing ghost variables; and*
- *if $\overline{s}$ contains* `return` *then $\theta(ret) = ret$.*

Proof. We can see that this holds for expressions from the definition of the substitution relation. Hence we can consider statements with only values and no expressions. The method call and constructor cases follow directly from Lemmas 3.7.8 and 3.7.9. The only case of interest is the update variable. We must show $VS_1[x \mapsto v] \prec_\theta VS_2[\theta(x) \mapsto v]$, which holds trivially. □

Now we define the semantics of the proof system.

Definition 3.7.14. We write $\Gamma \models \{P\}\overline{s}\{Q\}$ to mean if the specifications in Γ are true of the methods and constructors in the program, then so is $\{P\}\overline{s}\{Q\}$, i.e.

$$\Gamma \models \{P\}\overline{s}\{Q\} \quad \stackrel{\text{def}}{=} \quad \models \Gamma \ \Rightarrow \ \models \{P\}\overline{s}\{Q\}$$

where

$$\models \Gamma \quad \stackrel{\text{def}}{=} \quad \forall \{P\}\eta(\overline{x})\{Q\} \in \Gamma.\ \models \{P\}\eta(\overline{x})\{Q\}$$

$$\models \{P\}C.m(\overline{x})\{Q\} \quad \stackrel{\text{def}}{=} \quad \models \{P \wedge \texttt{this} : C\}\overline{s}\{Q\} \qquad \text{where } mbody(C, m) = (\overline{x}, \overline{s})$$

$$\models \{P\}C(\overline{x})\{Q\} \quad \stackrel{\text{def}}{=} \quad \models \{P * new(\texttt{this}, C)\}\overline{s}\{Q\} \qquad \text{where } cnbody(C) = (\overline{x}, \overline{s})$$

$$\models \{P\}\overline{s}\{Q\} \stackrel{\text{def}}{=} \forall VS, H, I.$$
$$VS, H, I \models P \ \wedge \ \exists \tau(\delta \vdash (H,\ VS, \overline{s}, []) : \tau) \Rightarrow \left(\begin{array}{l} (H,\ VS, \overline{s}, []) \ \textbf{safe} \\ \wedge\ ((H,\ VS, \overline{s}, []) \rightarrow^* (H,\ VS', \zeta, []) \Rightarrow (VS', H', I \uplus [\![\zeta]\!] \models Q) \end{array} \right)$$

where $[\![\texttt{return}\, v;]\!] = \{ret \mapsto v\}$ and $[\![;]\!] = \emptyset$.

Note: The semantics are defined in an implicit context of a program, π, and a class table, δ.

Now we come to the key contribution of this chapter: the soundness of the logic.

Theorem 3.7.15 (Soundness)**.** *If* $\Gamma \vdash \{P\}\overline{s}\{Q\}$*, then* $\Gamma \models \{P\}\overline{s}\{Q\}$

Proof. As we have recursive method calls we must prove this initially by induction on the recursive depth [71, 83].[5] We define

$$\begin{array}{l}\models_n \{P\}\overline{s}\{Q\} \stackrel{\text{def}}{=} \forall VS, H, I. \\ \quad VS, H, I \models P \ \wedge \ \exists \tau(\delta \vdash (H, VS, \overline{s}, []) : \tau) \\ \qquad\qquad \Rightarrow \left(\begin{array}{l} (H, VS, \overline{s}, []) \textbf{ safe} \\ \wedge\ ((H, VS, \overline{s}, []) \xrightarrow{rdepth_n}{}^{*} (H, VS', \zeta, []) \Rightarrow (VS', H', I \uplus [\![\zeta]\!] \models_{\Delta_f} Q) \end{array} \right)\end{array}$$

By Lemma 3.7.6, $\models \{P\}\overline{s}\{Q\}$ is equivalent to $\exists n.\ \models_n \{P\}\overline{s}\{Q\}$. Hence, we prove

$$\Gamma \vdash \{P\}\overline{s}\{Q\} \ \Rightarrow \left(\begin{array}{l} (\forall n. \ \models_n \Gamma \ \Rightarrow \ \models_{n+1} \{P\}\overline{s}\{Q\}) \\ \qquad\qquad \wedge \ \models_0 \{P\}\overline{s}\{Q\} \end{array} \right)$$

$$\Gamma \Vdash \Gamma' \Rightarrow (\forall n \models_n \Gamma \ \Rightarrow \ \models_{n+1} \Gamma') \ \wedge \ \models_0 \Gamma'$$

by rule induction. This induction is valid by Lemma 3.7.6. The definition introduction rules, Figure 3.5, all follow directly from their definition. The frame rule follows from Lemmas 3.7.2 and 3.7.3 and the free variable restriction. The variable substitution rule follows from Lemma 3.7.13. The L-STYPE rule follows directly from the definition of a well-formed configuration. Sequencing follows from the type preservation proof and the usual argument. The L-CALL, L-NEW, L-SUPER rules all follow from induction on the recursive depth, and that they do not modify their parameters. □

Finally in this section, we show that the transformation preserves the semantics, but first we must show that adding fresh variables does not affect the reduction.

Lemma 3.7.16 (Fresh variables or new blocks do not affect reduction)**.** *Adding a fresh variable, or a fresh block of variables, does not affect expression or statement reduction, i.e. if*

$$(H, MS \circ [], CF, []) \rightarrow^{*} (H', MS' \circ [], \zeta, [])$$

where $MS = MS_a \circ BS \circ MS_b$*,* $MS' = MS'_a \circ BS \circ MS'_b$*, and* CF *is a statement sequence or an expression.*

then

$$(H, MS_1 \circ [], CF, []) \rightarrow^{*} (H', (MS'_1 \circ [], \zeta, [])$$

where $MS_1 = MS_a \circ BS[z \mapsto v] \circ MS_b$ *and* $MS'_1 = MS'_a \circ BS[z \mapsto v] \circ MS'_b$ *and* z *not mentioned in* CF*.*

and

$$(H, (MS_2 \circ \{\} \circ BS \circ MS_2) \circ VS, CF, []) \quad \rightarrow^{*} \quad (H', (MS'_1 \circ \{\} \circ BS' \circ MS'_2) \circ VS, \zeta, [])$$

where $MS_2 = MS_a \circ \{\} \circ BS \circ MS_b$ *and* $MS'_1 = MS'_a \circ \{\} \circ BS \circ MS'_b$ *and if* CF *is of the form* $s_1 \ldots Cx; \ldots s_2$ *then* MS_a *is non-empty.*

Note: Above we abuse the use of $\circ$ to additionally mean stack append, in addition to cons.

[5] The $(H, VS, \overline{s}, [])$ **safe** does not depend on the recursive depth, so even programs that do not terminate are safe.

```
class Cell extends Object {
  Object cnts;
  ...
  void set(Object o) {
    this.cnts = o;
  }

  Object get() {
    Object temp;
    temp = this.cnts;
    return temp;
  }
}
```

```
class Recell extends Cell {
  Object bak;
  ...
  void set(Object o) {
    Object temp;
    temp = this.cnts;
    this.bak = temp;
    this.cnts = o;
  }
}
```

Figure 3.8: Source of Cell and Recell (Constructors omitted)

Proof. By induction on the structure of CF. □

Lemma 3.7.17 (Transformation preserves semantics)**.** *The transformation defined in Figure 3.2 preserves the semantics: i.e.*

$$\pi \vdash (H,\, VS, \overline{s}, []) \rightarrow^* (H',\, VS', ;, ;, []) \Rightarrow (\!|\pi|\!) \vdash (H,\, VS, (\!|\overline{s}|\!), []) \rightarrow^* (H',\, VS'+, ;, ;, [])$$

where $VS+$ is either VS or VS extended with a single variable in the top frame.

Proof. We prove the following

$$\Phi(n) \wedge \Psi(n) \Rightarrow \Phi(n+1) \wedge \Psi(n+1)$$

where

$$\begin{aligned}\Phi(n) \stackrel{\text{def}}{=} \quad & \forall H,\, VS, \pi, \overline{s}, H',\, VS'. \\ & \pi \vdash (H,\, VS, \overline{s}, []) \xrightarrow{rdepth_n}{}^*(H',\, VS', \zeta, []) \Rightarrow \\ & \qquad (\!|\pi|\!) \vdash (H,\, VS, (\!|\overline{s}|\!), []) \xrightarrow{rdepth_n}{}^*(H',\, VS'+, \zeta, [])\end{aligned}$$

and

$$\begin{aligned}\Psi(n) \stackrel{\text{def}}{=} \quad & \forall \pi, H,\, VS, e, H',\, VS', v, z. \\ & \pi \vdash (H,\, VS, e, []) \xrightarrow{rdepth_n}{}^*(H',\, VS', v, []) \Rightarrow \\ & (\!|\pi|\!) \vdash (H,\, VS + [z \mapsto \_, C], (\!|e|\!)_z, []) \xrightarrow{rdepth_n}{}^*(H',\, VS' + [z \mapsto v, C], ;, []) \end{aligned}$$

where $((BS \circ MS) \circ VS) + [x \mapsto (v, C)]$ is a shorthand for $(BS[x \mapsto (v, C)] \circ MS) \circ VS$.

We can prove this by induction on the translation, and use Lemma 3.7.16 to deal with the variables and block scopes the translation introduces. □

3.8 Encapsulation and inheritance

Although this logic allows reasoning about programs that manipulate pointer datastructures, it does not provide support for abstraction (encapsulation) or inheritance. Consider the definition of the `Cell` class given in Figure 3.8. We can provide the obvious specifications to the `Cell` class's methods:

Method	Pre-condition	Post-condition
`Cell.set(o)`	$\texttt{this.cnts} \mapsto X$	$\texttt{this.cnts} \mapsto \texttt{o}$
`Cell.get()`	$\texttt{this.cnts} \mapsto X$	$\texttt{this.cnts} \mapsto X \wedge ret = X$

These specifications can easily be verified using the rules described earlier. However, the specifications are sensitive to changes in the implementation. Renaming the `cnts` field would alter all the proofs using the `Cell`, even if they only used the `get` and `set` methods. We are lacking the common abstractions afforded to us by object-oriented programming.

Worse still, we cannot provide support for inheritance. Consider the standard subtype of `Cell`, `Recell` [1], also given in Figure 3.8. Consider giving its `set` method the obvious specification.

Method	Pre-condition	Post-condition
`Recell.set(o)`	$\texttt{this.cnts} \mapsto X$	$\texttt{this.cnts} \mapsto \texttt{o}$
	$* \ \texttt{this.bak} \mapsto \_$	$* \ \texttt{this.bak} \mapsto X$

This code is clearly a well-behaved subtype of `Cell` yet it is not specification compatible with its parent, because `Recell.set` has a footprint of two fields, while `Cell.set` has a footprint of one field. This is an example of the extended state problem mentioned in Chapter 1. The only way to gain specification compatibility is to modify the `Cell`'s specification to account for the `Recell`'s specification, i.e.

Method	Pre-condition	Post-condition
`Cell.set(o)`	$\texttt{this.cnts} \mapsto X$	$\texttt{this.cnts} \mapsto \texttt{o}$
	$* \ (\texttt{this} : \texttt{Recell}$	$* \ (\texttt{this} : \texttt{Recell}$
	$\Rightarrow \texttt{this.bak} \mapsto \_)$	$\Rightarrow \texttt{this.bak} \mapsto X)$
`Recell.set(o)`	$\texttt{this.cnts} \mapsto X$	$\texttt{this.cnts} \mapsto \texttt{o}$
	$* \ (\texttt{this} : \texttt{Recell}$	$* \ (\texttt{this} : \texttt{Recell}$
	$\Rightarrow \texttt{this.bak} \mapsto \_)$	$\Rightarrow \texttt{this.bak} \mapsto X)$

Adapting specifications in this way is perhaps feasible in small programs, but does not yield a scalable solution. In particular it would not be possible for open programs, i.e. programs where only part of the source is available.

The remainder of this thesis presents solutions to the complex problems of encapsulation and inheritance. In the next chapter we add support for encapsulation to the logic using a novel concept of an abstract predicate. In Chapter 6 we extend this concept to additionally deal with inheritance. The result is, we claim, the first program logic that is expressive enough to reason about real-world code written in realistic object-oriented languages, such as Java.

4

Abstract predicates

In the previous chapter we presented a logic for Java. However, that logic was not powerful enough to express the abstraction that encapsulation provides to object-oriented programs. In this chapter we extend the logic from the previous chapter with reasoning that supports encapsulation.

Various researchers have proposed the enrichment of a program logic to view the data abstractly (as in data groups [54]), or the methods/procedures abstractly (as in method groups [52, 80]). In contrast, we propose to add the abstraction to the logical framework itself, by introducing the notion of an *abstract predicate*. Intuitively, abstract predicates are used like abstract data types. Abstract data types have a name, a scope and a concrete representation. Operations defined within this scope can freely exchange the datatype's name and representation, but operations defined outside the scope can only use the datatype's name. Similarly abstract predicates have a name and a formula. The formula is *scoped*: code verified inside the scope can use both the predicate's name and its body, while code verified outside the scope must treat the predicate atomically. We do not reason directly about Java access modifiers `private` and `protected`, but provide a more abstract notion of scoping.

In separation logic it is common to use inductively defined predicates to represent data types, for example §3.6. In essence we allow predicates to additionally encapsulate state and not just represent it. This gives us two key advantages: (1) the impact of changing a predicate is easy to define; and (2) by encapsulating state we are able to reason about *ownership transfer* [66].

The rest of this chapter is structured as follows. We begin by giving the proof rules associated with abstract predicates. In §4.2, we provide examples to illustrate the use and usefulness of abstract predicates. Then in §4.3 we give the semantics and prove the new rules are sound. We conclude with a discussion of related work.

Note: A different version of this work, which uses a simple language of functions rather than Java, was presented at POPL [67].

4.1 Proof rules

We extend the assertion language given in Chapter 3 with predicates. We write α to range over predicate names and use a function $arity()$ from predicate names to a natural number representing the predicate's arity.

We extend judgements to contain predicate definitions as follows:

$$\Lambda; \Gamma \vdash \{Q_1\}\overline{s}\{Q_2\}$$

This is read "the statements, $\overline{s}$, satisfy the specification $\{Q_1\}\_\{Q_2\}$, given the method and constructor hypotheses, Γ, and predicate definitions, Λ." Predicate definitions are given by the following grammar:

$$\Lambda \quad := \quad \epsilon \mid (\alpha(\overline{x}) \stackrel{\text{def}}{=} P),\ \Lambda$$

For the predicate definitions, Λ, to be well-formed we require that each predicate, α, is contained at most once; the free variables of the body, P, are contained in the arguments, $\overline{x}$; and P is a positive formula.[1] We will only consider well-formed Λ. When it simplifies the presentation, we will treat Λ as a partial function from predicate names to formulae. We define $\Lambda(\alpha)[\overline{e}]$ as $P[\overline{e}/\overline{x}]$ where Λ contains $\alpha(\overline{x}) \stackrel{\text{def}}{=} P$. Additionally, we extend the definition rules in Figure 3.5 (page 51) to include the predicate definitions.

Note: In this thesis we only use abstract predicates to represent inductively defined datastructures (such as lists and trees), so the definitions are all continuous. However, the rules and semantics do not require the definitions to be continuous, only that the definitions have a fixed point.

The exchange between definitions and names occurs in the following two axioms[2] concerning abstract predicates:

OPEN	$(\alpha(\overline{x}) \stackrel{\text{def}}{=} P), \Lambda$	$\models$	$\alpha(\overline{e}) \Rightarrow P[\overline{e}/\overline{x}]$
CLOSE	$(\alpha(\overline{x}) \stackrel{\text{def}}{=} P), \Lambda$	$\models$	$P[\overline{e}/\overline{x}] \Rightarrow \alpha(\overline{e})$

These axioms embody our intuition that if (and only if) an abstract predicate is in scope (that is, it is contained in Λ) then we can freely move between its name and its definition. These axioms are used in the rule of consequence, L-CONSEQUENCE.

L-CONSEQUENCE
$$\frac{\Lambda \models P \Rightarrow P' \quad \Lambda; \Gamma \vdash \{P'\}C\{Q'\} \quad \Lambda \models Q' \Rightarrow Q}{\Lambda; \Gamma \vdash \{P\}C\{Q\}}$$

Taking the analogy with abstract datatypes: the rule of consequence can be seen as a subtyping rule, and the axioms open and close are the coercions, or subtyping relation, allowed when the datatype is in scope.

Next we present the rule to enforce the scoping of predicate definitions:

L-DINTROABS
$$\frac{\Lambda', \Lambda; \Gamma, \Gamma' \Vdash \Gamma' \quad \Lambda; \Gamma, \Gamma' \vdash \{P\}\overline{s}\{Q\}}{\Lambda; \Gamma \vdash \{P\}\overline{s}\{Q\}}$$

provided P, Q, Γ and Λ' do not contain the predicate names in $dom(\Lambda')$; and $dom(\Lambda)$ and $dom(\Lambda')$ are disjoint.

This rule allows a class, or package, author to use the definition of some abstract predicates, yet the client can only use the abstract predicate names. The methods and constructors in Γ' are within the scope of the predicates defined in Λ hence verifying the the bodies can use the predicate definitions. The client code, $\overline{s}$, is *not* in the scope of the predicates, so it can only use

[1] A positive formula is one where predicate names appear only under an even number of negations. This ensures that a fixed point can be found; this is explained in further detail in Definition 4.3.3 and Lemma 4.3.4.

[2] We present them as semantic implications as a complete proof theory for separation logic is not known.

the predicates atomically through the specifications in Γ'. The predicate names cannot occur in the conclusion's specification, $\{P\}\_\{Q\}$.

The side-conditions for this rule prevent both the predicates escaping the scope of the module, and repeated definitions of a predicate.

In fact, the previous definition introduction rule, L-DINTROABS, is a derived rule in our system. It is derived from the standard definition introduction rule and two new rules for manipulating abstractions. The first, L-ABSWEAK, allows the introduction of new definitions.

$$\text{L-ABSWEAK}\quad \frac{\Lambda;\Gamma \vdash \{P\}C\{Q\}}{\Lambda,\Lambda';\Gamma \vdash \{P\}C\{Q\}} \quad \text{provided } dom(\Lambda') \text{ and } dom(\Lambda) \text{ are disjoint}$$

The second, L-ABSELIM, allows any unused predicate to be removed.

$$\text{L-ABSELIM}\quad \frac{\Lambda,\Lambda';\Gamma \vdash \{P\}C\{Q\}}{\Lambda;\Gamma \vdash \{P\}C\{Q\}} \quad \text{provided the predicate names in } P, Q, \Gamma \text{ and } \Lambda \text{ are not in } dom(\Lambda').$$

We can derive the L-DINTROABS rule in the following way:

$$\text{L-DINTRO}\,\frac{\Lambda,\Lambda';\Gamma,\Gamma' \Vdash \Gamma' \qquad \text{L-ABSWEAK}\,\dfrac{\Lambda;\Gamma,\Gamma' \vdash \{P\}\overline{s}\{Q\}}{\Lambda,\Lambda';\Gamma,\Gamma' \vdash \{P\}\overline{s}\{Q\}}}{\text{L-ABSELIM}\,\dfrac{\Lambda,\Lambda';\Gamma \vdash \{P\}\overline{s}\{Q\}}{\Lambda;\Gamma \vdash \{P\}\overline{s}\{Q\}}}$$

4.2 Examples

To demonstrate the utility of abstract predicates, we shall present a couple of examples.

4.2.1 Connection pool

Our first example is a database connection pool. Constructing a database connection is generally an expensive operation, so this cost is reduced by pooling connections using the object pool design pattern [42, Chapter 5]. Programs regularly access several different databases, hence we require multiple connection pools and dynamic instantiation. The connection pool must prevent the connections being used after they are returned: ownership must be transferred between the client and the pool.

We assume a library constructor, `Conn(db)`, to construct a database connection. This routine takes a single parameter that specifies the database, and returns a handle to a connection. The specification of this constructor uses a predicate $conn$ to represent the state of the connection.

$\{true\}$ `Conn(db)` $\{conn(\texttt{this}, \texttt{db})\}$

Note: In a more realistic implementation, such as JDBC [35], several arguments would be used to specify how to access a database. Additionally an instance of the Factory pattern would be used so that the correct database driver could provide the connection.

We use two classes, given in Figure 4.1, to represent the state of a connection pool: `ConnPool` represents the connection pool, and `ConnList` holds the connections. We reflect this in the logic by defining two abstract predicates: $cpool$ and $clist$. The $cpool$ predicate is used to represent a connection pool; and the $clist$ predicate is used inside the $cpool$ to represent a list of connection

```
class CList {
  Conn hd;
  CList tl;
  CList(Conn hd, CList tl) {
    this.hd = hd;
    this.tl = tl;
  }
}
class ConnPool {
  Object db; CList l;
  ConnPool(Object db) {
    this.db = db;
    this.l = null;
  }
  Conn getConn() {
    CList l;
    Conn c;
    Object db;
    l = this.l;
    db = this.db;
    if(l == null) {
      c = new Conn(db);
    } else {
      c = l.hd;
      l = l.tl;
      this.l = l;
    }
    return c;
  }
  void freeConn(Conn y) {
    CList t; CList n;
    t = this.l;
    n = new CList(y,t);
    this.l = n;
  }
}
```

Figure 4.1: Source code for the connection pool

predicates.

$$cpool(x, db) \stackrel{\text{def}}{=} \exists i.\ x.\mathtt{l} \mapsto i * x.\mathtt{db} \mapsto db * clist(i, db)$$

$$clist(x, db) \stackrel{\text{def}}{=} x = \mathtt{null} \vee (\exists ij.\ x.\mathtt{hd} \mapsto i * x.\mathtt{tl} \mapsto j * conn(i, db) * clist(j, db))$$

The connection pool has three operations: `ConnPool` to construct a pool; `getConn` to get a connection; and `freeConn` to free a connection. These are specified as follows.

Method	Pre-condition	Post-condition
`ConnPool(db)`	*true*	$cpool(\mathtt{this}, \mathtt{db})$
`getConn()`	$cpool(\mathtt{this}, db)$	$cpool(\mathtt{this}, db) * conn(ret, db)$
`freeConn(y)`	$cpool(\mathtt{this}, db) * conn(\mathtt{y}, db)$	$cpool(\mathtt{this}, db)$

We give the Inner MJ code that implements these operations in Figure 4.1.

Now let us consider how we could use this class in a proof. We present a simple example where we assume no other classes are used. First we break the proof up using the L-DIntroAbs rule.

$$\dfrac{(A)\quad \Lambda;\Gamma \Vdash \Gamma' \qquad \Lambda;\Gamma \Vdash \left\{\begin{array}{l} cpool(\mathtt{this}, db) \\ *\ conn(\mathtt{y}, db) \end{array}\right\}_{.\mathtt{freeConn}(\mathtt{y})}^{\mathtt{ConnPool}} \{cpool(\mathtt{this}, db)\} \qquad (C)\quad \epsilon;\Gamma \vdash \{P\}\overline{s}\{Q\}}{\epsilon;\epsilon \vdash \{P\}\overline{s}\{Q\}}$$

$$\dfrac{\Lambda;\Gamma \Vdash \Gamma}{\Lambda;\Gamma \Vdash \left\{\begin{array}{l} cpool(\mathtt{this}, db) \\ *\ conn(\mathtt{y}, db) \\ \wedge\ \mathtt{this} : \mathtt{ConnPool} \end{array}\right\} \overline{s}' \{cpool(\mathtt{this}, db)\}}\ (B)$$

where Γ is the method and constructor specifications for the `ConnPool` class; Γ' is the same specifications excluding `freeConn`; Λ is the predicate definitions for *cpool* and *conn*; $\overline{s}$ is the client code; and $\overline{s}'$ is the body of `freeConn`.

We will only present branches (B) and (C). First we present (B): the proof that the `freeConn` implementation satisfies its specification. This illustrates the use of abstract predicates:

$\{cpool(\texttt{this}, db) * conn(\texttt{y}, db) \wedge \texttt{this} : \texttt{ConnPool}\}$
$\{\exists i.\ \texttt{this.l} \mapsto i * \texttt{this.db} \mapsto db * clist(i, db) * conn(\texttt{y}, db)\}$
```
  t=this.l;
```
$\{\texttt{this.l} \mapsto \texttt{t} * \texttt{this.db} \mapsto db * clist(\texttt{t}, db) * conn(\texttt{y}, db)\}$
```
  n=new CList(y,t);
```
$\{\texttt{this.l} \mapsto \texttt{t} * \texttt{this.db} \mapsto db * \texttt{n.hd} \mapsto \texttt{y} * \texttt{n.tl} \mapsto \texttt{t} * clist(\texttt{t}, db) * conn(\texttt{y}, db)\}$
```
  this.l=n
```
$\{\texttt{this.l} \mapsto \texttt{n} * \texttt{this.db} \mapsto db * \texttt{n.hd} \mapsto \texttt{y} * \texttt{n.tl} \mapsto \texttt{t} * clist(\texttt{t}, db) * conn(\texttt{y}, db)\}$
$\{\texttt{this.l} \mapsto \texttt{n} * \texttt{this.db} \mapsto db * clist(\texttt{n}, db)\}$
$\{cpool(\texttt{this}, db)\}$

In this proof the definitions from Λ of both *cpool* and *clist* are used with OPEN and CLOSE to give the following three implications

$$cpool(\texttt{this}, db) \Rightarrow \exists i.\ \texttt{this.l} \mapsto i * \texttt{this.db} \mapsto db * clist(i, db)$$
$$\texttt{n.hd} \mapsto \texttt{y} * \texttt{n.tl} \mapsto \texttt{t} * clist(\texttt{t}, db) * conn(\texttt{y}, db) \Rightarrow clist(\texttt{n}, db)$$
$$\texttt{this.l} \mapsto \texttt{n} * \texttt{this.db} \mapsto db * clist(\texttt{n}, db) \Rightarrow cpool(\texttt{this}, db)$$

These are used with the rule of L-CONSEQUENCE to complete the proof. It is essential that Λ is known in (B) for these implications to hold.

Next we present, and attempt to verify, a fragment of client code using the connection pool (branch (C)). It demonstrates both correct and incorrect usage, which causes the verification to fail. The example calls a function, `useConn`, that uses a connection.

$\{cpool(\texttt{x}, db)\}$
```
  y = x.getConn();
```
$\{cpool(\texttt{x}, db) * conn(\texttt{y}, db)\}$
$\{conn(\texttt{y}, db)\}$
```
  useConn(y);
```
$\{conn(\texttt{y}, db)\}$
$\{cpool(\texttt{x}, db) * conn(\texttt{y}, db)\}$
```
  x.freeConn(y);
```
$\{cpool(\texttt{x}, db)\}$
```
  useConn(y)
```
$\{???\}$

The client gets a connection from the pool, uses it and then returns it. However, after returning it, the client tries to use the connection. This command cannot be validated as the precondition does not contain the *conn* predicate. Even though this predicate is contained in *cpool*, the client is unable to expand the definition because it is out of scope. We do not have Λ in this branch of the proof. This illustrates how abstract predicates capture "ownership transfer", a notion first coined by O'Hearn *et al.* [66]. The connection passes from the client into the connection pool stopping the client from directly accessing it, even though the client has a pointer to the connection.

It is essential to be able to create new instances of a datatype. We conclude this section with a second example client that demonstrates the instantiation of abstract predicates. A connection pool library wants many instances of a connection pool; generally one per database. This can be easily handled by constructing the required number of `ConnPool` instances. Assume we have two different databases, `db1` and `db2`.

$\{true\}$
```
  y = new ConnPool(db1);
```
$\{cpool(\texttt{y}, \texttt{db1})\}$
```
  z = new ConnPool(db2);
```

$\{cpool(\texttt{y}, \texttt{db1}) * cpool(\texttt{z}, \texttt{db2})\}$

This code creates two connection pools. The parameter prevents us returning the connection to the incorrect pool.

```
{cpool(y, db1) * cpool(z, db2)}
  x = z.getConn();
{cpool(y, db1) * cpool(z, db2) * conn(x, db2)}
  y.freeConn(x)
{???}
```

The `freeConn` call can only be validated if `db1` = `db2`.[3]

This example has illustrated that abstract predicates capture the notion of "ownership transfer". We will compare our support for ownership transfer to theirs in §4.4.2.

4.2.2 Ticket machine

Our next example is a simple ticket machine. Consider the following specifications:

Function	Pre-condition	Post-condition
`getTicket()`	$true$	$Ticket(ret)$
`useTicket(x)`	$Ticket(\texttt{x})$	$true$

To call `useTicket` you must have called `getTicket`; each usage consumes a ticket. Trying to use a ticket twice fails:

```
{true}
  x = getTicket();
{Ticket(x)}
  useTicket(x);
{true}
  useTicket(x);
{???}
```

The second call to `useTicket` fails, because the first call removed the $Ticket$.

Any client that is validated against this specification must use the ticket discipline correctly. In fact the module is free to define the ticket in any way, e.g. $Ticket(x) \stackrel{\text{def}}{=} true$. Although duplication of this ticket would be logically valid, $true * true \Leftrightarrow true$, the client does not know this, and hence cannot.

4.2.3 Aside: Permissions interpretation

To better understand the previous example we turn to the permissions interpretation of separation logic. O'Hearn [63] has recently given separation logic an ownership, or permissions, interpretation: $\mapsto$ is the permission to read and write a memory location. (This has been extended by Bornat, Calcagno, O'Hearn and Parkinson [14] to separate read and total permissions; we will detail this in Chapter 5.) In the previous example, the $Ticket$ predicate is the permission to call `useTicket` once. The call consumes this permission. Using the permissions interpretation of separation logic, abstract predicates allow modules to define their own permissions. The concept of ownership transfer can be seen as transferring permission to and from the client.

[3] Given the specification it is always valid to return a connection to a pool if it is to the correct database. A tighter specification could be given to restrict returning to the allocating pool. See the malloc example in §4.4.1.

4.3 Semantics

In the previous sections we have introduced informally the notion of abstract predicates and detailed a couple of examples to demonstrate their usefulness. In this section we formalize them precisely and show that the two abstract predicate rules are sound: L-ABSWEAK and L-ABSELIM.

First we define the semantics of an abstract predicate. We define semantic predicate environments, Δ, as follows:

$$\Delta : \quad \prod \alpha : \mathcal{A}.(\mathsf{Values}^{arity(\alpha)} \to \mathcal{P}(\mathcal{H}))_\perp$$

where $\mathcal{A}$ is the set of predicate names. We use an indexed, or dependent, product to represent a function from predicate name to definition of the correct arity. Later we require the combination of disjoint Δs, so the always $false$ definition is different from an undefined predicate, hence we lift the lattice. The reader might have expected the use of $\mathcal{P}(\mathcal{H} \times \mathcal{S} \times \mathcal{I})$ where $\mathcal{S}$ is the set of all stacks and $\mathcal{I}$ is the set of all ghost stacks. However, this breaks substitution as the predicate can depend on variables that are not syntactically free.

The semantics of a predicate is as follows:

$$VS, H, I \models_\Delta \alpha(\overline{e}) \quad \stackrel{\text{def}}{=} \quad \alpha \in dom(\Delta) \ \wedge \ H \in (\Delta\alpha)[\![\overline{e}]\!]_{VS,I}]$$

We lookup the definition in Δ, evaluate the arguments with respect to the stack, VS, supply the arguments to the definition, and check the heap is contained in the result. The remaining semantics are defined the same as in §3.3 with the predicate environment added in the obvious way.

Lemma 4.3.1. *Semantic predicate environments form a complete lattice.*

Proof. The powerset gives us a complete lattice. Any function from a set to a complete lattice is also a complete lattice. Lifting a complete lattice is a complete lattice, and a product of complete lattices is a complete lattice. □

The complete lattice has the following ordering

$$\Delta \sqsubseteq \Delta' \stackrel{\text{def}}{=} \forall \alpha. \forall \overline{v} : \mathsf{Values}^{arity(\alpha)}. \Delta(\alpha) \neq \perp \Rightarrow \Delta(\alpha)(\overline{v}) \subseteq \Delta'(\alpha)(\overline{v})$$

and the least upper bound of this order is written $\sqcup$ and defined as:

$$\bigsqcup_{i \in I} (\Delta_i) \quad \stackrel{\text{def}}{=} \quad \lambda\alpha.\lambda\overline{v} : \mathsf{Values}^{arity(\alpha)}. \bigcup_{i \in I}^{\perp} (\Delta_i(\alpha)(\overline{v}))$$

where $\bigcup_{i \in I}{}^{\perp} X_i \stackrel{\text{def}}{=} \begin{cases} \perp & \forall i \in I. X_i = \perp \\ \bigcup_{i \in I \wedge X_i \neq \perp} X_i & \text{otherwise} \end{cases}$

Lemma 4.3.2. *Formulae only depend on the predicate names they mention, i.e. if Δ defines all the predicate names in P, and Δ and Δ' are disjoint, then*

$$\forall VS, H, I. \quad VS, H, I \models_\Delta P \Leftrightarrow VS, H, I \models_{\Delta \sqcup \Delta'} P$$

Proof. By induction on the structure of P. □

Definition 4.3.3 (Positive and negative formulae)**.**

$$
\begin{array}{llll}
pos(P \wedge Q) & = pos(P) \wedge pos(Q) & neg(P \wedge Q) & = neg(P) \wedge neg(Q) \\
pos(P \vee Q) & = pos(P) \wedge pos(Q) & neg(P \vee Q) & = neg(P) \wedge neg(Q) \\
pos(P * Q) & = pos(P) \wedge pos(Q) & neg(P * Q) & = neg(P) \wedge neg(Q) \\
pos(e = e) & = true & neg(e = e) & = true \\
pos(e.f \mapsto e) & = true & neg(e.f \mapsto e) & = true \\
pos(P -\!\!* Q) & = neg(P) \wedge pos(Q) & neg(P -\!\!* Q) & = pos(P) \wedge neg(Q) \\
pos(P \Rightarrow Q) & = neg(P) \wedge pos(Q) & neg(P \Rightarrow Q) & = pos(P) \wedge neg(Q) \\
pos(\alpha(\overline{v})) & = true & neg(\alpha(\overline{v})) & = false
\end{array}
$$

Lemma 4.3.4. *Positive formulae are monotonic with respect to semantic predicate environments, i.e.*

$$pos(P) \ \wedge \ \Delta \sqsubseteq \Delta' \ \wedge \ VS,H,I \models_{\Delta} P \ \Rightarrow \ VS,H,I \models_{\Delta'} P$$

Proof. We prove the following stronger result

$$
\begin{array}{lll}
\Delta \sqsubseteq \Delta' & \Rightarrow & (pos(P) \wedge VS,H,I \models_{\Delta} P \Rightarrow VS,H,I \models_{\Delta'} P) \\
& & \qquad \wedge (neg(P) \wedge VS,H,I \models_{\Delta'} P \Rightarrow VS,H,I \models_{\Delta} P)
\end{array}
$$

by induction on the structure of P. □

Now let us consider the construction of a semantic predicate environment from an abstract one, Λ. The abstract predicate environment does not necessarily define every predicate, so constructing a solution requires additional semantic definitions, Δ, to fill the holes. We use the following function to generate a fixed point:

$$step_{(\Delta,\Lambda)}(\Delta') \stackrel{\text{def}}{=} \lambda\alpha \in dom(\Lambda).\lambda\overline{n} \in \mathbb{N}^{arity(\alpha)}.\{H \mid VS,H,I \models_{\Delta' \sqcup \Delta} \Lambda(\alpha)[\overline{n}]\}$$

where Λ contains the definitions we want to solve; and Δ are the predicates not defined in Λ. $step$ is monotonic on predicate environments, because of Lemma 4.3.4 and that all the definitions are positive. Hence by Tarski's theorem and Lemma 4.3.1 we know a least fixed point always exists. We write $[\![\Lambda]\!]_{\Delta}$ for the least fixed point[4] of $step_{\Delta,\Lambda}$.

Consider the set of all solutions of Λ of the form $\Delta \sqcup [\![\Lambda]\!]_{\Delta}$:

$$close(\Lambda) \stackrel{\text{def}}{=} \{\Delta \sqcup [\![\Lambda]\!]_{\Delta} | \ \ dom(\Delta) = \mathcal{A} \setminus dom(\Lambda)\}$$

Next we consider a property of fixed points, and a couple of properties of this function.

Lemma 4.3.5. *Consider two monotonic functions: $f_1 : A \times B \to A$ and $f_2 : A \times B \to B$ where A and B are complete lattices. Consider the simultaneous fixed point (a, b):*

$$fix(x, y).(f_1(x, y), f_2(x, y)) = (a, b) \tag{4.1}$$

and the fixed point with one argument provided:

$$fix(x).(f_2(a', x)) = b'(a') \tag{4.2}$$

The fixed points are equal: $b'(a) = b$

Note: This Lemma is actually a corollary of Bekič's theorem [85, Page 163]. We present the following simplified proof for completeness.

[4] We could generalise the semantics to consider all possible fixed points. We use least fixed points to simplify the semantics.

Proof. We know b is a solution to fixed point equation in (4.2) from (4.1) as $f_1(a,b) = a$. Therefore $b'(a) \subseteq b$. Now prove $(a, b'(a))$ is a pre-fix point of equation in (4.1). This requires

$$(f_1(a,b'(a)), f_2(a,b'(a))) \quad \subseteq \quad (a,b'(a)) \tag{4.3}$$

The second element of the pair follows from (4.2). We know $b'(a) \subseteq b$ and f_1 monotone, so $f_1(a,b'(a)) \subseteq f_1(a,b)$. From (4.1) we know $f_1(a,b) = a$. Hence, proving the inequality, (4.3) for the first element of the pair. Therefore $(a,b'(a))$ is a prefix point of 4.1, but (a,b) is the least prefix point, so $b \subseteq b'(a)$. Hence they are equal. □

Lemma 4.3.6. *Adding new predicate definitions refines the set of possible semantic predicate environments, i.e.*

$$close(\Lambda_1) \supseteq close(\Lambda_1, \Lambda_2)$$

Proof. Consider an arbitrary element of $close(\Lambda_1,\Lambda_2)$: $[\![\Lambda_1,\Lambda_2]\!]_\Delta \sqcup \Delta = \Delta \sqcup \Delta_1 \sqcup \Delta_2$, $dom(\Delta_1) = dom(\Lambda_1)$ and $dom(\Delta_2) = dom(\Lambda_2)$. First show $[\![\Lambda_1]\!]_{\Delta\sqcup\Delta_2} = \Delta_1$. This follows directly from Lemma 4.3.5 by defining $f_1(a,b) \stackrel{\text{def}}{=} step_{\Lambda_1;\Delta\sqcup a}(b)$ and $f_2(a,b) \stackrel{\text{def}}{=} step_{\Lambda_2;\Delta\sqcup b}(a)$. From these definitions we get $fix(x).f_1(x,\Delta_2) = [\![\Lambda_1]\!]_{\Delta\sqcup\Delta_2}$ and $fix(x,y).(f_1(x,y), f_2(x,y)) = (\Delta_1,\Delta_2)$. So by the Lemma 4.3.5 we know $[\![\Lambda_1]\!]_{\Delta\sqcup\Delta_2} = \Delta_1$, therefore $[\![\Lambda_1,\Lambda_2]\!]_\Delta \sqcup \Delta = [\![\Lambda_1]\!]_{\Delta\sqcup\Delta_2} \sqcup \Delta \sqcup \Delta_2 \in close(\Lambda_1)$. Therefore every element is contained in $close(\Lambda_1)$. □

Lemma 4.3.7. *The removal of predicate definitions does not affect predicates that do not use them. Given Λ which is disjoint from Λ' and does not mention predicate names in its domain; we have*

$$\forall \Delta \in close(\Lambda).\exists \Delta' \in close(\Lambda,\Lambda').\ \Delta \upharpoonright dom(\Lambda') = \Delta' \upharpoonright dom(\Lambda')$$

where $f \upharpoonright S$ is $\{a \mapsto b | a \mapsto b \in f \wedge a \notin dom(S)\}$

Proof. We only need consider $\alpha \in dom(\Lambda)$. Consider $\Delta_1 \sqcup \Delta_2 = [\![\Lambda,\Lambda']\!]_\Delta$ where $dom(\Delta_1) = dom(\Lambda)$ and $dom(\Delta_2) = dom(\Lambda')$. Using Lemma 4.3.2 we get $\forall \Delta'.\ [\![\Lambda]\!]_{\Delta\sqcup\Delta'} = [\![\Lambda]\!]_\Delta = [\![\Lambda]\!]_{\Delta\sqcup\Delta_2}$. By the proof of the previous Lemma we know

$$\forall \alpha \in dom(\Lambda). \quad [\![\Lambda]\!]_{\Delta\sqcup\Delta'}(\alpha) = [\![\Lambda]\!]_{\Delta\sqcup\Delta_2}(\alpha) = [\![\Lambda,\Lambda']\!]_\Delta(\alpha)$$

Therefore, for every element in $close(\Lambda)$, there is an element in $close(\Lambda,\Lambda')$ that agrees on all the definitions not in $dom(\Lambda')$. □

We define validity with respect to an abstract predicate environment, written $\Lambda \models P$, as follows:

$$\forall\, VS, H, I, \Delta \in close(\Lambda).\ VS, H, I \models_\Delta P$$

Theorem 4.3.8. OPEN *and* CLOSE *are valid, i.e.*

$$(\alpha(\overline{x}) \stackrel{def}{=} P), \Lambda \models \alpha(\overline{e}) \Rightarrow P[\overline{e}/\overline{x}]$$
$$(\alpha(\overline{x}) \stackrel{def}{=} P), \Lambda \models P[\overline{e}/\overline{x}] \Rightarrow \alpha(\overline{e})$$

Proof. Directly from definitions. □

We are now in a position to define a semantics for our reasoning system. We write $\Lambda;\Gamma \models \{P\}\overline{s}\{Q\}$ to mean that, if every specification in Γ holds for the methods and constructors, and every abstract predicate definition in Λ is true of a predicate environment, then so is $\{P\}\overline{s}\{Q\}$:

$$\forall \Delta \in close(\Lambda).\ \Delta \models \Gamma \ \Rightarrow\ \Delta \models \{P\}\overline{s}\{Q\}$$

The rest of the definitions are modified in the obvious way to pass the abstract predicate environment around.

Given this definition we can show that the two new rules for abstract predicates are sound.

Command	Description	Rule
`x=[e]`	read heap location e	$\left\{\begin{array}{l} e = Z \\ \wedge\ Z \mapsto Y \end{array}\right\}$ `x = [e]` $\left\{\begin{array}{l} Z \mapsto Y \\ \wedge\ \mathtt{x} = Y \end{array}\right\}$
`[e]=`e'	write e' to heap location e	$\{e \mapsto \_\}[e] = e'\{e \mapsto e'\}$
`dispose` e	dispose of heap location e	$\{e \mapsto \_\}$`dispose` $e\{empty\}$
`while(B)`$\overline{s}$	while B holds execute $\overline{s}$	$\dfrac{\{R \wedge B\}\overline{s}\{R\}}{\{R\}\mathtt{while}\,(B)\overline{s}\{R \wedge \neg B\}}$
`newvar x in` $\overline{s}$	new local variable x	$\dfrac{\{P\}\overline{s}\{Q\}}{\{P\}\mathtt{newvar\ x\ in}\ \overline{s}\ \{Q\}}$ where $\mathtt{x} \notin FV(P) \cup FV(Q)$

Figure 4.2: Details required for Malloc and free

Theorem 4.3.9. *Abstract weakening,* L-ABSWEAK*, is sound.*

Proof. Direct consequence of definition of judgements and Lemma 4.3.6. □

Theorem 4.3.10. *Abstract elimination,* L-ABSELIM*, is sound.*

Proof. Follows from Lemmas 4.3.2 and 4.3.7. □

4.4 Discussion

We conclude this chapter by considering, in detail, an example of a memory manager and three related works: the hypothetical frame rule, higher-order separation logic, and typestates.

4.4.1 Malloc and free

In this subsection we present an example of a memory manager that allocates and deallocates variable sized blocks of memory. We present the example for two reasons: it demonstrates that abstract predicates are applicable to non-object-oriented programming; and it allows us to present a better comparison with the hypothetical frame rule. (This example was suggested to us by O'Hearn.)

In this example we use a different model of separation logic, as we require disposal and address arithmetic. Recalling the discussion from §3.3 this requires a move from intuitionistic separation logic to classical separation logic to exclude weakening on the spatial connective, $*$: we do not want memory leaks. This alters the semantics of $\mapsto$ to mean that the heap contains a single cell, rather than at least a single cell. We have an additional assertion $empty$ for the empty heap. The heap is accessed by integers, rather than *oids* and fields, and we use e to range over integer expressions. We present a summary of the programming language in Figure 4.2. This language is similar to those used in most recent separation logic papers [65, 75, 67].

We also use some additional features for handling arrays, described by Reynolds [75]: the iterated separating conjunctions, $\circledast_{x=e_1}^{e_2}.P$; and a system routine `allocate` that allocates variable sized blocks. Intuitively, the iterated separating conjunction, $\circledast_{x=e_1}^{e_2}.P$, is the expansion $P[e_1/x] * \ldots * P[e_2/x]$ where x ranges from e_1 to e_2. If e_2 is less than e_1, it is equivalent to $empty$. More formally its semantics are:

$$VS,H,I \models_\Delta \circledast_{x=e_1}^{e_2}.P \stackrel{\text{def}}{=} (\llbracket e_1 \rrbracket_{VS,I} = n_1 \wedge \llbracket e_2 \rrbracket_{VS,I} = n_2) \Rightarrow ((n_1 \leq n_2 \Rightarrow VS,H,I \models_\Delta P[n_1/x] * \circledast_{x=n_1+1}^{n_2}.P) \wedge (n_1 > n_2 \Rightarrow VS,H,I \models_\Delta empty))$$

Returning to the example, consider the following naïve specifications, which demonstrate the difficulties in reasoning about a memory manager:

Function	Pre-condition	Post-condition
`malloc(n)`	$empty$	$\circledast_{i=0}^{n-1}.(ret+i \mapsto \_)$
`free(x)`	$\circledast_{i=0}^{n-1}.x+i \mapsto \_$	$empty$

`malloc` returns an array `n` cells long starting at ret, and `free` consumes an array n cells long starting at `x`. The problem is with `free`'s specification: it does not specify how much memory is returned as n is a free variable.

The standard specification [50] of `free` only requires it to deallocate blocks provided by `malloc`. It is undefined on all other arguments. Using abstract predicates we *are* able to provide an adequate specification.

Function	Pre-condition	Post-condition
`malloc(n)`	$empty$	$\circledast_{i=0}^{n-1}.(ret+i \mapsto \_) * Block(ret, n)$
`free(x)`	$\circledast_{i=0}^{n-1}.x+i \mapsto \_ * Block(x,n)$	$empty$

The $Block$ predicate is used as a modular certificate, or permission, that `malloc` actually produced the block. The client cannot construct a $Block$ predicate as its definition is not in scope.

Typical implementations of `malloc` and `free` store the block's size in the cell before the allocated block [50]. This can be specified directly by defining the $Block$ predicate as follows.

$$Block(x,n) \stackrel{\text{def}}{=} (x-1) \mapsto n$$

This allows `free` to determine the quantity of memory returned. More complicated specifications can be used which account for padding and other bookkeeping.

We can give a simple implementations of both `malloc` and `free`, which call system routines to construct (`allocate`) and dispose (`dispose`) the blocks.[5]

```
malloc n =(newvar x in x=allocate(n+1);
           [x]=n; return x+1)
free x =(newvar n in n=[x-1];
         while(n≥0) (n=n-1; dispose(x+n))
```

Both of their implementations can be verified; here we present the proof of `malloc`:

$\{empty\}$
`x=allocate(n+1);`
$\{\circledast_{i=0}^{n}.\ (x+i) \mapsto \_\}$
$\{x \mapsto \_ * \circledast_{i=1}^{n}.\ (x+i) \mapsto \_\}$
`[x]=n;`
$\{x \mapsto n * \circledast_{i=1}^{n}.\ (x+i) \mapsto \_\}$
`return x+1`
$\{(ret-1) \mapsto n * \circledast_{i=1}^{n}.\ (ret-1+i) \mapsto \_\}$
$\{(ret-1) \mapsto n * \circledast_{i=0}^{n-1}.\ (ret+i) \mapsto \_\}$
$\{\circledast_{i=0}^{n-1}.\ (ret+i) \mapsto \_ * Block(ret,n)\}$

[5] One could extend the specification to have an additional memory manager predicate that contains a free list, as in the connection pool example.

The final step in this proof abstracts the cell containing the block's length, hence the client cannot directly access it. The following code fragment attempts (but fails) to break this abstraction:

$\{emp\}$
```
x=malloc(30);
```
$\{\circledast_{i=0}^{29}.\ (\mathtt{x}+i) \mapsto \_ * Block(\mathtt{x}, 30)\}$
```
[x-1]=15;
```
$\{???\}$
```
free(x);
```

The client attempts to modify the information about the block's size. This would be a clear failure in modularity as the client is dependent on the implementation of $Block$. Fortunately, we are unable to validate the assignment as the pre-condition does not contain $\mathtt{x}-1 \mapsto \_$. Although the $Block$ contains the cell, the client does not have the definition in scope and hence cannot use it.

This example has demostrated that abstract predicates are applicable even in languages without direct support for abstraction. The C memory manager uses encapsulation, but this abstraction is not protected by the run-time or the language design. However, it can be protected at the logical level using abstract predicates. Abstract predicates provide encapsulation to languages that do not have encapsulation.

4.4.2 Hypothetical frame rule

The first attack on modularity in separation logic was made by O'Hearn, Reynolds and Yang [66]. They added *static* modularity to separation logic. They hide the internal resources of a module from its clients using the *hypothetical frame rule*.

In this section we present a detailed comparison between abstract predicates and the hypothetical frame rule. In particular we compare the different forms of modularity they allow.

O'Hearn, Reynolds and Yang extend separation logic with the following rule,[6] known as hypothetical frame rule, to allow the hiding of information.

$$\frac{\Gamma, \{P_1\}k_1\{Q_1\}[X_1], \ldots, \{P_n\}k_n\{Q\}[X_n] \vdash \{P\}\overline{s}\{Q\}}{\Gamma, \{P_1 * R\}k_1\{Q_1 * R\}[X_1, Y], \ldots, \{P_n * R\}k_n\{Q * R\}[X_n, Y] \vdash \{P * R\}\overline{s}\{Q * R\}}$$

This rule can be used to derive a modular procedure definition rule, which they use to explain the modularity they enable.

$$\frac{\begin{array}{c}\Gamma \vdash \{P_1 * R\}\overline{s}_1\{Q_1 * R\} \\ \vdots \\ \Gamma \vdash \{P_n * R\}\overline{s}_n\{Q_n * R\} \\ \Gamma, \{P_1\}k_1\{Q_1\}[X_1], \ldots, \{P_n\}k_1\{Q_n\}[X_n] \vdash \{P\}\overline{s}\{Q\}\end{array}}{\Gamma \vdash \{P * R\}\mathtt{let}\ \ k_1 = \overline{s}_1, \ldots, k_n = \overline{s}_n\ \ \mathtt{in}\ \ \overline{s}\ \ \mathtt{end}\{Q * R\}}$$

Note: The rule additionally has several side-conditions on free variables, and also on the form of R, which we discuss later.

In this rule, R is the hidden resource that is only available to the procedures, or functions, $k_1, \ldots, k_n$. The parameterless procedures can modify the resource R, but the procedures must always restore R before they return. Returning to the motivating example of the memory manager, R could be a list of unallocated fixed size[7] blocks of memory. `malloc` and `free` can

[6] Side-conditions omitted.

[7] The memory manager verified with the hypothetical frame rule can only allocate fixed size blocks of memory. It cannot be used with variable sized blocks as described earlier.

temporarily break this invariant but it must always hold when they return. The client code, $\overline{s}$, is verified without R, and hence is unable to *directly* modify the free list. Returning to the memory manager, a client should only modify the free list using `malloc` and `free`, it should not have direct access to this list. This rule describes the partitioning between the client's and the module's state.

This partitioning of resources allows "ownership transfer": state can safely be transferred between the module and the client without fear of dereferencing dangling pointers. This allows reasoning about examples such as a simple memory manager, which allocates fixed size blocks of memory, and a queue.

Though this is a significant advance, their work is severely limited as it only models static modularity. Their modules are based on Parnas' work on information hiding [68], which deals with single instances of the hidden data structure: there is only one R and it is statically scoped. It cannot be used for many common forms of abstraction, including ADTs and classes, where we require multiple dynamically scoped instances of the hidden resource.

Abstract predicates can represent dynamically scoped abstractions. Their use captures "ownership transfer." Our connection pool example captures all the key properties of O'Hearn *et al.*'s idealization of a memory manager. In addition in §4.4.1 we showed that abstract predicates could be used to represent modular permissions, or certificates, and hence verify a memory manager for variable sized blocks. O'Hearn, Reynolds and Yang's [66] idealization of a memory manager does not support variable sized blocks. Their specifications cannot be extended to cover this without exposing the representation of the block. Additionally, it is impossible for them to enforce that `malloc` must provide the blocks that `free` deallocates without extending the logic.

There are several side-conditions to the modular procedure definition rule:

1. $\overline{s}$ does not modify variables in R, *except through using* $k_1, \ldots, k_n$;

2. Y is disjoint from P, Q, $\overline{s}$, and the context $\Gamma, \{P_1\}k_1\{Q_1\}[X_1], \ldots, \{P_n\}k_1\{Q_n\}[X_n]$;

3. $\overline{s}_i$ only modifies variables in X_i and Y; and

4. R is precise.

The first three clauses restrict the use of variables. The variables $X_1, \ldots, X_n$ can appear in both the procedure specifications and the invariant, but cannot be modified by the client. These variables are required to specify the protocol, or behaviour, of the procedures. In addition global "hidden" variables Y are also used to represent the invariant's state. These variables all require careful restriction on how they can be modified.

Abstract predicates do not suffer these complications because they express behaviour using arguments to the predicates: these arguments can be seen by anyone, but can only be modified if the underlying definition is known. There is no need for a special modifies clause.

Clause 4 restricts the forms of invariants the modular procedure definition rule can use. Without these restrictions it is unsound. Consider the following counterexample, in classical separation logic, due to Reynolds. 1 represents a one element heap, 0 represents a zero element

heap.

$$\dfrac{\dfrac{\vdash \{\mathsf{true}\}\texttt{skip}\{\mathsf{true}\}}{\vdash \{(0 \vee 1) * \mathsf{true}\}\texttt{skip}\{0 * \mathsf{true}\}} \quad \dfrac{\dfrac{\dfrac{\Gamma \vdash \{0 \vee 1\}k\{0\}}{\Gamma \vdash \{1\}k\{0\}} \quad \dfrac{\dfrac{\Gamma \vdash \{0 \vee 1\}k\{0\}}{\Gamma \vdash \{0\}k\{0\}}}{\Gamma \vdash \{0 * 1\}k\{0 * 1\}}}{\Gamma \vdash \{1 \wedge 1\}k\{1 \wedge 0\}}}{\Gamma \vdash \{1\}k\{\mathsf{false}\}}}{\dfrac{\vdash \{1 * \mathsf{true}\}\texttt{let } k = \texttt{skip in } k \texttt{ end}\{\mathsf{true} * \mathsf{false}\}}{\vdash \{1 * \mathsf{true}\}\texttt{let } k = \texttt{skip in } k \texttt{ end}\{\mathsf{false}\}}}$$

where $\Gamma = \{0 \vee 1\}k\{0\}$.

In classical separation logic precise sizes of heaps can be expressed so $1 \wedge 0$ is false. Clearly the code above will terminate but the proof above shows it will not terminate. The logic is unsound!

The problem arises in the rule of conjunction:

$$\frac{\Gamma \vdash \{P\}\overline{s}\{Q\} \quad \Gamma \vdash \{P_2\}\overline{s}\{Q_2\}}{\Gamma \vdash \{P \wedge Q\}\overline{s}\{P_2 \wedge Q_2\}}$$

If we are using the information hiding outlined above, then we really have a hidden $* R$ attached to each pre- and post-condition. However, we cannot directly rewrite the proof rule with these. We must insert a use of the rule of consequence.

$$\dfrac{\dfrac{\Gamma \vdash \{P * R\}k\{Q * R\} \quad \Gamma \vdash \{P_2 * R\}k\{Q_2 * R\}}{\Gamma \vdash \{P * R \wedge P_2 * R\}k\{Q * R \wedge Q_2 * R\}}}{\Gamma \vdash \{(P \wedge P_2) * R\}k\{(Q \wedge Q_2) * R\}} \tag{4.4}$$

This leaves us with two implications to prove:

$$\begin{aligned}(P \wedge P_2) * R &\Rightarrow P * R \wedge P_2 * R \\ Q * R \wedge Q_2 * R &\Rightarrow (Q \wedge Q_2) * R\end{aligned}$$

However, in classical separation logic, the second implication only holds for *precise* assertions [66].

> A predicate P is *precise* if and only if for all states (VS, H, I) there is at most one subheap H_p of H for which $VS, H_p, I \models P$ holds.

For soundness, O'Hearn, Reynolds and Yang restrict their rule to use only these precise assertions.[8]

The soundness of abstract predicates is less subtle and has no need to restrict the forms of predicates; except to ensure recursive definitions exist.

To summarize, the hypothetical frame rule can only deal with static modularity: modularity with a single hidden resource. It cannot deal with dynamic modularity: modularity where the hidden resource can be dynamically allocated. Hence it cannot be used to reason about abstract datatypes or objects. When reasoning about static modularity the proofs are more compact and elegant with the hypothetical frame rule. However, all these proofs can be encoded into abstract predicates.We do not believe the converse is true: the static nature of the hypothetical frame rule prevents simple reasoning about abstract datatypes.

[8] In intuitionistic separation logic there is a different restriction called *supported* [66].

```
class Footballer {
  {true}
   Footballer() {...}
  {Healthy(this)}

  {Healthy(this)}
   play() {...}
  {Tired(this)}

  {Tired(this)}
   pub() {...}
  {Aching(this)}

  {Tired(this)}
   warmDown() {...}
  {Healthy(this)}

  {Aching(this)}
   longRest() {...}
  {Healthy(this)}
}
```

Figure 4.3: Simple protocol example

4.4.3 Higher-order separation logic

In this chapter we have built a logic for reasoning about encapsulation and abstract types. It is well-known that abstract types are related via the Curry-Howard correspondence to existential types [58]. Abstract predicates appear to be analogous, but at the level of the propositions themselves. In recent unpublished work, Birkedal and Torp-Smith [10] show how to use existential quantifiers to model abstract predicates. They use the higher-order separation logic of Biering, Birkedal and Torp-Smith [8] and present the connection pool example given earlier. In addition to modelling abstract predicates with existentially quantified predicates, they also show how to use universal quantifiers to model parametric datatypes.

Similarly Reddy [73] has extended specification logic [78] to allow existential quantification over predicates. This quantification allows him to represent the encapsulation of state, and hence objects and abstract datatypes.

4.4.4 Typestates

Finally we discuss typestates as they allow similar styles of proofs to those presented in this chapter.

Fähndrich and DeLine [36] have developed a simple method for protocol checking in class-based languages. This work is based on Typestates of Strom and Yemini [81]. Typestates were initially developed to help compiler writers in formalizing checking for programming errors such as accessing uninitialized variables. Each variable has a Typestate associated to it, e.g. $x : (int \times uninitialized), y : (int \times initialized)$. The Typestate can be altered by operations, e.g. assigning an initialized variable to an uninitialized variable alters the Typestate to initialized.

Fähndrich and DeLine extend this to allow the programmer to express protocols for a class. We illustrate the protocol ideas with a simple example of a footballer and how certain actions affect the footballer's state. We give the specifications in Figure 4.3 and present a state diagram in Figure 4.4. Note the dotted transition in the state diagram is not represented in the specification. Unfortunately this specification might leave the average footballer unhappy as he would not be able to go to the pub without playing football and ending up $Aching$. We could change the specification to the `pub()` method to be:[9]

```
{Tired(this) ∨ Healthy(this)}
 pub() {...}
{Aching(this)}
```

[9] Not shown in Figure 4.4.

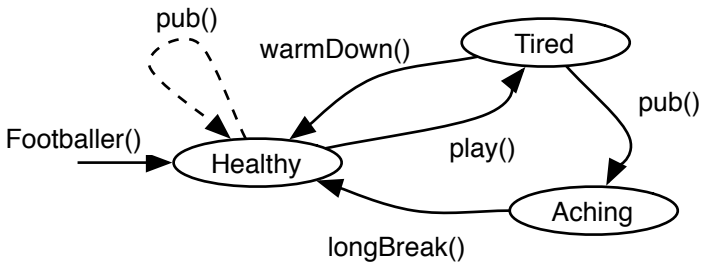

Figure 4.4: State diagram for protocol

This is still unsatisfactory as we really want the footballer to be able to go to the pub if healthy and still be healthy afterwards. Fähndrich and DeLine's logic cannot express this, because they cannot express dependencies between input and output states. Our logic is able to express this through the use of ghost variables

$\{(X = 1 \wedge \mathit{Tired}(\texttt{this})) \vee (X = 2 \wedge \mathit{Healthy}(\texttt{this}))\}$
`pub() {...}`
$\{(X = 1 \wedge \mathit{Aching}(\texttt{this})) \vee (X = 2 \wedge \mathit{Healthy}(\texttt{this}))\}$

The logic developed in this thesis is more expressive than Fähndrich and DeLine, but their aims are different: their logic is designed to automatically check protocols, rather than to prove correctness of programs. Their logic also deals with inheritance in a similar way to that developed in Chapter 6, but they give more structure to the Typestates than we give to abstract predicate families, which allows them to more easily inherit method bodies, but restricts the potential changes a subclass can make. This is discussed in more detail in § 6.5.

5

Read sharing

In the previous chapter we presented abstract predicates and showed their utility in reasoning about encapsulation. However, abstract predicates necessarily prevent sharing between data-structures.[1] If a predicate represents some state, then no other disjoint predicate can represent the same piece of state. This is required to prevent updates to one data-structure affecting another's representation (the raison d'être of separation logic!).

However, it is perfectly reasonable to have several data-structures with *read* access to the same state, for example list iterators. A list iterator requires access to the internal structure of a list: several iterators can validly traverse a single list as long as they do not modify it. To reason about this we need shared read access.

In this chapter we will extend separation logic to allow shared read access, and explain this extension using the permissions interpretation of separation logic, given in §4.2.3. In the previous chapter we saw how abstract predicates could be used to generate modular permissions: the permission to use a method or function. The permissions interpretation liberates us from seeing the field predicate, $e.f \mapsto e'$, as simply describing a part of the heap, but instead as the permission to read and write to that part of the heap. With this interpretation in mind it is reasonable to consider other predicates that give us weaker permissions, in particular, read-only permissions.

In this chapter, we introduce a "read-only points-to" predicate, $\rightarrow$. We must ensure that a write permission and a separate read permission cannot exist to the same cell, otherwise our model would be unsound. The classic concurrent example of multiple readers and a single writer [25] keeps a count of the number of readers to the resource. We will mirror this in the logic.

One problem Bornat *et al.* [14] identified with counting read-only permissions is that $*$ can no longer be used to express disjointness. In particular, the obvious definition of a read-only tree admits directed acyclic graphs. We address this problem by considering a new model of separation logic which uses sets of named permissions.

The rest of the chapter is organised as follows. In §5.1 we present a sound logic for read-only assertions, and in §5.2 demonstrate its use. In §5.3 we show that $*$ no longer expresses disjointness, and present an extension to the logic that solves this problem. In §5.4, we present an abstract model that embodies both permissions idioms, and then in §5.5 we present the instantiations of these models and show the named model properly contains the counting model. We

[1] This is not completely true because one can use standard classical conjunction, but this removes much of the benefit of using separation logic.

conclude by demonstrating some other idioms of permission accounting that can be expressed in the named permissions model.

Collaboration This chapter is based on joint work with Richard Bornat, Cristiano Calcagno, and Peter O'Hearn. The author discovered the counting model independently of Calcagno, O'Hearn and Bornat. This lead to a collaborative publication about read-only permissions in a non-object-oriented setting [14]. This chapter extends that published work by demonstrating how read-only permissions can be used with datatypes, and solves the open problem of combining disjointness and shared access. The presentation of the abstract semantics, in §5.4, is due to Calcagno.

5.1 Counting

In this section we present a simple extension that allows the counting of read-only permissions. The counting logic mirrors the programming paradigm of reference counting by keeping a count of the number of read permissions to a field.

This model uses two types of permissions: source permissions, $e.f \overset{n}{\longmapsto} e'$, and read-only permissions, $e.f \to e'$. We define a source permission $e.f \overset{n}{\longmapsto} e'$ as a permission that has issued n read permissions to $e.f$. This definition gives rise to the following axiom:

$$e.f \overset{n}{\longmapsto} e' \quad \Leftrightarrow \quad e.f \to e' * e.f \overset{n+1}{\longmapsto} e'$$

If we have a source permission that has issued n read permissions, it is equivalent to a source permission that has issued $n+1$ read permissions and a read permission. This axiom allows the source permission to dispense and collect read permissions to the cell. We have a distinguished source permission written $e.f \overset{0}{\longmapsto} e'$, which represents a total, or write, permission. We use the shorthand $e.f \mapsto e'$ for such a total permission.

There is an additional axiom that is required for some proofs:

$$e.f \overset{0}{\longmapsto} e' * e.f \to e'' \quad \Longleftrightarrow \quad \mathit{false}$$

This prevents us having a total permission and a read permission. It also prevents us having two source permission to the same field.

We generalise the field read axiom to only require a read permission to the field. The other axioms are unchanged as they require total permissions.[2]

L-FREAD $\quad \Gamma \vdash \{x = X \wedge X.f \to Y\} z = x.f; \{X.f \to Y \wedge z = Y\}$

Note: As we are working in an intuitionistic setting we can have permissions leaks: state can no longer be modified because part of the permission has been lost. This may at first seem unsettling but it allows us to define an immutable location. $\exists n.\ \ e.f \overset{n}{\longmapsto} e'$ is a location that can never be modified as we do not know how many permissions have been removed.

5.2 Example: List iterators

In this section we present our motivating example of a list class with iterators. An iterator has a constructor and two methods, `hasNext` and `next`. The `hasNext` method returns `true` if there

[2] One might say the other axioms have all the occurrences of $e.f \mapsto e'$ replaced with $e.f \overset{0}{\longmapsto} e'$. Our shorthand makes this syntactic substitution.

```
class Iter {
  List n;
  Iter(List x) {
    this.n = x;
  }
  boolean hasNext() {
    return this.n!=null;
  }
  Object next() {
    List hd = this.n.hd;
    List temp = this.n.tl;
    this.n = temp;
    return hd;
  }
}

class List {
  Object Hd;
  List tl;
  ...
  Iter iterator() {
    return new Iter(this);
  }
  ...
}
```

Figure 5.1: List iterator source

Method	Pre-condition	Post-condition
`x.iterator()`	$list(\mathtt{x}, n)$	$list(\mathtt{x}, n+1) * iter(ret, \mathtt{x})$
`x.hasNext()`	$iter(\mathtt{x}, y)$	$(ret = true \wedge iter_b(\mathtt{x}, y))$ $\vee\ (ret = false \wedge iter(\mathtt{x}, y))$
`x.next()`	$iter_b(\mathtt{x}, y)$	$iter(\mathtt{x}, y)$
`Iter(x)`	$list_R(\mathtt{x})$	$iter(\mathtt{this}, \mathtt{x})$

Table 5.1: Specifications for iterator methods

are more elements to be examined, and `next` returns the next element. We give the source code in Figure 5.1 and its specification in Table 5.1. The implementation uses `boolean`s to aid clarity; this can easily be encoded into Inner MJ. The specifications use two predicates to represent an iterator: $iter$ is just the datastructure for an iterator, and $iter_b$ is the same with a constraint that it has a current element: the current node is not `null`. The predicates are defined as:

$$iter(x,y) \stackrel{\text{def}}{=} \exists z.\ x.\mathtt{n} \mapsto z * list_R(z) * (list_R(z) \mathrel{-\!\!*} list_R(y))$$
$$iter_b(x,y) \stackrel{\text{def}}{=} \exists z.\ x.\mathtt{n} \mapsto z * list_R(z) * (list_R(z) \mathrel{-\!\!*} list_R(y)) \wedge z \neq \mathtt{null}$$

The second predicate is used to prevent clients trying to access the next element when it does not exist.

Note: The use of two predicates represents the calling protocol of the class, and mirrors the work on Typestates [36] mentioned in §4.4.4.

Now we define the predicates for representing a list:

$$list(i,n) \stackrel{\text{def}}{=} (\exists k, l.\ i.\mathtt{hd} \stackrel{n}{\mapsto} k * i.\mathtt{tl} \stackrel{n}{\mapsto} l * list(l,n)) \vee i = \mathtt{null}$$
$$list_R(i) \stackrel{\text{def}}{=} (\exists k, l.\ i.\mathtt{hd} \to k * i.\mathtt{tl} \to l * list_R(l)) \vee i = \mathtt{null}$$

The n parameter is used to count the number of passive readers of the list. $list_R$ is the read-only list predicates that allows multiple iterators over the same list.[3]

We have a lemma about these list definitions: it allows read-only list segments to be formed from a list source.

[3] In what follows we will adopt the convention that a subscript R on predicates represents read-only state.

Lemma 5.2.1. $list(i,n) \quad \Leftrightarrow \quad list_R(i) * list(i,n+1)$

Proof. By induction on the length of the list segment. □

Next we present proofs that the constructor, `Iter()`, and the methods, `iterator()` and `hasNext()` meet their specifications. First we consider `Iter()`:

$\{List_R(\texttt{x}) * \texttt{this.n} \mapsto \texttt{null}\}$
`this.n = x;`
$\{List_R(\texttt{x}) * \texttt{this.n} \mapsto \texttt{x}\}$
$\{List_R(\texttt{x}) * (List_R(\texttt{x}) \mathrel{-\!\!*} List_R(\texttt{x})) * \texttt{this.n} \mapsto \texttt{x}\}$
$\{iter(\texttt{this}, \texttt{x})\}$

This proof follows directly from the definitions of $\mathrel{-\!\!*}$ and $iter$. We can prove the `iterator` method with Lemma 5.2.1 as follows.

$\{list(\texttt{this}, n)\}$
$\{list(\texttt{this}, n+1) * list_R(\texttt{this})\}$
`return new Iter(this);`
$\{list(\texttt{this}, n+1) * iter(ret, \texttt{this})\}$

Finally we give the proof for `hasNext`:

$\{iter(\texttt{this}, y)\}$
$\{\exists z.\ \texttt{this.n} \mapsto z * list_R(z) * (list_R(z) \mathrel{-\!\!*} list(y))\}$
`Object t = this.n;`
$\{\texttt{this.n} \mapsto \texttt{t} * list_R(\texttt{t}) * (list_R(\texttt{t}) \mathrel{-\!\!*} list_R(y))\}$
`return t != null;`
$\{ret = (\texttt{t} \neq \texttt{null}) \wedge \texttt{this.n} \mapsto \texttt{t} * list_R(\texttt{t}) * (list_R(\texttt{t}) \mathrel{-\!\!*} list_R(y))\}$
$\left\{\begin{array}{l}((ret = true \wedge \texttt{t} \neq \texttt{null}) \vee (ret = false \wedge \texttt{t} = \texttt{null})) \\ \quad * \texttt{this.n} \mapsto \texttt{t} * list_R(\texttt{t}) * (list_R(\texttt{t}) \mathrel{-\!\!*} list_R(y))\end{array}\right\}$
$\{(ret = true \wedge iter_b(\texttt{this}, y)) \vee (ret = false \wedge iter(\texttt{this}, y))\}$

Now we present an example of client code incorrectly using an iterator:

$\{list(\texttt{l}, 0)\}$
`Iter x;`
$\{list(\texttt{l}, 0)\}$
`x = l.iterator();`
$\{list(\texttt{l}, 1) * iter(\texttt{x}, \texttt{l})\}$
$\{iter(\texttt{x}, \texttt{l})\}$
`if(x.hasNext())`
$\{iter_b(\texttt{x}, \texttt{l})\}$
`o = x.next();`
$\{iter(\texttt{x}, \texttt{l})\}$
`else`
$\{iter(\texttt{x}, \texttt{l})\}$
`x.next();`
$\{???\}$
$\{???\}$
$\{list(\texttt{l}, 1) * ???\}$

The client constructs an iterator, checks if it has a next element, but then attempts to move to the next element in both branches of the `if`. However, this cannot be validated as the $iter_b$ predicate is not in the pre-condition. The verification has prevented a null pointer exception without any additional runtime checks.

Now let us consider how we can modify a list. With Java's linked list implementation, changing a list and then using an iterator created before the change throws an exception. Here is an excerpt from the API:[4]

> The iterators returned by [the list's] `iterator` and `listIterator` methods are fail-fast: if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own remove or add methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

We want the logic to prevent this exception statically. We could disallow updates if the list has iterators, e.g.

Method	Pre-condition	Post-condition
`List.reverse()`	$list(\texttt{this}, 0)$	$list(ret, 0)$

The specification stops us calling `reverse` if the list has any iterators. This is too restrictive: using an iterator after a modification is the problem, not their existence. Consider the following specification:

Method	Pre-condition	Post-condition
`List.reverse()`	$list(\texttt{this}, n) \ * \ \circledast_{i=1}^{n}(\exists z.(iter(z, \texttt{this}) \vee iter_b(z, \texttt{this}))$	$list(ret, 0)$

See §4.4.1 for an explanation of the iterated separating conjunction, $\circledast_{i=1}^{n}$.

Although the pre-condition implies $list(\texttt{this}, 0)$ the client must treat the predicates abstractly, so cannot know this.

This allows us to have iterators to the list, but at the point we call `reverse` all n iterators will be absorbed back into the list and hence cannot be used. Consider the following usage example

```
{list(x,0)}
y = x.iterator();
{list(x,1) * iter(y,x)}
z = y.hasNext();
{list(x,1) * (z = true ∧ iter_b(y,x)) ∨ (iter(y,x) ∧ z = false)}
if( z )
   {list(x,1) * iter_b(y,x)}
   {list(x,1) * ⊛_{i=1}^{1}(iter_b(y,x))}
    x = x.reverse();
   {list(x,0)}
    w = y.next(); (1)
   {???}
else
    ;
```

When we reach (1) we will not be able to validate the function call as the `reverse` will have swallowed the iterator predicate. Pleasingly the static verification fails at the same point the actual libraries would throw an exception.

Note: We would also prevent a call to `hasNext` if the iterator has been invalidated, which does not throw an exception in the actual Java 1.4.1 implementation (but arguably should).

[4] `http://java.sun.com/j2se/1.4.2/docs/api/java/util/LinkedList.html`

5.3 Disjointness

So far we have seen a logic for read-only permissions and used it to reason about list iterators. Bornat *et al*. [14] have noted that this logic, and the other logics in [14], are not capable of expressing disjointness: a formula cannot *simply*[5] enforce that a read-only tree is not a direct acyclic graph (DAG). Consider the following formula:

$$tree_r(i) \quad \stackrel{\text{def}}{=} \quad i = \texttt{null} \vee (i.\texttt{l} \rightarrow j \ast tree_r(j) \ast i.\texttt{r} \rightarrow k \ast tree_r(k))$$

From a single node we can generate several read only references to that node:

$$k.\texttt{l} \mapsto \texttt{null} \ast k.\texttt{r} \mapsto \texttt{null} \quad \Rightarrow \quad k.\texttt{l} \stackrel{2}{\mapsto} \texttt{null} \ast k.\texttt{r} \stackrel{2}{\mapsto} \texttt{null} \ast tree_r(k) \ast tree_r(k)$$

Clearly, if we have another node where both branches point to k, then we can construct a "tree" in the following way:

$$i.\texttt{l} \mapsto k \ast i.\texttt{r} \mapsto k \ast tree_r(k) \ast tree_r(k) \quad \Rightarrow \quad tree_r(i)$$

This is not a *tree*!

This problem arises as $\ast$ no longer represents disjoint heaps, or even disjoint permissions: we can have the same permission twice, i.e. two read permissions to a single cell. Any two read permissions on a field look identical and we cannot distinguish them.

To distinguish permissions we need to give them names. Consider a countably infinite set of permissions $\mathbb{P}$. The key idea is that we can use subsets of $\mathbb{P}$. Hence we can count by considering the size of sets, and disjointness by considering the names of the elements. Later we will see how to allow the division of sets. We annotate the points-to relation with non-empty subsets of $\mathbb{P}$: $e.f \stackrel{p}{\mapsto} e'$. Any non-empty subset of $\mathbb{P}$ is a read permission and the whole set $\mathbb{P}$ is the total permission. Again we define a shorthand $e.f \mapsto e'$ for $e.f \stackrel{\mathbb{P}}{\mapsto} e'$. Additionally, to use this logic we require some operations on sets: disjoint union, set difference, set cardinality and set equality; and also an existential over subsets of $\mathbb{P}$. The subsets, p, are semanticly treated like ghost variables, and can be eliminated using L-VARELIM. We could redefine the list predicates given earlier, on Page 81, as:

$$\begin{array}{rcl} list(i,n) & \stackrel{\text{def}}{=} & (\exists k, l, p.\ i.\texttt{hd} \stackrel{\mathbb{P}\setminus p}{\longmapsto} k \ast i.\texttt{tl} \stackrel{\mathbb{P}\setminus p}{\longmapsto} l \ast list(l,n) \ast |p| = n) \vee i = \texttt{null} \\ list_R(i) & \stackrel{\text{def}}{=} & (\exists k, l, p.\ i.\texttt{hd} \stackrel{p}{\mapsto} k \ast i.\texttt{tl} \stackrel{p}{\mapsto} l \ast list_R(l) \ast |p| = 1) \vee i = \texttt{null} \end{array}$$

We present a general encoding in §5.5.1 of the counting model into this model.

There are two axioms for permissions in the logic:

DISJOINT	$e.f \stackrel{p}{\mapsto} e' \ast e.f \stackrel{p}{\mapsto} e' \Leftrightarrow \mathit{false}$
SPLIT	$(p_1 \uplus p_2 = p) \quad \Leftrightarrow \quad (e.f \stackrel{p_1}{\mapsto} e' \ast e.f \stackrel{p_2}{\mapsto} e' \Leftrightarrow e.f \stackrel{p}{\mapsto} e')$ where p, p_1 and p_2 are all non-empty

The first, DISJOINT, means we cannot have the same permission set twice and the second, SPLIT, allows us to split and combine disjoint permission sets.

[5] Elaborate mechanisms where trees carry sets of locations could be used, but this solution could hardly be called *simple*.

Note: One might have expected DISJOINT to be defined as $e.f \overset{p}{\longmapsto} e' \ * \ e.f \overset{p'}{\longmapsto} e' \wedge p \cap p' \neq \emptyset \Leftrightarrow false$. This is derivable as

$$\begin{array}{lll} & p \cap p' \neq \emptyset \wedge \overset{p \cap p'}{\longmapsto} * \overset{p \cap p'}{\longmapsto} & \Leftrightarrow \ false \\ \Rightarrow & p \cap p' \neq \emptyset \wedge \overset{p \backslash p'}{\longmapsto} * \overset{p' \backslash p}{\longmapsto} * \overset{p \cap p'}{\longmapsto} * \overset{p \cap p'}{\longmapsto} & \Leftrightarrow \ false * \overset{p \backslash p'}{\longmapsto} * \overset{p' \backslash p}{\longmapsto} \\ \Rightarrow & p \cap p' \neq \emptyset \wedge \overset{p}{\longmapsto} * \overset{p'}{\longmapsto} & \Leftrightarrow \ false \end{array}$$

Again the only programming axiom that needs changing is the field read axiom:

L-FREAD $\quad \Gamma \vdash \{x = X \wedge X.f \overset{p}{\longmapsto} Y \wedge |p| \geq 1\} y = x.f \{X.f \overset{p}{\longmapsto} Y \wedge Y = y\}$

Earlier we showed that the counting model cannot *simply* express read-only trees without admitting DAGs. However, in this model we can simply state that each node in the tree must have the *same* permission set. Consider the formula:

$$tree(i,p) \quad \overset{\text{def}}{=} \quad i \ = \ null \ \vee \ (\exists j,k. \ \ i.\mathtt{l} \overset{p}{\longmapsto} j \ * \ i.\mathtt{r} \overset{p}{\longmapsto} k \ * \ tree(j,p) \ * \ tree(k,p))$$

This definition cannot admit sharing. We can prove this by induction on the definition of $tree$. By induction we can see that $tree(x,p)$ can only contain locations with permission set p. Hence $tree(j,p) \ * \ tree(k,p)$ implies the locations representing $tree(j,p)$ and $tree(k,p)$ are disjoint. Hence, we cannot have sharing: only a tree.

5.4 Abstract semantics

In this section, we provide a semantic model of read-only permissions in separation logic. We use the abstract model of permissions due to Calcagno [14]. We begin by giving this model in terms of a partial commutative semi-group of permissions: $(\mathsf{M}, *)$. We write m for an element of M and redefine heaps, from page 45, as

$$H : \mathcal{H} \overset{\text{def}}{=} (\mathsf{Locations} \times \mathsf{Fields} \rightharpoonup \mathsf{M} \times \mathsf{Values}) \times (\mathsf{Locations} \rightharpoonup \mathsf{Classes})$$

That is each field is represented by a value and an element from the permissions semi-group. We define composition on field values as

$$(m_1, v_1) \ * \ (m_2, v_2) \overset{\text{def}}{=} \begin{cases} (m_1 \ * \ m_2, v_1) & v_1 = v_2, \text{ and } m_1 \ * \ m_2 \text{ is defined.} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

and then extend this to value heaps as

- $H_{v_1} * H_{v_2}$ is defined iff $H_{v_1}(l,f) \ * \ H_{v_2}(l,f)$ is defined for all $(l,f) \in (dom(H_{v_1}) \cap dom(H_{v_2}))$
- $(H_{v_1} \ * \ H_{v_2})(l,f) \overset{\text{def}}{=} \begin{cases} H_{v_1}(l,f) & H_{v_2}(l,f) \text{ is } \textit{undefined} \\ H_{v_2}(l,f) & H_{v_1}(l,f) \text{ is } \textit{undefined} \\ H_{v_1}(l,f) \ * \ H_{v_2}(l,f) & \text{otherwise} \end{cases}$

We write $H_1 \ * \ H_2$ for composition of heaps, which is defined as:

$$(H_{v_1}, H_{t_1}) \ * \ (H_{v_2}, H_{t_2}) \quad \overset{\text{def}}{=} \quad \begin{cases} ((H_{v_1} \ * \ H_{v_2}), H_{t_1}) & H_{v_1} \ * \ H_{v_2} \wedge H_{t_1} = H_{t_2} \\ \text{undefined} & \text{otherwise} \end{cases}$$

E-FIELDACCESS $(H,\, VS, o.f, FS) \rightarrow (H,\, VS, v, FS)$
where $(o,f) \in dom(H)$ and $H(o,f) = m, v$ and $m \in \mathsf{M}$.

E-FIELDWRITE $(H,\, VS, o.f = v;, FS) \rightarrow (H',\, VS, ;, FS)$
where $(o,f) \in dom(H)$ and $H' = H[(o,f) \mapsto m_w, v]$ and $H(o,f) = m_w, v'$

E-NEW $(H,\, VS, \texttt{new}\ C(\overline{v}), FS) \rightarrow (H', (BS \circ []) \circ VS, \overline{s}, (\texttt{return}\ o;) \circ FS)$
where $cnbody(C) = (\overline{x}, \overline{s})$, $\Delta_c(C) = \overline{C}$, $H' = H[o \mapsto C][(o, \overline{f}) \mapsto (m_w, \texttt{null})]$
$o \notin dom(H)$, and $BS = \{\texttt{this} \mapsto (o, C), \overline{x} \mapsto (\overline{v}, \overline{C})\}$

Figure 5.2: Alterations to operational semantics for read-only permissions

We define the semantics of the points-to relation as

$$VS, H, I \models e.f \xmapsto{m} e' \quad \stackrel{\text{def}}{=} \quad H = \{([\![e]\!]_{VS,I}, f) \mapsto (m, [\![e']\!]_{VS,I})\} \wedge m \in \mathsf{M}$$

and define the logical $*$ using the new heap composition operator, $*$.

We define a distinguished element of M, m_w that corresponds to a write permission such that

$$\forall m \in \mathsf{M}.\ \ m * m_w \text{ is undefined} \tag{5.1}$$

The semantics of the language must be altered to accomodate the permissions model. Only the rules for manipulating the heap require any changes: the new definitions of E-FIELDACESS, E-FIELDWRITE and E-NEW are given in Figure 5.2.

The alterations to the semantics correspond to run-time checks that the relevant permissions are available. These new semantics with the constraint (5.1) allow us to show the heap locality, Lemma 3.7.3, and the safety monotonicity, Lemma 3.7.2, properties hold. Hence the frame rule is still sound.

Theorem 5.4.1. *Permissions model is sound.*

Proof. The field read rule is sound with these semantics, and so is the frame rule. The rest of the soundness proof remains unchanged. □

5.5 Concrete models

Now we consider instantiating the abstract model for the two logics given earlier.

Counting model We can define the permissions semi-group for the counting model as: $(\mathbb{Z}, *_1)$ where

$$z_1 \ *_1 \ z_2 = \begin{cases} \text{undefined} & z_1 \geq 0 \ \ \wedge \ \ z_2 \geq 0 \\ \text{undefined} & (z_1 \geq 0 \ \ \vee z_2 \geq 0) \wedge z_1 + z_2 < 0 \\ z_1 + z_2 & \text{otherwise} \end{cases}$$

Note: We equate $\xmapsto{-1}$ with the read permission $\rightarrow$.

Theorem 5.5.1. *The counting axioms are sound.*

$$e.f \xmapsto{n} e' \iff e.f \xmapsto{-1} e' * e.f \xmapsto{n+1} e'$$
$$e.f \xmapsto{0} e' * e.f \xmapsto{-1} e' \iff \mathit{false}$$

Proof. Directly from the definition of the semi-group. □

Named model We can define the permissions semi-group for the named permissions model as: $(\mathcal{P}_{\text{non-empty}}(\mathbb{P}), \uplus)$, where $\mathcal{P}_{\text{non-empty}}(\mathbb{P})$ is the set of non-empty subsets of $\mathbb{P}$.

Additionally, we must add a few simple set operations to the syntax of formula, e.g. cardinality, set minus and quantification over sets, and provide them with the obvious semantics. We will not go into the rather tedious detail here.

Theorem 5.5.2. DISJOINT *and* SPLIT *are sound.*

Proof. Direct from the definition of the semi-group. □

5.5.1 Encoding the counting model in the named model.

Next, we show the semantics of the named model properly encodes the semantics of the counting model. To aid clarity we write, H^c, H^n, P^c, and P^n for heaps (H) in the counting and named models, and formulae (P) in the counting and named models respectively. We define the translation from terms in the counting logic to terms in the named logic as follows:

$$\begin{array}{rcl} [\![e.f \overset{n}{\longmapsto} e']\!] & \overset{\text{def}}{=} & \exists p.\ e.f \overset{\mathbb{P}\setminus p}{\longmapsto} e' \wedge |p| = n \\ [\![e \to e']\!] & \overset{\text{def}}{=} & \exists p.\ e.f \overset{p}{\longmapsto} e' \wedge |p| = 1 \end{array}$$

The source permission with n missing read-only permissions is the set of permissions with n elements missing. The read-only permission is represented by a singleton set of permissions. We omit the obvious definition for the other predicates and connectives.

We must also define a translation from the named model's heaps to counting model's heaps. First, we define a predicate on the permissions in the named model. The predicate identifies the permissions that correspond to permission in the counting model:

$$\mathit{fin}(p) \quad \overset{\text{def}}{=} \quad \exists n.\ |p| = n \vee |\mathbb{P} \setminus p| = n$$

We restrict permissions to finite and cofinite sets as their cardinality can be represented by an integer. We then define a translation from the elements of the named permissions semi-group to the counting permissions semi-group.

$$[\![p]\!] \quad \overset{\text{def}}{=} \quad \begin{cases} -n & |p| = n \\ n & |\mathbb{P} \setminus p| = n \\ \text{undefined} & \text{otherwise} \end{cases}$$

Lemma 5.5.3. *Translation preserves the semi-group action for finite and cofinite permissions, that is*

$$n_1 *_1 n_2 = [\![p_3]\!] \ \Rightarrow\ \exists p_1, p_2.\ [\![p_1]\!] = n_1 \wedge [\![p_2]\!] = n_2 \wedge p_1 \uplus p_2 = p_3 \tag{5.2}$$

$$[\![p_1]\!] *_1 n_2 = n_3 \ \Rightarrow\ \exists p_2, p_3.\ [\![p_2]\!] = n_2 \wedge [\![p_3]\!] = n_3 \wedge p_1 \uplus p_2 = p_3 \tag{5.3}$$

$$\forall p_1, p_2 \in \mathit{fin}.\quad p_1 \uplus p_2 \textit{ defined} \ \Rightarrow\ [\![p_1]\!] *_1 [\![p_2]\!] = [\![p_1 \uplus p_2]\!] \tag{5.4}$$

Proof. The three properties can be shown from the definitions of the semi-group and using case analysis: (5.2) and (5.4) by case analysis on positive and negative ns; and for (5.4) by case analysis on finite and cofinite sets. □

We extend the restriction on permissions, $\mathit{fin}(\_)$, to heaps in the obvious way, and extend the translation on permissions to heaps in the obvious way. We can extend the previous lemma to heaps.

Theorem 5.5.4. *Encoding counting model into named model is sound and complete. For all formulae, P^c, in the counting model, the following holds for all heaps in the named model:*

$$[\![H^n]\!] = H^c \quad \Rightarrow \quad (VS, H^n, I \models [\![P^c]\!] \quad \iff \quad VS, H^c, I \models P^c)$$

Proof. By induction on the structure of P^c. The $*$ and $-\!\!*$ cases require the previous lemma, and $\mapsto$ follows from encoding. Other cases follow trivially. □

5.5.2 Other models

To conclude this chapter, we will discuss how the named permissions model can be used to represent other permissions idioms. Bornat *et al*. [14] identify two key idioms for the dissemination of read-only permissions: counting, given earlier, and splitting. In this subsection we first show how to represent splitting and then we how to properly combine splitting and counting.

Infinite splitting

Boyland [16] first proposed the idea of permissions with weights. The key idea is to split a permission into two pieces such that they sum to the same weight as the original. This idea of splitting is very useful in algorithms which divide work recursively, such as parallel tree substitution [14]. The key to this idiom is that splitting can occur to an unbounded depth: a permission can always be split into two permissions. This style of reasoning exists in the general model due to the following fact: any countably infinite set can be split into two countably infinite sets, i.e.

$$|p| = |\mathbb{N}| \quad \Leftrightarrow \quad \exists p_1, p_2.\ |p_1| = |p_2| = |\mathbb{N}| \wedge p = p_1 \uplus p_2$$

When combined with SPLIT this allows any infinite read permissions to be split into two infinite read permissions.

Note: This does not precisely model the fractional permissions of Bornat *et al*. [14], as they use rational numbers to name these permissions. This additional structure can be encoded into this model, if required, by considering a bijection between the permissions set, $\mathbb{P}$, and an interval in $\mathbb{Q}$. However, this does not present any interesting problems.

Combining splitting and counting

We conclude by showing how splitting and counting can be combined. We want to allow the following:

1. split permissions that have been counted;
2. split permissions that can count, i.e. split source permissions; and
3. count permissions that can count, i.e. count source permissions.

Notation: We use $X \leftrightarrow Y$ to mean a bijection exists from X to Y.

Counting removes finite sets from an infinite set: $\mathbb{P} \leftrightarrow \mathbb{P} + 1$; and splitting takes an infinite set and produces two infinite sets: $\mathbb{P} \leftrightarrow \mathbb{P} + \mathbb{P}$. Everything needs an infinite set, but counting only provides finite sets: we need to count infinite sets!

We can make the following two observations:

$$p \leftrightarrow p' \quad \Rightarrow p \times \mathbb{P} \leftrightarrow p' \times \mathbb{P}$$
$$\mathbb{P} \leftrightarrow \mathbb{P} \times \mathbb{P}$$

Hence, we can count infinite sets with

$$(\mathbb{P} \times \mathbb{P}) \quad \leftrightarrow \quad (\mathbb{P} \times \mathbb{P}) + (1 \times \mathbb{P})$$

Clearly, $1 \times \mathbb{P}$ is infinite so we can count permissions, that themselves can count or be split.

Now let us provide an example of how one might use these ideas for multilevel counting. We will write $prod_{\boldsymbol{p}}$ for a bijection from $\mathbb{P} \times \mathbb{P}$ to $\boldsymbol{p}$, a countably infinite set. We will write $f(p)$ as a shorthand for $\{v | f(v') = v \wedge v' \in p\}$. We have three important facts about this bijection:

$$\begin{gathered}\boldsymbol{p} = prod_{\boldsymbol{p}}(\mathbb{P} \times \mathbb{P}) \\ prod_{\boldsymbol{p}}((\mathbb{P} \setminus (p' \uplus p'')) \times \mathbb{P}) \uplus prod_{\boldsymbol{p}}(p'' \times \mathbb{P}) = prod_{\boldsymbol{p}}((\mathbb{P} \setminus p') \times \mathbb{P}) \\ |p'| \geq 1 \quad \Rightarrow \quad |prod_{\boldsymbol{p}}(p' \times \mathbb{P})| = |\mathbb{P}|\end{gathered}$$

Let us consider how we could use this to do multilevel counting. We define a predicate to represent counting permissions for a resource:

$$counter(x, \boldsymbol{p}, n) \quad \stackrel{\text{def}}{=} \quad \exists p''.\ x \xmapsto{\boldsymbol{p} \setminus prod_{\boldsymbol{p}}(p'' \times \mathbb{P})} \_ \wedge |p''| = n \wedge |\boldsymbol{p}| = |\mathbb{P}|$$

This definition leads to the following equivalence, which can be used to construct and destruct counters.

$$|\boldsymbol{p}| = |\mathbb{P}| \wedge x \xmapsto{\boldsymbol{p}} \_ \quad \Leftrightarrow \quad counter(x, \boldsymbol{p}, 0)$$

Now we want implications to allow us to count out permissions

$$counter(x, \boldsymbol{p}, n) \quad \Leftrightarrow \quad counter(x, \boldsymbol{p}, n+1) * (x \xmapsto{p'} \_ \wedge |p'| = |\mathbb{P}|) * counted(\boldsymbol{p}, p')$$

We need an additional predicate to represent the information that p' is a permission counted from $\boldsymbol{p}$. This can be defined as

$$counted(\boldsymbol{p}, p') \quad \stackrel{\text{def}}{=} \quad \exists p''.\ p' = prod_{\boldsymbol{p}}(p'' \times \mathbb{P}) \wedge |p''| = 1$$

These predicates show how this model supports multilevel counting.

5.6 Concluding remarks

The named permissions model is highly expressive and for many examples using a simpler model, such as counting, may be useful. One issue is a sensible syntax for representing the different idioms. The predicates for counting given above abstract the details of the required bijections, and similar predicates can be provided for splitting. However, it may be that a better syntax can be found for expressing the permissions. Ross and Bornat have independently suggested using strings to represent permissions. This may prove a more concise notion than the sets and bijections used above, but this remains further work.

6

Inheritance

So far we have given a logic that is capable of reasoning about programs which use encapsulation. There is a second form of abstraction used in object-oriented programming: inheritance. In this chapter we present the extensions required to allow reasoning about inheritance.

We present an extension to abstract predicates, called abstract predicate families, that allow each class to specify its own definition for that class. This allows subtypes to have a different internal representation. We demonstrate the usefulness of abstract predicate families by considering the Cell/Recell classes, and an extended example of the visitor design pattern [40].

The rest of the chapter is structured as follows. We begin, in §6.1 by discussing the difficulties in reasoning about inheritance. In §6.2, we present our solution: abstract predicate families. In §6.3, we illustrate their use with two examples. In §6.4, we present the formal semantics of this extension. We conclude, in §6.5, by discussing an open issue of reverifying inherited methods and present some possible solutions.

6.1 The problem

First let us revisit the problem shown in §3.8 to highlight why standard separation logic (even extended with abstract predicates) cannot deal with inheritance. Consider the `Cell` class and a subclass that has backup, `Recell`, presented in Figure 6.1. We give the obvious specifications

```
class Cell extends Object {
  Object cnts;

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

  Object get() {
    Object temp;
    temp = this.cnts;
    return temp;
  }
}
```

```
class Recell extends Cell {
  Object bak;

  void set(Object o) {
    Object temp;
    temp = this.cnts;
    this.bak = temp;
    this.cnts = o;
  }

}
```

Figure 6.1: Source code for Cell and Recell classes

Method	Pre-condition	Post-condition
`Cell.set(n)`	$\texttt{this.cnts} \mapsto \_$	$\texttt{this.cnts} \mapsto \texttt{n}$
`Recell.set(n)`	$\texttt{this.cnts} \mapsto X$ $* \texttt{this.bak} \mapsto \_$	$\texttt{this.cnts} \mapsto \texttt{n}$ $* \texttt{this.bak} \mapsto X$

Table 6.1: Specifications of `set` methods.

of the `set` methods in Table 6.1. Unfortunately these specifications do not satisfy specification compatibility, or behavioural subtyping.

We require the following two implications to hold

$$\mathit{pre}(\texttt{Cell}, \texttt{set}) \Rightarrow \mathit{pre}(\texttt{Recell}, \texttt{set})$$
$$\mathit{post}(\texttt{Recell}, \texttt{set}) \Rightarrow \mathit{post}(\texttt{Cell}, \texttt{set})$$

where $\mathit{pre}(C, m)$ denotes the pre-condition for the method m in class C, and post denotes the post-condition.

Given the specifications in Table 6.1, these implications can never hold as they require a one element heap to be the same size as a two element heap. What about abstract predicates? The specification for `Recell` must use a different abstract predicate to `Cell` as it has a different body, that is we define the abstract predicates as

$$\mathit{Val}_{\texttt{Cell}}(y, x) \stackrel{\text{def}}{=} y.\texttt{cnts} \mapsto x \tag{6.1}$$
$$\mathit{Val}_{\texttt{Recell}}(x, y, z) \stackrel{\text{def}}{=} x.\texttt{cnts} \mapsto y * x.\texttt{bak} \mapsto z \tag{6.2}$$

and the specifications as

Method	Pre-condition	Post-condition
`Recell.set(n)`	$\mathit{Val}_{\texttt{Recell}}(\texttt{this}, X, \_)$	$\mathit{Val}_{\texttt{Recell}}(\texttt{this}, \texttt{n}, X)$
`Cell.set(n)`	$\mathit{Val}_{\texttt{Cell}}(\texttt{this}, \_)$	$\mathit{Val}_{\texttt{Cell}}(\texttt{this}, \texttt{n})$

Unfortunately the predicates are treated parametrically: no implications hold between them.

As it stands, abstract predicates do not, by themselves, help with behavioural subtyping. They provide support for encapsulation but not inheritance. In an object-oriented setting we require predicates to have multiple definitions, hence we introduce *abstract predicate families* where the families are sets of definitions indexed by class. An instance of an abstract predicate family is written $\alpha(x; \vec{v})$ to indicate that the object x satisfies *one* definition from the abstract predicate family α with arguments $\vec{v}$. The particular definition satisfied depends on the dynamic type of x. In object-oriented programming an object could be from one of many classes; abstract predicate families provide a similar choice of predicate definitions when considering their behaviour.

6.2 Abstract predicate families

We define abstract predicate family definitions, Λ^f, with the following syntax.

$$\Lambda^f := \epsilon \mid (\alpha_C(x; \overline{x}) \stackrel{\text{def}}{=} P), \Lambda^f$$

Λ^f is well-formed if it has at most one entry for each predicate and class name pair, and the free variables of the body, P, are in its argument list, $x; \overline{x}$. The first argument is distinguished as it

is used to index by class. We use a semi-colon, ;, to separate this distinguished arguments, and commas to separate the remaining arguments. We treat Λ^f as a function from predicate and class name pairs to predicate definitions. Each entry corresponds to the definition of an abstract predicate family for a particular class.

One can think of an abstract predicate family as a predicate that existentially hides the type:

$$\alpha(e;\overline{e}) \quad \approx \quad \exists C.\ \Lambda^f(\alpha, C)[e;\overline{e}] \wedge e : C$$

This is *not* the complete truth as it does not account for partial definitions, where the whole class hierarchy is unavailable. This intuition mirrors the static typing of an object: `void m(C c) {...}` is a method that expects something that behaves like a `C`: it might not be a `C` but it will be a subtype of `C`. Abstract predicate families mirror this in the logic by allowing assertions that require a state satisfying a property but where the property's precise definition is unknown.

This semantic intuition, leads to the obvious adaptation of the OPEN and CLOSE axioms from Chapter 4.

OPEN	$\Lambda^f \models (x : C \wedge \alpha(x;\overline{x})) \Rightarrow \Lambda^f(\alpha, C)[x;\overline{x}]$
CLOSE where $\alpha, C \in dom(\Lambda^f)$.	$\Lambda^f \models (x : C \wedge \Lambda^f(\alpha, C)[x;\overline{x}]) \Rightarrow \alpha(x;\overline{x})$

To OPEN or CLOSE a predicate we must know which class contains the definition, and must have that definition in scope.

Our example shows the need to alter the arity of the predicate to reflect casting; the `Recell`'s *Val* predicate has three arguments while the `Cell`'s only has two. Hence we provide the following pair of implications:

WIDEN	$\Lambda^f \models \alpha(x;\overline{x}) \Rightarrow \exists\overline{y}.\alpha(x;\overline{x},\overline{y})$
NARROW	$\Lambda^f \models \exists\overline{y}.\alpha(x;\overline{x},\overline{y}) \Rightarrow \alpha(x;\overline{x})$

As we allow change in arity we must provide an operation to provide too few or too many arguments to the definition. We use square brackets to denote this unusal argument application operation.

$$(\alpha_C(x;\overline{x}) \stackrel{\text{def}}{=} P)[e;\overline{e}] \stackrel{\text{def}}{=} \begin{cases} P[e/x, \overline{e_1}/\overline{x}] & |\overline{e_1}| = |\overline{x}| \text{ and } \overline{e} = \overline{e_1}, \overline{e_2} \\ \exists\overline{y}.P[e/x, (\overline{e},\overline{y})/\overline{x}] & |\overline{e},\overline{y}| = |\overline{x}| \end{cases}$$

If we give a predicate more variables than its definition requires, it ignores them, and if too few, it treats the missing arguments as existentially quantified. This definition of substitution is used to give the families' version of OPEN and CLOSE.

Note: We can OPEN predicates at incorrect arities as the substitution will correctly manipulate the arguments. An alternative approach would be to restrict opening to the correct arity, and use WIDEN and NARROW to get the correct arity. However, this alternative approach complicates the semantics.

Again we have two rules for introducing and eliminating abstract predicate families.

ABSTRACT WEAKENING
$$\frac{\Lambda^f; \Gamma \vdash \{P\}C\{Q\}}{\Lambda^f, \Lambda^{f\prime}; \Gamma \vdash \{P\}C\{Q\}}$$
provided $dom(\Lambda^{f\prime})$ and $dom(\Lambda^f)$ are disjoint.

ABSTRACT ELIMINATION
$$\frac{\Lambda^f, \Lambda^{f\prime}; \Gamma \vdash \{P\}C\{Q\}}{\Lambda^f; \Gamma \vdash \{P\}C\{Q\}}$$
provided the predicate names in P, Q, Γ and Λ^f are not in $dom_a(\Lambda^{f\prime})$ and $dom_a(\Lambda^f)$ is defined as $\{\alpha | (\alpha, C) \in dom(\Lambda^f)\}$.

Abstract predicate families are less symmetric than abstract predicates: weakening allows the introduction for a particular class and predicate, while elimination requires the entire family of definitions to be removed, i.e. it must remove all the classes' definitions for a predicate. This is because it is not possible to give a simple syntactic check for which parts, i.e. classes, of a family will be used.

6.3 Examples

Next we will present a couple of examples to illustrate the use of abstract predicate families.

6.3.1 Cell/Recell

Let us return to our original motivating example. We define an abstract predicate family, Val, with the definitions for `Cell` and `Recell` given earlier in equations (6.1) and (6.2).

We have to validate four methods: `Cell.set`, `Cell.get`, `Recell.set` and `Recell.get`. Even though the bodies of `Cell.get` and `Recell.get` are the same, we must validate both, because they have different predicate definitions. We will return to this point in §6.5.

The proofs all follow in a straightforward manner: we present the proof of the `Recell`'s `set` method here.

$\{Val(\texttt{this}; X, \_) \wedge \texttt{this} : \texttt{Recell}\}$
$\{\texttt{this.cnts} \mapsto X * \texttt{this.bak} \mapsto \_ \wedge \texttt{this} : \texttt{Recell}\}$
```
  temp = this.cnts;
```
$\{\texttt{this.cnts} \mapsto X * \texttt{this.bak} \mapsto \_ \wedge \texttt{this} : \texttt{Recell} \wedge X = \texttt{temp}\}$
```
  this.bak = temp;
```
$\{\texttt{this.cnts} \mapsto X * \texttt{this.bak} \mapsto \texttt{temp} \wedge \texttt{this} : \texttt{Recell} \wedge X = \texttt{temp}\}$
```
  this.cnts = o;
```
$\{\texttt{this.cnts} \mapsto \texttt{o} * \texttt{this.bak} \mapsto \texttt{temp} \wedge \texttt{this} : \texttt{Recell} \wedge X = \texttt{temp}\}$
$\{\texttt{this.cnts} \mapsto \texttt{o} * \texttt{this.bak} \mapsto X \wedge \texttt{this} : \texttt{Recell}\}$
$\{Val(\texttt{this}; \texttt{o}, X)\}$

Additionally, we must prove the method specifications are compatible, defined in the sense of definition 3.5.2 (Page 52). The compatibility of the `set` method follows from the rule of L-CONSEQUENCE and L-AUXVARELIM.

$$\frac{\frac{\vdash \{Val(\texttt{this}; X, \_)\}\_\{Val(\texttt{this}; n, X)\}}{\vdash \{Val(\texttt{this}; \_, \_)\}\_\{Val(\texttt{this}; n, \_)\}}}{\vdash \{Val(\texttt{this}; \_)\}\_\{Val(\texttt{this}; n)\}}$$

The two `get` methods have the same specification, so are obviously compatible.

A client that uses these methods does not need to worry about dynamic dispatch. Consider the following method:

Method	Pre-condition	Post-condition
`NonNullCell.set(x)`	$Val(\texttt{this};\_) \wedge \texttt{x} \neq \texttt{null}$	$Val(\texttt{this};\texttt{x})$
`NonNullCell.get()`	$Val(\texttt{this};X)$	$Val(\texttt{this};X) \wedge X \neq \texttt{null} \wedge X = ret$

$$Val_{\texttt{NonNullCell}}(x;y) \stackrel{\text{def}}{=} x.\texttt{cnts} \mapsto y \wedge y \neq \texttt{null}$$

Table 6.2: Specification for `NonNullCell`.

```
m(Cell c) {
  c.set(c);
}
```

This code simply sets the Cell to point to itself. The code is specified as

Method	Pre-condition	Post-condition
`m(c)`	$Val(c;\_)$	$Val(c;c)$

Now consider calling `m` with a `Recell` object.

$\{true\}$
```
   Recell r = new Recell(x);
```
$\{Val(\texttt{r};\texttt{x},\_) * \texttt{r} : \texttt{Recell}\}$
$\{Val(\texttt{r};\texttt{x},\_)\}$
$\{Val(\texttt{r};\_)\}$
```
   m(r);
```
$\{Val(\texttt{r};\texttt{r})\}$
$\{Val(\texttt{r};\texttt{r},\_)\}$
$\{Val(\texttt{r};\texttt{r},\_) * \texttt{r} : \texttt{Recell}\}$

We use L-CONSEQUENCE to cast the Val predicate to have the correct arity. We need not consider dynamic dispatch at all because of specification compatibility.

The specification of method `m` is weaker than we might like. Based on the implementation we might expect the example's post-condition $\{Val(\texttt{r};\texttt{r},\texttt{x})\}$. However, there are several bodies that satisfy `m`'s specification: for example `c.set(x);c.set(c);`. We can set the `Cell` to have any value, as long as the last value we set is the `Cell` itself. This body acts identically on a `Cell` to the previous body, however on a `Recell` it acts differently. Hence only using the specification we cannot infer the tighter post-condition. This could be deduced if `m` was specified for a `Recell` as well.

Now let us briefly consider another `Cell` class: a `NonNullCell`. It specifies the arguments to `set` must not be `null`. We present its specification in Figure 6.2. A `NonNullCell` is not a subtype of `Cell` as we cannot replace `Cell` with a `NonNullCell`, i.e. `x.set(null);` can only be called on a `Cell`. However, it is possible to make `Cell` a subtype of `NonNullCell`. We must alter the specification of `Cell` so that it has two predicate families: one for the non-null state, and one for the null state. We present the specifications and predicate definitions in Figure 6.2. Now we can show the specification compatibility of these two methods as follows. First we

Method	Pre-condition	Post-condition
`set(y)`	$Val(\texttt{this};\_)$	$(\texttt{y} = \texttt{null} \wedge NVal(\texttt{this})) \vee$
	$\vee\ NVal(\texttt{this})$	$(\texttt{y} \neq \texttt{null} \wedge Val(\texttt{this};\texttt{y}))$
`get()`	$Val(\texttt{this};x) \wedge X = 1$	$(X = 1 \wedge Val(\texttt{this};x) \wedge ret = x \wedge x \neq \texttt{null})$
	$\vee\ NVal(\texttt{this}) \wedge X = 2$	$\vee\ (X = 2 \wedge NVal(\texttt{this}) \wedge ret = \texttt{null})$

$$Val_{\texttt{Cell}}(x;y) \stackrel{\text{def}}{=} x.\texttt{cnts} \mapsto y \wedge y \neq \texttt{null}$$
$$NVal_{\texttt{Cell}}(x) \stackrel{\text{def}}{=} x.\texttt{cnts} \mapsto \texttt{null}$$

Figure 6.2: New specification for `Cell` class.

present the `set` method.

$$\frac{\left\{\begin{array}{r} Val(\texttt{this};x) \\ \vee\ NVal(\texttt{this}) \end{array}\right\} \_ \left\{\begin{array}{r} (y = \texttt{null} \wedge NVal(\texttt{this})) \\ \vee\ (Val(\texttt{this};y) \wedge y \neq \texttt{null}) \end{array}\right\}}{\frac{\left\{y \neq \texttt{null} * \left(\begin{array}{r} Val(\texttt{this};x) \\ \vee\ NVal(\texttt{this}) \end{array}\right)\right\} \_ \left\{\left(\begin{array}{r} (y = \texttt{null} \wedge NVal(\texttt{this})) \\ \vee\ (Val(\texttt{this};y) \wedge y \neq \texttt{null}) \end{array}\right) * y \neq \texttt{null}\right\}}{\{y \neq \texttt{null} \wedge Val(\texttt{this};x)\} \_ \{Val(\texttt{this};y)\}}}$$

Although it is possible for the `Cell`'s set method to end in a state that is not expected by the `NonNullCell`'s post-condition, this proof shows it is not possible to end in an unexpected state if the `NonNullCell`'s pre-condition is satisfied. We use the frame rule to preserve the pre-condition that $y \neq \texttt{null}$ across the call.

Note: We have used the ghost variable to encode the specification intersection as described in §3.5.

Next we present a similar proof for the `get()` method.

$$\frac{\left\{\begin{array}{r} Val(\texttt{this};x) \wedge X = 1 \\ \vee\ NVal(\texttt{this}) \wedge X = 2 \end{array}\right\} \_ \left\{\begin{array}{r} (X = 1 \wedge Val(\texttt{this};x) \wedge ret = x \neq \texttt{null}) \\ \vee\ (X = 2 \wedge NVal(\texttt{this}) \wedge ret = \texttt{null}) \end{array}\right\}}{\frac{\left\{\begin{array}{r} \left(\begin{array}{r} Val(\texttt{this};x) \wedge X = 1 \\ \vee\ NVal(\texttt{this}) \wedge X = 2 \end{array}\right) \\ * \ X = 1 \end{array}\right\} \_ \left\{\begin{array}{r} \left(\begin{array}{r} (X = 1 \wedge Val(\texttt{this};x) \wedge ret = x \neq \texttt{null}) \\ \vee\ (X = 2 \wedge NVal(\texttt{this}) \wedge ret = \texttt{null}) \end{array}\right) \\ * \ X = 1 \end{array}\right\}}{\frac{\{Val(\texttt{this};x) \wedge X = 1\} \_ \{Val(this;x) \wedge x = ret \wedge x \neq \texttt{null}\}}{\frac{\{\exists X.\ Val(\texttt{this};x) \wedge X = 1\} \_ \{Val(this;x) \wedge x = ret \wedge x \neq \texttt{null}\}}{\{Val(\texttt{this};x)\} \_ \{Val(\texttt{this};x) \wedge x = ret \wedge x \neq \texttt{null}\}}}}$$

This proof is slightly more complicated as we must eliminate the ghost variable X and preserve its value across the call using the frame rule. However, once again we see that although the method has a more general post-condition, it is a valid replacement for the non-null version.

6.3.2 Visitor pattern

Now let us consider an extended example using the visitor design pattern [40]. The visitor pattern allows code in object-oriented programs to be function-oriented rather than data-oriented. That is, all the methods for a particular operation are grouped in a single class, rather than

```
class Visitor {
 void visitC(Const x) {;}
 void visitP(Plus x) {;}
}

class Ast extends Object {
  void accept(Visitor x) {;}
}

class Const extends Ast {
  int v;
  void accept(Visitor x) {
    x.visitC(this);
  }
}

class Plus extends Ast {
  Ast l; Ast r;
  void accept(Visitor x) {
    x.visitP(this);
  }
}
```

Figure 6.3: Source code for formulae and visitor classes.

grouping all the operations on a particular type of data in a single class. In functional languages, such as ML, a particular operation (function) is defined using pattern matching. The visitor pattern allows object-oriented programmers to express this common idiom by allowing matching on an object's type without using the `instanceof` keyword found in Java. We consider an imperative, external visitor [18] as we do not wish to focus on any of the complexities in typing visitors [17].

We consider a visitor over a very simple syntax of formulae with just addition of two terms and constants. We present the source code in Figure 6.3. In this example we will use a primitive type of integer, `int`, but for compactness we will not formally extend the logic or language's semantics. The `Visitor` class has two methods: the first, `visitC` is invoked when the visitor visits a `Const` node; and the second, `visitP` is for a `Plus` node. The `Visitor` class is a template that should be overridden to produce more interesting visitors.[1] We define three classes to represent an abstract syntax tree of these formulae: `Ast` which is used as a common parent for the term constructors; `Const` that represents constant terms; and `Plus` that represents the addition of two terms. We define a single method, `accept`, in `Ast` that is overridden in each of the subclasses. The `Const` class represents an integer constant, and calls `visitC` when it accepts a visitor. The `Plus` class represents an addition of two formulae and calls `visitP` when it accepts a visitor. The double invocation of calling `accept` followed by `visit` is called double dispatch, and allows the case switch to occur without any downcasts or `instanceof` checks. We will return to this point in the proof later.

Before we can formally specify the visitor, we must extend the logic to have values of the formulae's syntax:

$$\begin{array}{rcl} \tau & ::= & n \mid \tau \underline{+} \tau \\ \mu & ::= & \bullet \mid \tau \underline{+} \mu \mid \mu \underline{+} \tau \end{array}$$

We underline the plus constructor, $\underline{+}$, to distinguish it from the arithmetic operaton, and use $\mu[\mu']$ to mean replace $\bullet$ by μ' in μ. We give the specifications for the `visit` and `accept` methods in Table 6.3. There are many choices one can make for the specification of a visitor. The μ parameter is used to allow the evaluation to depend on the context. In the example that follows we use the context to allow us to accumulate the value of the expression as we traverse it, rather than having to calculate it in a bottom up fashion. One can remove the μ parameter from the predicates if we do not want context sensitive visitors. That is, the state of the visitor does not depend on its context. One might have expected the methods all to have post-conditions of

[1] One would normally make the `Visitor` class abstract, but this is not in our syntax. Instead, we can make its constructor's pre-condition *false* (not presented). Thus it can never be instantiated.

Method	Pre-condition	Post-condition
`visitC(x)`	$\texttt{x} : \texttt{Const} \wedge \mathit{Ast}(\texttt{x}, \tau) * \mathit{Visitor}(\texttt{this}, \mu)$	$\mathit{Visited}(\texttt{this}; \texttt{x}, \tau, \mu)$
`visitP(x)`	$\texttt{x} : \texttt{Plus} \wedge \mathit{Ast}(\texttt{x}, \tau) * \mathit{Visitor}(\texttt{this}, \mu)$	$\mathit{Visited}(\texttt{this}; \texttt{x}, \tau, \mu)$
`accept(x)`	$\mathit{Ast}(\texttt{this}, \tau) * \mathit{Visitor}(\texttt{x}, \mu)$	$\mathit{Visited}(\texttt{x}; \texttt{this}, \tau, \mu)$

Table 6.3: Specifications for accept and visit methods.

```
class Calc
    extends Visitor {
 int amount;
 void visitC(Const x) {
   this.amount += x.n;
 }
 void visitP(Plus x) {
   x.l.accept(this);
   x.r.accept(this);
 }
}
```

Figure 6.4: Source code for `Calc` visitor

the form $\mathit{Visited}(\ldots) * \mathit{Ast}(x, \tau)$. However, this kind of specification prevents us altering the structure of the expression.

We define the Ast predicate family for the three classes.

$$
\begin{array}{lcl}
\mathit{Ast}_{\texttt{Ast}}(x, \tau) & \stackrel{\text{def}}{=} & \mathit{false} \\
\mathit{Ast}_{\texttt{Const}}(x, \tau) & \stackrel{\text{def}}{=} & x.\texttt{v} \mapsto n \wedge \tau = n \\
\mathit{Ast}_{\texttt{Plus}}(x, \tau) & \stackrel{\text{def}}{=} & x.\texttt{l} \mapsto i * x.\texttt{r} \mapsto j * \mathit{Ast}(i, \tau_l) * \mathit{Ast}(j, \tau_r) \wedge \tau = \tau_l \pm \tau_r
\end{array}
$$

Note: Defining the predicate for the `Ast` class as *false* prevents any invocation of its methods.

We are now in a position to verify the `accept` methods for the `Plus` and `Const` classes. We only present the `Plus` case as the `Const` case is very similar.

$\{\mathit{Ast}(\texttt{this} : \tau) * \mathit{Visitor}(\texttt{x}; \mu) \wedge \texttt{this} : \texttt{Plus}\}$
```
  x.visitP(this);
```
$\{\mathit{Visitor}(\texttt{x}; \texttt{this}, \tau, \mu)\}$

The verification of the method call introduces the current class's type: this is exactly what is required to meet `visitP`'s specification. This method call simply provides information about which type of node this is to the `Visitor`: it is providing a case select using dynamic dispatch. This proof does not need changing no matter how many subclasses of `Visitor` are added.

Let us consider an actual visitor implementation. The implementation given in Figure 6.4 calculates the value of the formulae. We can define the $\mathit{Visitor}$ predicate for this class as

$$
\begin{array}{rcl}
\mathit{Visitor}_{\texttt{Calc}}(x; \mu) & \stackrel{\text{def}}{=} & x.\texttt{amount} \mapsto \mathit{lcalc}(\mu) \\
\mathit{Visited}_{\texttt{Calc}}(x; y, \tau, \mu) & \stackrel{\text{def}}{=} & x.\texttt{amount} \mapsto (\mathit{lcalc}(\mu) + \mathit{calc}(\tau)) * \mathit{Ast}(y; \tau)
\end{array}
$$

where we define the function $calc$ and $lcalc$ as:

$$calc(\tau) \stackrel{\text{def}}{=} \begin{cases} n & \tau = n \\ calc(\tau_1) + calc(\tau_2) & \tau = \tau_1 \pm \tau_2 \end{cases}$$

$$lcalc(\mu) \stackrel{\text{def}}{=} \begin{cases} 0 & \mu = \bullet \\ lcalc(\mu_1) & \mu = \mu_1 \pm \tau \\ calc(\tau) + lcalc(\mu_1) & \mu = \tau \pm \mu_1 \end{cases}$$

The $lcalc$ function is used to calculate the accumulated total from the context: the sum of everything to the left of the hole, $\bullet$, i.e. the nodes we have already visited.

We verify the `visitP` method as[2]

$\{\mathtt{x} : \mathtt{Plus} \wedge \mathit{Ast}(\mathtt{x}; \tau) * \mathit{Visitor}(\mathtt{this}; \mu) \wedge \mathtt{this} : \mathtt{Calc}\}$
$\{P * \mathtt{x.l} \mapsto i * \mathtt{x.r} \mapsto j * \mathit{Ast}(i, \tau_1) * \mathit{Ast}(j, \tau_2) * \mathit{Visitor}(\mathtt{this}; \mu)\}$
$\{\mathtt{x.l} \mapsto i * \mathit{Ast}(i, \tau_1) * \mathit{Visitor}(\mathtt{this}; \mu, \bullet)\}$
`x.l.accept(this);`
$\{\mathtt{x.l} \mapsto i * \mathit{Visited}(\mathtt{this}; i, \tau_1, \mu)\}$
$\{P * \mathtt{x.l} \mapsto i * \mathtt{x.r} \mapsto j * \mathit{Ast}(j, \tau_2) * \mathit{Visited}(\mathtt{this}; i, \tau_1, \mu)\}$
$\{P * \mathtt{x.l} \mapsto i * \mathtt{x.r} \mapsto j * \mathit{Ast}(i, \tau_1) * \mathit{Ast}(j, \tau_2) * \mathit{Visitor}(\mathtt{this}; \mu[\tau_1 \pm \bullet])\}$
$\{\mathtt{x.r} \mapsto j * \mathit{Ast}(j, \tau_2) * \mathit{Visitor}(\mathtt{this}; \mu[\tau_1 \pm \bullet])\}$
`x.r.accept(this);`
$\{\mathtt{x.r} \mapsto j * \mathit{Visited}(\mathtt{this}; j, \tau_2, \mu[\tau_1 \pm \bullet])\}$
$\{P * \mathtt{x.l} \mapsto i * \mathtt{x.r} \mapsto j * \mathit{Ast}(i, \tau_1) * \mathit{Visited}(\mathtt{this}; j, \tau, \mu)\}$
$\{\mathit{Visited}(\mathtt{this}; x, \tau, \mu)\}$

where $P = \mathtt{x} : \mathtt{Plus} \wedge \mathtt{this} : \mathtt{Calc} \wedge \tau = \tau_1 \pm \tau_2$

This proof follows from the following two formulae.

$$x : \mathtt{Calc} \quad \Rightarrow \quad \mathit{Visited}(x; y, \tau, \mu) \quad \Leftrightarrow \quad \mathit{Visitor}(x; \mu[\tau \pm \bullet]) * \mathit{Ast}(y, \tau)$$

$$\begin{pmatrix} \mathit{Ast}(i, \tau_1) * x.\mathtt{l} \mapsto i * x.\mathtt{r} \mapsto j \wedge y : \mathtt{Calc} \\ * \ \mathit{Visited}(y; j, \tau_2, \mu[\tau_1 \pm \bullet]) \wedge x : \mathtt{Plus} \end{pmatrix} \quad \Leftrightarrow \quad \mathit{Visited}(this; x, \tau_1 \pm \tau_2, \mu)$$

Both of which follow directly from $lcalc(\mu) + calc(\tau) = lcalc(\mu[\tau \pm \bullet])$ which can be shown by induction on μ.

We can produce different subclasses of `Visitor` to perform many different functions, such as checking whether an AST contains a particular constant, or to clone an AST. As the proof is modular we only need to verify the new classes we write: we know all the other classes interact correctly.

Next we consider a subclass of `Visitor` that mutates the structure of an expression by removing any addition of zero. First, we present a function that operates on the abstract syntax.

$$rz(\tau_1 \pm \tau_2) \stackrel{\text{def}}{=} \begin{cases} rz(\tau_1) & rz(\tau_2) = 0 \\ rz(\tau_2) & rz(\tau_1) = 0 \\ rz(\tau_1) \pm rz(\tau_2) & \text{otherwise} \end{cases}$$

$$rz(n) \stackrel{\text{def}}{=} n$$

This function would simplify $0 \pm (3 \pm 4)$ to 3 ± 4, and 0 ± 0 to 0. It cannot remove all occurrences of zero, but it removes all additions of zero. We present a program that does this in Figure 6.5. We have used MJ syntax extended with booleans for compactness and legibility. The `Visitor` does an in-place removal: it does not allocate any storage; instead it just modifies the original

```
class RemoveZeros
         extends Visitor {
  boolean isZero;
  boolean isChanged;
  Ast newl;

  void visitC(Const c) {
    if(c.n==0) {
      this.isZero = true;
    }
  }

  void visitP(Plus p) {
    p.l.accept(this);
    if(this.isZero) {
      this.isChanged = false;
      this.isZero = false;
      p.r.accept(this);
      if(!this.isChanged) {
        this.newl = p.r;
        this.isChanged = true;
      }
    } else {
      if(this.isChanged) {
        p.l = this.newl;
        this.isChanged = false;
      }
      p.r.accept(this);
      if(this.isZero) {
        this.isChanged = true;
        this.newl = p.l;
        this.isZero = false;
      }
      else if(isChanged) {
        p.r = this.newl;
        this.isChanged = false;
      }
    }
  }
}
```

Figure 6.5: Source code of `RemoveZeros` visitor

structure. Rather than explain the rather difficult corners cases we will present the proof. This highlights why each item is required. We use `RZ` as a shorthand for `RemoveZeros`. We specify the *Visitor* and *Visited* predicates as follows

$$\begin{array}{rcl} \mathit{Visitor}_{\texttt{RZ}}(x;\mu) & \stackrel{\text{def}}{=} & x.\texttt{newl} \mapsto \_ * x.\texttt{isZero} \mapsto \mathit{false} * x.\texttt{isChanged} \mapsto \mathit{false} \\ \mathit{Visited}_{\texttt{RZ}}(x;y,\tau,\mu) & \stackrel{\text{def}}{=} & x.\texttt{newl} \mapsto a * x.\texttt{isZero} \mapsto b * x.\texttt{isChanged} \mapsto c \\ & & * \ ((c \wedge \mathit{Ast}(a;rz(\tau)) \vee (\neg c \wedge \mathit{Ast}(y;rz(\tau))) \wedge (b \Leftrightarrow rz(\tau) = 0) \end{array}$$

The `isZero` variable specifies that the expression just visited is equivalent to zero: it might have been a single constant 0 or any composite expression with only zero constants, e.g. 0 ± 0. The `isChanged` variable indicates if the removal of zeros altered the top node, and hence we must now link to a sub-term and the `newl` variable denotes this sub-term. We present an overview of the proof in Figure 6.6. We do not present all the tedious details, but instead consider the highlighted part of the proof in Figure 6.7. The most interesting step is at the end where we use weakening to remove the `p.l` and `p.r` fields.

This proof demonstrates the support that abstract predicate families provides for inheritance. The imperative nature of the visitor is well suited to separation logic. The reader should note that this algorithm is non-trivial to implement: indeed, the author's initial implementation had several small mistakes. Although testing would have caught some of these errors, formal verification guarantees none still exist.

6.4 Semantics

In this section we consider the extensions to the semantics of §4.3 sufficient to model abstract predicate families. The semantic predicate environment as defined in §4.3 has to be extended to handle the arity changes that predicate families require. We define a semantic predicate family

[2]Rather than present the code broken into simple Inner MJ statements, we will present a proof skeleton using MJ statements.

$\{\texttt{p} : \texttt{Plus} \wedge \mathit{Ast}(\texttt{p};\tau) * \mathit{Visitor}(\texttt{this};\mu) \wedge \texttt{this} : \texttt{RZ}\}$
$\{\texttt{p} : \texttt{Plus} \wedge \texttt{p.l} \mapsto i * \mathit{Ast}(i;\tau_1) * \texttt{p.r} \mapsto j * \mathit{Ast}(j;\tau_2) * \tau_1 \pm \tau_2 = \tau$
$\quad * \mathit{Visitor}(\texttt{this};\mu) \wedge \texttt{this} : \texttt{RZ}\}$

```
  p.l.accept(this);
```

$\{\texttt{p} : \texttt{Plus} \wedge \texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * \mathit{Ast}(j;\tau_2) * \tau_1 \pm \tau_2 = \tau$
$\quad * \mathit{Visited}(\texttt{this};i,\tau_1,\mu) \wedge \texttt{this} : \texttt{RZ}\}$

```
  if(this.isZero) {
```

$\{\texttt{p} : \texttt{Plus} \wedge \texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * \mathit{Ast}(j;\tau_2) * \tau_1 \pm \tau_2 = \tau$
$* \mathit{Visited}(\texttt{this};i,\tau_1,\mu) \wedge \texttt{this.isZero} \mapsto \mathit{true} \wedge \texttt{this} : \texttt{RZ}\}$

```
    this.isZero = false;
    this.isChanged = false;
```

$\{\texttt{p} : \texttt{Plus} \wedge \texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * \mathit{Ast}(j;\tau_2) * \tau_1 \pm \tau_2 = \tau$
$\quad * \mathit{Visitor}(\texttt{this};\mu) \wedge \texttt{this} : \texttt{RZ} \wedge rz(\tau_1) = 0\}$

```
    p.r.accept(this);
```

$\{\texttt{p} : \texttt{Plus} \wedge \texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * \tau_1 \pm \tau_2 = \tau$
$* \mathit{Visited}(\texttt{this};j,\tau_2,\mu) \wedge \texttt{this} : \texttt{RZ} \wedge rz(\tau_1) = 0\}$

```
    if(!this.isChanged) {
       this.newl = p.r;
       this.isChanged = true;
    }
```

$\{\mathit{Visited}(\texttt{this};p,\tau,\mu)\}$

} Figure 6.7

```
  } else {
```

$\{\texttt{p} : \texttt{Plus} \wedge \texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * \mathit{Ast}(j;\tau_2) * \tau_1 \pm \tau_2 = \tau$
$* \mathit{Visited}(\texttt{this};i,\tau_1\mu) \wedge \texttt{this.isZero} \mapsto \mathit{false} \wedge \texttt{this} : \texttt{RZ}\}$

```
    if(this.isChanged) {
      p.l = this.newl;
      this.isChanged = false;
    }
```

$\{\exists k.\ \texttt{p} : \texttt{Plus} \wedge \texttt{p.l} \mapsto k * \texttt{p.r} \mapsto j * \mathit{Ast}(j;\tau_2) * \mathit{Ast}(k,rz(\tau_1)) * \tau_1 \pm \tau_2 = \tau$
$\quad * \mathit{Visitor}(\texttt{this};\mu) \wedge \texttt{this} : \texttt{RZ} \wedge rz(\tau_1) \neq 0\}$

```
    p.r.accept(this);
```

$\{\exists k.\ \texttt{p} : \texttt{Plus} \wedge \texttt{p.l} \mapsto k * \texttt{p.r} \mapsto j * \mathit{Ast}(k,rz(\tau_1)) * \tau_1 \pm \tau_2 = \tau$
$\quad * \mathit{Visited}(\texttt{this};j,\tau_2,\mu) \wedge \texttt{this} : \texttt{RZ} \wedge rz(\tau_1) \neq 0\}$

```
    if(this.isZero) {
      this.isChanged = true;
      this.newl = p.l;
      this.isZero = false;
    }
    else if(isChanged) {
      p.r = this.newl;
      this.isChanged = false;
    }
```

$\{\mathit{Visited}(\texttt{this};p,\tau,\mu)\}$

```
  }
```

$\{\mathit{Visited}(\texttt{this};p,\tau,\mu)\}$

Figure 6.6: `RemoveZeros`' proof overview.

$\{\texttt{p} : \texttt{Plus} \wedge \texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * \tau_1 \pm \tau_2 = \tau \wedge \texttt{this} : \texttt{RZ} \wedge rz(\tau_1) = 0 * \mathit{Visited}(\texttt{this}; j, \tau_2, \mu)\}$

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * rz(\tau) = rz(\tau_2) \wedge \texttt{this} : \texttt{RZ} * \mathit{Visited}(\texttt{this}; j, \tau_2, \mu) \wedge \texttt{this.isChanged} \mapsto \_\}$

```
b = this.isChanged;
```

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * rz(\tau) = rz(\tau_2) \wedge \texttt{this} : \texttt{RZ} * \mathit{Visited}(\texttt{this}; j, \tau_2, \mu) \wedge \texttt{this.isChanged} \mapsto \texttt{b}\}$

```
if(!b) {
```

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * rz(\tau) = rz(\tau_2) \wedge \texttt{this} : \texttt{RZ} * \mathit{Visited}(\texttt{this}; j, \tau_2, \mu) \wedge \texttt{this.isChanged} \mapsto \mathit{false}\}$

```
t = p.r;
```

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto t * rz(\tau) = rz(\tau_2) \wedge \texttt{this} : \texttt{RZ} * \mathit{Visited}(\texttt{this}; t, \tau_2, \mu) \wedge \texttt{this.isChanged} \mapsto \mathit{false}\}$

By expanding the definition of *Visited*

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto t * rz(\tau) = rz(\tau_2) \wedge \texttt{this} : \texttt{RZ} * \mathit{Ast}(t; rz(\tau_2)) * \texttt{this.newl} \mapsto \_ * \texttt{this.isChanged} \mapsto \mathit{false} * \texttt{this.isZero} \mapsto c * (c \Leftrightarrow rz(\tau_2) = 0)\}$

By replacing $rz(\tau_2)$ with $rz(\tau)$

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto t \wedge \texttt{this} : \texttt{RZ} * \mathit{Ast}(t; rz(\tau)) * \texttt{this.newl} \mapsto \_ * \texttt{this.isChanged} \mapsto \mathit{false} * \texttt{this.isZero} \mapsto c * (c \Leftrightarrow rz(\tau) = 0)\}$

```
this.newl = t;
```

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto t \wedge \texttt{this} : \texttt{RZ} * \mathit{Ast}(t; rz(\tau)) * \texttt{this.newl} \mapsto t * \texttt{this.isChanged} \mapsto \mathit{false} * \texttt{this.isZero} \mapsto c * (c \Leftrightarrow rz(\tau) = 0)\}$

```
this.isChanged = true;
```

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto t \wedge \texttt{this} : \texttt{RZ} * \mathit{Ast}(t; rz(\tau)) * \texttt{this.newl} \mapsto t * \texttt{this.isChanged} \mapsto \mathit{true} * \texttt{this.isZero} \mapsto c * (c \Leftrightarrow rz(\tau) = 0)\}$

By the definition of *Visited*

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto t * \mathit{Visited}(\mathit{this}; p, \tau, \mu)\}$

```
} else {
```

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * rz(\tau) = rz(\tau_2) \wedge \texttt{this} : \texttt{RZ} * \mathit{Visited}(\texttt{this}; j, \tau_2, \mu) \wedge \texttt{this.isChanged} \mapsto \mathit{true}\}$

By expanding the definition of *Visited*

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * rz(\tau) = rz(\tau_2) \wedge \texttt{this} : \texttt{RZ} * \texttt{this.newl} \mapsto k * \mathit{Ast}(k; rz(\tau_2)) * \texttt{this.isChanged} \mapsto \mathit{true} * \texttt{this.isZero} \mapsto c * (c \Leftrightarrow rz(\tau_2) = 0)\}$

By replacing $rz(\tau)$ with $rz(\tau_2)$

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j \wedge \texttt{this} : \texttt{RZ} * \texttt{this.newl} \mapsto k * \mathit{Ast}(k; rz(\tau)) * \texttt{this.isChanged} \mapsto \mathit{true} * \texttt{this.isZero} \mapsto c * (c \Leftrightarrow rz(\tau) = 0)\}$

By the definition of *Visited*

$\{\texttt{p.l} \mapsto i * \texttt{p.r} \mapsto j * \mathit{Visited}(\texttt{this}; \texttt{p}, \tau, \mu)\}$

```
;
}
```

$\{\mathit{Visited}(\texttt{this}; p, \tau, \mu) * \texttt{p.l} \mapsto \_ * \texttt{p.r} \mapsto \_\}$

By weakening

$\{\mathit{Visited}(\texttt{this}; p, \tau, \mu)\}$

Figure 6.7: Part of `RemoveZeros`' proof

environment as

$$\Delta_f : \mathcal{A} \times \mathsf{Classes} \rightharpoonup \left(\mathbb{N}^+ \to \mathcal{P}(\mathcal{H})\right)$$

This is a partial function from pairs of predicate, $\mathcal{A}$, and class name, Classes, to semantic definitions. An abstract predicate family is defined for all arities, so the semantic definition must be a function from *all* tuples of non-zero arity. This semantically supports the change in arity required by WIDEN and NARROW.

We can now give the semantics of the new abstract predicate family assertion as follows

$$VS, H, I \models_{\Delta_f} \alpha(e; \overline{e}) \quad \Leftrightarrow \quad H \in (\Delta_f(\alpha, C))[[\![e; \overline{e}]\!]_{VS,I}] \wedge H([\![e]\!]_{VS,I}) = C$$

The assertion $\alpha(e; \overline{e})$ holds for some heap H, iff $[\![e]\!]_{VS,I}$ has class C, and H satisfies the predicate definition for C, given arguments $[\![e, \overline{e}]\!]_{VS,I}$, in the predicate family α.

To ensure that WIDEN and NARROW hold we restrict our attention to *argument refineable* environments.

Definition 6.4.1 (Argument refineable)**.** A semantic predicate family environment is said to be argument refineable if adding an argument cannot increase, or decrease, the set of accepting states, i.e.

$$AR(\Delta_f) \quad \Leftrightarrow \quad \forall \alpha, n, \overline{n}.\ \Delta_f(\alpha)[n; \overline{n}] = \bigcup_{n'} \Delta_f(\alpha)[n; \overline{n}, n']$$

Next we prove this semantics condition allows the arity change provided by WIDEN and NARROW.

Proposition 6.4.2. *Argument refinement coincides precisely with* WIDEN *and* NARROW*:* $\forall\, VS, H, I, \alpha, n, \overline{n}.$

$$AR(\Delta_f) \Longleftrightarrow (VS, H, I \models_{\Delta_f} \alpha(n; \overline{n}) \ \Leftrightarrow\ \exists n'.\ \alpha(n; \overline{n}, n'))$$

Proof. Straight from the definitions of argument refineable, and the semantics of abstract predicate families and existential quantifiers. □

Again the semantic predicate family environment is a complete lattice. The lattice has the following order:

$$\Delta_f \sqsubseteq \Delta'_f \stackrel{\text{def}}{=} \forall \alpha, C, n, \overline{n}.\ (\alpha, C) \in dom(\Delta_f) \ \Rightarrow\ \Delta_f(\alpha, C)[n; \overline{n}] \subseteq \Delta_f(\alpha, C)[n; \overline{n}]$$

Again, the least upper bound of the order is written $\sqcup$. Lemmas 4.3.1 and 4.3.4 can be extended to semantic predicate family environments as follows:

Lemma 6.4.3. *Argument refineable predicate family environments form a complete lattice with respect to* $\sqsubseteq$.

Proof. We must show that the least upper bound of a set of argument refineable environments is also argument refineable.

$$(\forall i. AR(\Delta^i_f)) \quad \Rightarrow \quad AR(\bigsqcup_i \Delta^i_f)$$

That is shown by

$$\begin{aligned}
H \in (\bigsqcup_i \Delta^i_f)(\alpha)[n; \overline{n}] \ &\Leftrightarrow\ \exists i.\ H \in \Delta^i_f(\alpha)[n; \overline{n}] \\
&\Leftrightarrow\ \exists i, n'.\ H \in \Delta^i_f(\alpha)[n; \overline{n}, n'] \\
&\Leftrightarrow\ \exists n'.\ H \in (\bigsqcup_i \Delta^i_f)(\alpha)[n; \overline{n}, n'] \\
&\Leftrightarrow\ H \in \bigcup_{n'} (\bigsqcup_i \Delta^i_f)(\alpha)[n; \overline{n}, n']
\end{aligned}$$

□

Lemma 6.4.4. *Positive formulae are monotonic with respect to semantic predicate family environments*

$$\Delta_f \sqsubseteq \Delta'_f \ \wedge \ VS,H,I \models_{\Delta_f} P \ \Rightarrow \ VS,H,I \models_{\Delta'_f} P$$

Proof. By induction on P. □

However extending Lemma 4.3.2 is less straightforward, as it is not possible to tell which predicate name, class name pairs are used in a formula.

Lemma 6.4.5. *Formulae only depend on the abstractions they mention. If Δ_f contains all the abstractions in P, and $dom_a(\Delta_f) \cap dom_a(\Delta'_f) = \emptyset$, then*

$$\forall VS,H,I.\ VS,H,I \models_{\Delta_f} P \Leftrightarrow VS,H,I \models_{\Delta_f \sqcup \Delta'_f} P$$

Proof. By induction on P. □

Now let us consider the construction of semantic predicate family environments from their abstract syntactic counterparts. We define a new function, $stepf$, that accounts for the first argument's type and uses the special substitution,

$$stepf_{(\Lambda^f,\Delta_f)}(\Delta'_f)(\alpha,C)[n;\overline{n}] \ \stackrel{\text{def}}{=} \ \{H \mid H(n) = C \wedge VS,H,I \models_{\Delta_f \sqcup \Delta'_f} \Lambda^f(\alpha,C)[n;\overline{n}]\}$$

This function is monotonic on predicate family environments, because of Lemma 6.4.4 and that all the predicate definitions are positive. Hence by Lemma 6.4.3 and Tarski's theorem we know a fixed point must always exist. We write $[\![\Lambda^f]\!]_{\Delta_f}$ as the least fixed point of $stepf_{(\Lambda^f,\Delta_f)}$.

Lemma 6.4.6 (The fixed point is argument refineable)**.** *If $AR(\Delta_f)$ then $AR([\![\Lambda^f]\!]_{\Delta_f})$*

Proof. We show any fixed point is argument refineable. We know the following two equations hold:

$$\begin{aligned} VS,H,I \models_{\Delta_f} \Lambda^f(\alpha,C)[n;\overline{n}] \ \wedge \ n : C \quad &\Longleftrightarrow \quad H \in \Delta_f(\alpha,C)[n;\overline{n}] \\ VS,H,I \models_{\Delta_f} \Lambda^f(\alpha,C)[n;\overline{n}] &\Leftrightarrow \exists n'.\ \Lambda^f(\alpha,C)[n;\overline{n},n'] \end{aligned}$$

The first follows from the definition of the fixed point, and the second follows from the definition of substitution on predicate family definitions. Combining these we get: for some n'

$$\begin{array}{c} VS,H,I \models_{\Delta_f} \Lambda^f(\alpha,C)[n;\overline{n}] \ \wedge \ n : C \Longleftrightarrow H \in \Delta_f(\alpha,C)[n;\overline{n}] \\ \Updownarrow \\ VS,H,I \models_{\Delta_f} \Lambda^f(\alpha,C)[n;\overline{n},n'] \ \wedge \ n : C \Longleftrightarrow H \in \Delta_f(\alpha,C)[n;\overline{n},n'] \end{array}$$

Hence any fixed point is argument refineable. □

Consider the following set of solutions:

$$close(\Lambda^f) \quad \stackrel{\text{def}}{=} \quad \{[\![\Lambda^f]\!]_{\Delta_f} \sqcup \Delta_f \mid ((\mathcal{A} \times \mathsf{Classes}) \setminus dom(\Delta_f)) = dom(\Lambda^f) \ \wedge \ AR(\Delta_f)\}$$

This satisfies the analogues of Lemmas 4.3.6 and 4.3.7.

Lemma 6.4.7. *Adding new predicate definitions refines the set of possible semantic predicate environments.*

$$close(\Lambda^f) \supseteq close(\Lambda^f, \Lambda^{f'})$$

Proof. Identical to proof of Lemma 4.3.6. □

Lemma 6.4.8. *The removal of predicate definitions does not affect predicates that do not use them, i.e. given Λ^f which is disjoint from $\Lambda^{f'}$ and does not mention predicates in its domain; we have*

$$\forall \Delta \in close(\Lambda^f).\exists \Delta'_f \in close(\Lambda^f, \Lambda^{f'}).\ \Delta_f \upharpoonright dom(\Lambda^{f'}) \ = \ \Delta'_f \upharpoonright dom(\Lambda^{f'})$$

where $f \upharpoonright S$ is $\{a \mapsto b | a \mapsto b \in f \wedge a \notin dom(S)\}$

Proof. Identical to proof of Lemma 4.3.7 with Lemma 6.4.5 replacing Lemma 4.3.2. □

Validity is defined identically to the previous section, i.e.

$$\Lambda^f \models P \ \stackrel{\text{def}}{=} \ \forall VS, H, I, \Delta_f \in close(\Lambda^f).\, VS, H, I \models_{\Delta_f} P$$

Theorem 6.4.9. OPEN *and* CLOSE *are valid, i.e.*

$$\Lambda^f \models \alpha(e; \overline{e}) \wedge e : C \Rightarrow \Lambda^f(\alpha, C)[e, \overline{e}/x, \overline{x}]$$
$$\Lambda^f \models \Lambda^f(\alpha, C)[e, \overline{e}/x, \overline{x}] \wedge e : C \Rightarrow \alpha(e; \overline{e})$$

where $(\alpha, C) \in dom(\Lambda^f)$.

Proof. Direct from definition of validity and the definition of the fixed point. □

Theorem 6.4.10. WIDEN *and* NARROW *are valid, i.e.*

$$\Lambda^f \models \alpha(e; \overline{e}) \Rightarrow \exists X.\alpha(e; \overline{e}, X)$$
$$\Lambda^f \models \alpha(e; \overline{e}, e') \Rightarrow \alpha(e; \overline{e}),$$

Proof. Direct from the definition of validity, Lemma 6.4.2 and Lemma 6.4.6. □

We are now in a position to define the semantics for our reasoning system. We write $\Lambda^f; \Gamma \models \{P\}C\{Q\}$ to mean that if every specification in Γ is true of a method environment, and every abstract predicate family in Λ^f is true of a predicate family environment, then so is $\{P\}C\{Q\}$, i.e.

$$\Lambda^f; \Gamma \models \{P\}C\{Q\} \stackrel{\text{def}}{=} \forall \Delta_f \in close(\Lambda^f).\ (\Delta_f \models \Gamma) \Rightarrow \ \Delta_f \models \{P\}C\{Q\}$$

Given this definition we can show that the two new rules for abstract predicate families are sound.

Theorem 6.4.11. *Abstract weakening is sound.*

Proof. Direct consequence of the definition of judgements and Lemma 6.4.7. □

Theorem 6.4.12. *Abstract elimination is sound.*

Proof. Follows from Lemmas 6.4.5 and 6.4.8 □

6.5 Reverification for inherited methods

We conclude this chapter by briefly discussing the issues of reverifying inherited methods. We will not give details of these ideas as proper consideration would require `super` calls to methods (calling a parent's version of a method), which neither our operational semantics or programming logic supports.

In the development given so far we must verify every method of every class, even if the method is inherited. This is due to the method bodies being verified with respect to the current class's type:

L-DMETHOD
$$\frac{\Lambda^f;\Gamma \vdash \{P \wedge \texttt{this} : C\}\overline{s}\{Q\}}{\Lambda^f;\Gamma \Vdash \{P\}C.m(\overline{x})\{Q\}} \quad \text{where } mbody(C,m) = (\overline{x},\overline{s})$$

Prior to this chapter we do not actually require $\texttt{this} : C$ in method verification, so we could alter the proof rules and hence allow methods to be inherited without reverification. However, the examples in this chapter depend completely on it. This assertion is required to OPEN and CLOSE the predicate families. Without it, the proofs simply fail.

We need to restrict changes to abstract predicate family definitions to allow the reuse of proofs in subtypes. We begin by presenting a generalized notion with respect to an abstract relation, and then provide two concrete relations.

We assume a relation, $\prec$, that constrains the changes of abstract predicate families between types. Rather than proving each method with the defining class's precise type as in L-DMETHOD, we will consider proving for all types that are related:

$$\forall \mathcal{D} \prec C.\ \ \{P \wedge \texttt{this} : \mathcal{D}\}\_\{Q\} \tag{6.3}$$

We use $\mathcal{C}, \mathcal{D}$ for variables over class names.

In particular the relation must be transitive and reflexive. Reflexive means (6.3) contains the original proof required for the method call. Transitive means that if we have a proof for C of the form (6.3), and D is related to C, then we also have a proof of the form (6.3) for D. The programmer must prove that all classes related by $\prec$ can inherit this method.

The subtype must prove it is contained in the relation, to inherit a method without reverifying the method body.

We briefly consider two possible relations: (1) if the predicate family definitions are unchanged in the subtype; and (2) if the predicate family definitions are "proper" extensions.

(1) If we do not change any of the definitions a method uses then it should be valid to inherit the method. We define the relation $D \prec_{\{\alpha_1,\ldots,\alpha_n\}} C$ as holding only if all the predicates $\alpha_1, \ldots, \alpha_n$ are defined the same for both classes, that is

$$\Lambda^f \vdash D \prec_{\{\alpha_1,\ldots\alpha_n\}} C \quad \iff \quad \forall \alpha_i \in \{\alpha_1, \ldots, \alpha_n\}.\ \Lambda^f(\alpha_i, C) = \Lambda^f(\alpha_i, D)$$

This relation is useful when a method uses no predicate families, because $\prec_{\{\}}$ is the universal relation. This allows any subtype to inherit that method. This is useful for the template method pattern [40]. This is where a class defines the outline or skeleton of an algorithm, leaving the details of the specific implementations to the subclasses. The method specifying the algorithm uses methods to manipulate the underlying data, so it does not depend on the representation, and hence the subclass can change it. Consider adding the following method into the `Cell` class.

```
Object swap(Object new) {
   Object old;
   old = this.get();
```

```
    this.set(new);
    return old;
}
```

It is always valid to inherit this method into a subtype of `Cell` as it does not depend on the cell's representation. In particular, in a subclass, `twiceCell`, whose representation is:

$$Value(\texttt{this}; x) \stackrel{\text{def}}{=} \texttt{this.contents} \mapsto x \ * \ \texttt{this.contents2} \mapsto x$$

It is perfectly valid to inherit the `swap` method, but not the `set` or `get` method.

The first relation does not allow extension without reverifying methods. We have made some progress considering the following relation:

$$\Lambda^f \vdash D \prec_{\{\alpha_1, \ldots \alpha_n\}} C \quad \Longleftrightarrow$$
$$\exists P.\ \Lambda^f(\alpha_i, C) = \lambda(x; \overline{x}).P_C \ \ \wedge \ \ \Lambda^f(\alpha_i, D) = \lambda(x; \overline{x}, \overline{y}).P_D \ \wedge$$
$$P_C \ * \ P = P_D \ \ \wedge \ \ FV(P) \subseteq \{x, \overline{y}\}$$

This relation expresses that a class's definition contains its parent's definition, $P_C \ * \ P = P_D$, and that the extra specification does not depend on the free variables of the parent's definition, $FV(P) \subseteq \{x, \overline{y}\}$. The second condition is required so the parent cannot invalidate the subclass's predicate. For example in the `twiceCell` example its definition contained the `Cell`'s definition, but treating it like the `Cell`'s definition would break the property that `contents` and `contents2` have the same value. Using this relation one must alter the definition of OPEN and CLOSE to account for the extra state that the subtype has introduced.

These ideas are related to the stacks of specifications of Fähndrich and DeLine [36]. Their work allows extensions in a similar form to this relation. To properly express modularity, they make calls to the superclass's methods, which our semantics does not support.

Abstract predicate families help in allowing classes to change their implementation, and we leave open how best to tame them to reduce the verification burden.

7

Conclusion

This thesis has presented a logic, based on the local reasoning approach of O'Hearn, Reynolds, and Yang [65], to reason about Java. We have demonstrated several extensions to separation logic designed to allow reasoning about languages with class-based modularity. In particular, we have extended separation logic with three key features: abstract predicates; read-only points-to predicates; and abstract predicate families.

The first, abstract predicates, allows reasoning about encapsulation. They are relevant not only to class-based languages, but also to any language with dynamically hidden encapsulated state, such as abstract datatypes. They can be used to prove a `struct` in C is used as an abstract datatype, by disallowing code that depends on the `struct`'s layout. Abstract predicates do not just allow data encapsulation, but also protocol specification. They can be used to enforce a call sequence of methods in a similar fashion to Typestates [36].

The second extension, the read-only points-to predicate, allows datastructures read access to state, which is essential for many common idioms. Although important for datatypes and classes, the read-only predicate is essential for many concurrent algorithms [13, 14].

The third and final extension is the notion of an abstract predicate family. This concept allows abstract predicates to have multiple definitions indexed by class. This allows each class to define how it represents a datatype. We have shown the utility of this concept by reasoning about the Visitor design pattern.

7.1 Open questions

We conclude this thesis by outlining a series of open issues:

Integrating hypothetical frame rule with abstract predicates There are currently two ways to deal with information hiding in separation logic: abstract predicates and the hypothetical frame rule. These two approaches have different strengths and weaknesses. We conjecture the two approaches could successfully be combined in a logic, as they appear to be orthogonal concepts. However, the semantic development in this thesis is very different to the work on the hypothetical frame rule [66], so further work is required to prove this conjecture.

Higher-order separation logic and abstract predicates Birkedal and Torp-Smith [10] have shown how proofs similar to those expressed in Chapter 4 can be expressed in higher-order separation logic [8]. However, they do not encode the rules in this thesis, but simply show similarly

structured proofs. It would prove useful to encode the rules given in this thesis directly into higher-order separation logic in order to show that existential predicates do indeed capture all of the required ideas. Birkedal and Torp-Smith also remark that abstract predicate families can be encoded in higher-order separation logic, but provide no details of this encoding. A detailed exploration of possible encodings of abstract predicate families into higher order separation logic should be conducted.

Higher-order functions and code pointers The semantics in this thesis has avoided the view that objects use code pointers to represent dynamic dispatch on methods. Instead we have relied on the class hierarchy and behavioural subtyping. It would be interesting to explore the use of abstract predicate families to reason about code pointers or higher-order functions.

Concurrency O'Hearn [63] has shown how to reason about statically scoped concurrency primitives in separation logic. His rules use the same method of information hiding as the hypothetical frame rule: this style of reasoning cannot deal with Java's concurrency primitives as these are dynamically scoped. We are currently exploring integrating the information hiding aspects of abstract predicates with O'Hearn's concurrency rules to allow reasoning about Java's concurrency primitives.

Nomenclature

$*$ The separating conjunction, page 47

$-\!\!*$ The separating implication (magic wand), page 47

$\circledast_{=}$ The iterated separating conjunction, page 72

$\prec$ The subtype relation, page 22

$\prec_1$ The immediate subtype relation, page 22

$\prec_\theta$ The substitution relation on stacks, page 59

$[]$ An empty stack, page 28

α An abstract predicate name, page 64

Δ A semantic predicate environment, page 69

Δ_f A semantic predicate family environment, page 103

δ A class table, page 22

δ_{C} A constructor environment, page 22

δ_{F} A field environment, page 22

δ_{M} A method environment, page 22

η A method or constructor name, page 49

Γ A specification environment, page 48

γ A typing environment, page 24

Λ An abstract predicate environment, page 64

Λ^f An abstract predicate family environment, page 92

μ A method type, page 21

π An MJ program, page 20

τ A return or statement type, page 20

θ A substitution, page 50

ζ A return statement or skip, page 56

$\mathcal{A}$ The set of all abstract predicate names, page 69

BS A block scope, page 28

C, D A class name, page 20

cd A class definition, page 20

CF A closed frame, page 28

cnd A constructor definition, page 20

$config$ A configuration, page 28

e An expression, page 20

e An expression, page 44

EF An exception frame, page 28

f A field name, page 20

F A frame, page 28

fd A field definition, page 20

FS A frame stack, page 28

H A heap, page 45

H A heap, page 28

$\mathcal{H}$ The set of all heaps, page 45

I A ghost stack, page 46

m A method name, page 20

M The abstract permissions partial commutative semi-group, page 85

m An element of the abstract permissions semi-group M, page 85

md A method definition, page 20

md A method definition, page 44

MS A method scope, page 28

m_w The abstract write permission, page 86

o An object identifier, page 28

OF An open frame, page 28

$\boldsymbol{p}$ A countably infinite subset of $\mathbb{P}$, page 89

p A non-empty subset of $\mathbb{P}$, page 84

P, Q, R A separation logic formula, page 47

$\mathbb{P}$ The set of all permissions, page 84

pe A promotable expression, page 20

ret The return variable for specifications, page 49

s A statement, page 20

s A statement, page 44

v A value, page 28

VS A variable stack, page 28

x A program variable, page 20

X, Y, ... A ghost variable, page 46

Bibliography

[1] Martin Abadi and Luca Cardelli. *A theory of objects*. Springer, 1996. (Cited on pages 16, 17, and 62.)

[2] Martin Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *Proceedings of CAAP/FASE*, volume 1214 of *LNCS*, pages 682–696. Springer, 1997. (Cited on page 17.)

[3] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, volume 489 of *LNCS*, pages 60–90. Springer, 1991. (Cited on pages 12 and 13.)

[4] Krzysztof R. Apt. Ten years of Hoare's logic: A survey: Part I. *ACM TOPLAS*, 3(4):431–483, 1981. (Cited on pages 14 and 53.)

[5] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfrum Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. (Cited on page 17.)

[6] Michael Barnett, K. Rustan M. Leino, and Wolfrum Schulte. The Spec$^\sharp$ programming system: An overview. In *Proceedings of CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005. (Cited on page 17.)

[7] Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *Proceedings of ICFP*, volume 40(9) of *SIGPLAN Not.*, pages 280–293. ACM, 2005. (Cited on page 18.)

[8] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI hyperdoctrines and separation logic. In *Proceedings of ESOP*, volume 3444 of *LNCS*, pages 233–247. Springer, 2005. (Cited on pages 77 and 109.)

[9] Gavin M. Bierman, Matthew J. Parkinson, and Andrew M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2004. (Cited on pages 27 and 44.)

[10] Lars Birkedal and Noah Torp-Smith. Higher order separation logic and abstraction. Unpublished manuscript, 2005. (Cited on pages 77, 109, and 110.)

[11] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of POPL*, volume 39(1) of *SIGPLAN Not.*, pages 220–231. ACM, 2004. (Cited on page 43.)

[12] Richard Bornat. Proving pointer programs in Hoare logic. In *Proceedings of MPC*, volume 1837 of *LNCS*, pages 102–126. Springer, 2000. (Cited on page 15.)

[13] Richard Bornat. Variables as resources in separation logic. In *Proceedings of MFPS*, 2005. to appear in ENTCS. (Cited on pages 15, 48, and 109.)

[14] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, volume 40(1) of *SIGPLAN Not.*, pages 259–270. ACM, 2005. (Cited on pages 68, 79, 80, 84, 85, 88, and 109.)

[15] Chandrasekhar Boyapati, Barbara H. Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of POPL*, volume 38(1) of *SIGPLAN Not.*, pages 213–223. ACM, 2003. (Cited on page 14.)

[16] John Boyland. Checking interference with fractional permissions. In *Proceedings of SAS*, volume 2694 of *ACM*, pages 55–72. Springer, 2003. (Cited on page 88.)

[17] Kim B. Bruce. Some challenging typing issues in object-oriented languages. *ENTCS*, 82(7), 2003. (Cited on page 97.)

[18] Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the visitor pattern. In *Proceedings of MFPS*, 2005. to appear in ENTCS. (Cited on page 97.)

[19] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *ENTCS*, 80:73–89, 2003. (Cited on page 17.)

[20] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972. (Cited on page 15.)

[21] Cristiano Calcagno and Peter W. O'Hearn. On garbage and program logic. In *Proceeding of FOSSACS*, volume 2030 of *LNCS*, pages 137–151. Springer, 2001. (Cited on page 47.)

[22] David G. Clarke. *Object ownership and containment*. PhD thesis, University of New South Wales, Australia, 2001. (Cited on page 14.)

[23] David G. Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of OOPSLA*, volume 37(11) of *SIGPLAN Not.*, pages 292–310. ACM, 2002. (Cited on pages 14 and 15.)

[24] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of OOPSLA*, volume 33(10) of *SIGPLAN Not.*, pages 48–64. ACM, 1998. (Cited on page 14.)

[25] Pierre-Jacques Courtois, F. Heymans, and David L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971. (Cited on page 79.)

[26] Frank S. de Boer. A WP-calculus for OO. In *Proceeding of FOSSACS*, volume 1578 of *LNCS*, pages 135–149. Springer, 1999. (Cited on page 18.)

[27] Frank S. de Boer and Cees Pierik. How to cook a complete Hoare logic for your pet OO language. In *Proceedings of FMCO*, volume 3188 of *LNCS*, pages 111–133. Springer, 2003. (Cited on page 18.)

[28] Robert DeLine and Manuel Fähndrich. The Fugue protocol checker: Is your software baroque. Technical Report MSR-TR-2004-07, Microsoft Research, 2004. (Cited on page 18.)

[29] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq SRC, Palo Alto, CA, 1998. (Cited on pages 9 and 17.)

[30] Krishna K. Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of ICSE*, pages 258–267. IEEE Computer Society, 1996. (Cited on page 13.)

[31] Krishna K. Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. Technical Report 95-20c, Department of Computer Science, Iowa State University, March 1997. (Cited on page 13.)

[32] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 2005. To appear. (Cited on pages 14 and 17.)

[33] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 7(1):3–24, 1999. (Cited on pages 41 and 44.)

[34] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited, 2000. Unpublished. (Cited on page 41.)

[35] Jon Ellis and Linda Ho. JDBC 3.0 specification, 2001. `http://java.sun.com/products/jdbc/download.html`. (Cited on page 65.)

[36] Manuel Fähndrich and Robert DeLine. Typestates for objects. In *Proceedings of ECOOP*, volume 3086 of *LNCS*, pages 465–490. Springer, 2004. (Cited on pages 17, 77, 81, 107, and 109.)

[37] Robert B. Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Proceedings of OOPSLA*, volume 36(11) of *SIGPLAN Not.*, pages 1–15. ACM, 2001. (Cited on page 13.)

[38] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR-97-293, Rice University, 1997. Corrected June, 1999. (Cited on pages 19, 41, and 44.)

[39] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967. (Cited on page 10.)

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994. (Cited on pages 17, 91, 96, and 106.)

[41] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, second edition, 2000. (Cited on pages 19, 21, 24, 25, and 31.)

[42] Mark Grand. *Patterns in Java*. Wiley, second edition, 2002. (Cited on pages 16 and 65.)

[43] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer, 1993. (Cited on page 14.)

[44] C. A. R. Hoare. Assertions: a personal perspective. In *Software pioneers: contributions to software engineering*, pages 356–366. Springer, 2002. (Cited on page 9.)

[45] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. (Cited on pages 10 and 13.)

[46] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972. (Cited on page 12.)

[47] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001. (Cited on pages 19, 24, and 41.)

[48] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of POPL*, volume 36(3) of *SIGPLAN Not.*, pages 14–26. ACM, 2001. (Cited on pages 15, 43, 44, 46, and 47.)

[49] Bart Jacobs, K. Rustan M. Leino, and Wolfrum Schulte. Verification of multithreaded object-oriented programs with invariants. In *Proceedings of SAVCBS*, pages 2–9, November 2004. Published in Technical Report #04-09, Department of Computer Science, Iowa State University. (Cited on page 17.)

[50] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988. (Cited on page 73.)

[51] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004. (Cited on page 41.)

[52] John Lamping. Typing the specialization interface. In *Proceedings of OOPSLA*, volume 28(10) of *SIGPLAN Not.*, pages 201–214. ACM, 1993. (Cited on page 63.)

[53] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06v, Iowa State University, Department of Computer Science, May 2003. (Cited on page 17.)

[54] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of OOPSLA*, volume 33(10) of *SIGPLAN Not.*, pages 144–153. ACM, 1998. (Cited on pages 14 and 63.)

[55] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, 1994. (Cited on pages 12 and 13.)

[56] Ronald Middlekoop. A proof system for object oriented programming using separation logic. Master's thesis, Technische Universiteit Eindhoven, Eindhoven, the Netherlands, November 2003. (Cited on page 18.)

[57] Ronald Middlekoop, Kees Huizing, and Ruurd Kuiper. A separation logic proof system for a class-based language. In *Proceedings of LRPP*, 2004. (Cited on page 18.)

[58] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM TOPLAS*, 10(3):470–502, 1988. (Cited on page 77.)

[59] Joseph M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, pages 25–51. D. Reidel, 1982. (Cited on pages 14 and 18.)

[60] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002. PhD thesis, FernUniversität Hagen. (Cited on pages 14 and 17.)

[61] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. Technical Report 02-02a, Department of Computer Science, Iowa State University, October 2002. (Cited on pages 14 and 17.)

[62] Tobias Nipkow and David von Oheimb. $\text{Java}_{\text{light}}$ is type-safe—definitely. In *Proceedings of POPL*, pages 161–170. ACM, 1998. (Cited on page 41.)

[63] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *Proceedings of CONCUR*, volume 3170 of *LNCS*, pages 49–67. Springer, 2004. Invited Talk. (Cited on pages 68 and 110.)

[64] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999. (Cited on page 15.)

[65] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001. (Cited on pages 10, 43, 44, 49, 72, and 109.)

[66] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proceedings of POPL*, volume 39(1) of *SIGPLAN Not.*, pages 268–280. ACM, 2004. (Cited on pages 16, 63, 67, 74, 75, 76, and 109.)

[67] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, volume 40(1) of *SIGPLAN Not.*, pages 247–258. ACM, 2005. (Cited on pages 63 and 72.)

[68] David L. Parnas. The secret history of information hiding. In *Software Pioneers: Contributions to Software Engineering*, pages 399–411. Springer, 2002. (Cited on page 75.)

[69] Cees Pierik and Frank S. de Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In *Proceedings of FMOODS*, volume 2884 of *LNCS*, pages 64–78. Springer, 2003. (Cited on page 18.)

[70] Cees Pierik and Frank S. de Boer. Modularity and the rule of adaptation. In *Proceedings of AMAST*, volume 3116 of *LNCS*, pages 394–408. Springer, 2004. (Cited on page 18.)

[71] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In *Proceedings of ESOP*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999. (Cited on pages 17 and 60.)

[72] Arnd Poetzsch-Heffter and Peter Müller. Logical foundations for typed object-oriented languages. In *Proceedings of PROCOMET*, volume 125 of *IFIP Conference Proceedings*, pages 404–423. Chapman & Hall, 1998. (Cited on pages 13, 17, and 53.)

[73] Uday S. Reddy. Objects and classes in Algol-like languages. *Information and Computation*, 172(1):63–97, 2002. (Cited on page 77.)

[74] Bernhard Reus. Class-based versus object-based: A denotational comparison. In *Proceedings of AMAST*, volume 2422 of *LNCS*, pages 473–488. Springer, 2002. (Cited on page 17.)

[75] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, pages 55–74. IEEE Computer Society, 2002. (Cited on pages 43, 44, 49, 50, and 72.)

[76] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000. (Cited on pages 15, 43, 44, and 47.)

[77] John C. Reynolds. *The Craft of Programming*. Prentice-Hall International, London, 1981. (Cited on pages 14 and 53.)

[78] John C. Reynolds. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982. (Cited on page 77.)

[79] Matthew Smith and Sophia Drossopoulou. Cheaper reasoning with ownership types. In *Proceedings of IWACO*, 2003. Published in technical report UU-CS-2003-030, Institute of Information and Computing Sciences, Utrecht University. (Cited on page 14.)

[80] Raymie Stata. Modularity in the presence of subclassing. Technical Report 145, Digital Systems Research Center, Palo Alto, CA, April 1997. (Cited on page 63.)

[81] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986. (Cited on pages 17 and 77.)

[82] Donald Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, 1997. (Cited on page 41.)

[83] David von Oheimb. Hoare logic for mutual recursion and local variables. In *Proceedings of FSTTCS*, volume 1738 of *LNCS*, pages 168–180. Springer, 1999. (Cited on pages 50 and 60.)

[84] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In *Proceedings of FME*, volume 2391 of *LNCS*, pages 89–105. Springer, 2002. (Cited on page 17.)

[85] Glynn Winskel. *The Formal Semantics of Programming Languages: an introduction*. MIT press, 1993. (Cited on page 70.)

[86] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. (Cited on page 35.)

[87] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of PLDI*, volume 33(5) of *SIGPLAN Not.*, pages 249–257. ACM, 1998. (Cited on page 9.)

[88] Hongseok Yang. *Local reasoning for stateful programs*. PhD thesis, University of Illinois, July 2001. (Cited on pages 43, 56, and 57.)