Mars
An Imperative/Declarative Higher-Order Programming Language With Automatic Destructive Update

Matthew Giuca

Submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy

July 2014

Department of Computing and Information Systems
The University of Melbourne
Abstract

For years, we have enjoyed the robustness of programming without side-effects in languages such as Haskell and Mercury. Yet in order to use languages with true referential transparency, we seem to have to give up basic imperative constructs that we have become accustomed to: looping constructs, variable update, destructive update of arrays, and so on. In this thesis, we explore the design of programming languages which combine the benefits of referential transparency with imperative-style programming.

First, we present a framework for classifying programming languages according to the benefits of pure programming. Our definition applies to a wider range of languages than common terms such as “declarative” and “pure,” capturing the specific benefit of prohibiting global side-effects, without ruling out imperative programming languages.

Second, we present the design and implementation for a new programming language, Mars, which allows the programmer to write imperative-style code using Python syntax, yet has the benefits of a language with referential transparency. The design philosophy behind the language, and its future directions, are discussed.

Third, we note the tendency for imperative programs to use array update operations, which are very slow in a naïve implementation of any language such as Mars. We explore static analyses for automatically converting slow array copying code into fast destructive update instructions, and present the design and implementation of an optimiser for Mars, which improves on previous work by precisely handling higher-order functions, including those that both accept and return functions. The sum of this work is an efficient imperative language with the demonstrable software engineering benefit of non-interfering functions.
Declaration

This is to certify that:

(i) the thesis comprises only my original work towards the Ph.D. except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is fewer than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Matthew Laurence Giuca
Department of Computing and Information Systems
The University of Melbourne
July 2014
Preface

Drafts of Chapter 3 were written in collaboration with Dr. Peter Schachte (my Ph.D. supervisor), with the intention of publishing as a separate conference paper. The text was originally my own work, and after our collaboration, the chapter was edited heavily by myself.
Acknowledgements

Six years is a non-trivial amount of time to spend on any project, and so I have a long list of people to thank for their support and assistance.

First, and foremost, I wish to thank my supervisor, Dr. Peter Schachte, for his support throughout my candidature. From the weekly insightful (and often lengthy) discussions in his office, to an ongoing email support line long after I officially left the university, Peter has always been a source of good ideas, helpful advice, and encouragement.

I would also like to thank my advisory committee, Dr. Harald Søndergaard and Dr. Bernard Pope: in particular, Harald for his detailed feedback and help with Chapter 1, and Bernie for his many helpful comments and suggestions on earlier drafts of Chapter 3. I also thank the other research staff in what was the Department of Computer Science and Software Engineering during my time there, for their attendance at my talks and ongoing interest in my research. In particular, Dr. Lee Naish deserves credit for proof-reading my thesis, and pointing out the polymorphic recursion problem discussed in Section 9.6.3. I would also like to thank my two examiners, who provided many pages of detailed notes on the thesis, with thoughtful comments and suggestions.

I thank my parents, Anne and Vince, and my brothers, David and Dominic, for their support. Of course, I must also thank my amazing partner, Shanika Kuruppu, who started this journey with me, and who encouraged me and waited patiently while I worked through the weekends. I would not have completed this work without Shani’s continual love and support.

Finally, I would like to thank the Australian Department of Education for funding the first three and a half years of my candidature.
Contents

List of Figures ix

1 Introduction 1
  1.1 A purely imperative programming language .................. 2
  1.2 Automatic destructive update ................................ 4
  1.3 Contributions .............................................. 7
  1.4 Conventions in this thesis ................................... 8
  1.5 Terminology .................................................. 9
  1.6 Preliminaries ............................................... 10
  1.7 Outline of the thesis ........................................ 17

2 Survey of hybrid imperative/declarative languages 19
  2.1 Introduction ................................................ 19
  2.2 Imperative languages with functional features ............... 23
  2.3 Imperative languages with purity constraints ................ 24
  2.4 Declarative languages with imperative constructs .......... 26
  2.5 Languages with explicit effect annotations .................. 31
  2.6 Conclusion .................................................. 33

3 Interface integrity 35
  3.1 Introduction ................................................ 35
  3.2 Interface integrity ........................................... 38
  3.3 Interface integrity and language features .................... 43
  3.4 Comparison to related concepts ................................ 52
  3.5 Language construction ....................................... 53
  3.6 Conclusion .................................................. 59

4 Mars: An imperative/declarative language 61
4.1 Introduction .................................................. 61
4.2 Overview ....................................................... 62
4.3 Standard libraries ........................................... 74
4.4 Example ......................................................... 75
4.5 Conclusion ...................................................... 77

5 Code representation for analysis .......................... 79
5.1 Introduction .................................................... 79
5.2 Converting Mars to MIR .................................... 80
5.3 Converting other languages to MIR ....................... 89
5.4 The implementation of MIR ................................. 93
5.5 Abstract syntax ............................................... 95
5.6 Concrete value denotational semantics ................. 96
5.7 Concrete memory denotational semantics .............. 103
5.8 Conclusion ...................................................... 108

6 Abstract interpretation ....................................... 111
6.1 Introduction .................................................... 111
6.2 The abstract domain ....................................... 113
6.3 Abstraction and concretisation .......................... 116
6.4 Abstract semantics ......................................... 119
6.5 Making use of the semantics ............................... 122
6.6 Condensing semantics and context-insensitive analysis .. 125
6.7 Efficient bottom-up analysis .............................. 128
6.8 Conclusion ...................................................... 131

7 Existing sharing and reuse analyses ...................... 133
7.1 Introduction .................................................... 133
7.2 Runtime reference counting ................................ 134
7.3 Type-based approaches ..................................... 134
7.4 Compile-time garbage collection ......................... 140
7.5 Dynamic memory reuse flags ....................... 151
7.6 Aliasing analysis for impure and unsafe languages .... 152
7.7 Dynamic aliasing analysis ................................. 155
7.8 Static pointer nullification ............................... 156
7.9 Conclusion ...................................................... 157

8 First-order uniqueness inference .......................... 159
# CONTENTS

8.1 Introduction ...................................................... 159  
8.2 Motivating examples ............................................ 160  
8.3 Abstract memory denotational semantics .................. 174  
8.4 Comparison to Mercury’s CTGC system ................. 189  
8.5 Conclusion ....................................................... 190  

9 Higher-order uniqueness inference 191  
9.1 Introduction ..................................................... 191  
9.2 Why analyse higher-order? ................................... 192  
9.3 Overview ......................................................... 196  
9.4 The upward higher-order problem ......................... 199  
9.5 The non-applicative downward higher-order problem ... 205  
9.6 The applicative downward higher-order problem ........ 208  
9.7 Abstract memory denotational semantics ............... 218  
9.8 Algebraic analysis .............................................. 227  
9.9 Analysing non-Mars language features ................. 236  
9.10 Conclusion ....................................................... 239  

10 Destructive update optimisation 241  
10.1 Introduction .................................................... 241  
10.2 Live-variable analysis ......................................... 242  
10.3 Call mode selection .......................................... 245  
10.4 Dynamic call mode vectors .................................. 253  
10.5 Multiple specialisation ........................................ 257  
10.6 Higher-order call modes ..................................... 261  
10.7 Structure reuse ................................................. 262  
10.8 Future work ..................................................... 270  
10.9 Conclusion ....................................................... 271  

11 Conclusion 273  
11.1 A pure imperative programming language ............. 273  
11.2 Automatic destructive update .............................. 275  
11.3 Benchmarks ...................................................... 276  
11.4 Future work ...................................................... 279  
11.5 What we have built ............................................ 280
List of Figures
1.1

Linked list of boxed numbers, showing the meaning of various terms . . .

10

1.2

Hasse diagram for the partial order ℘({0, 1}) . . . . . . . . . . . . . . . . .

14

3.1

Example function declaration and call syntax . . . . . . . . . . . . . . . . .

40

3.2

Effective interface (pure) . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

41

3.3

Effective interface (with ref) . . . . . . . . . . . . . . . . . . . . . . . . . . .

42

3.4

Unintended aliasing of the player and rocket position . . . . . . . . . . . .

47

3.5

The denotational semantics for L0 . . . . . . . . . . . . . . . . . . . . . . . .

54

3.6

Additions to the denotational semantics for L1 . . . . . . . . . . . . . . . .

56

3.7

Additions to the denotational semantics for L2 . . . . . . . . . . . . . . . .

57

3.8

Additions to the denotational semantics for L3 . . . . . . . . . . . . . . . .

58

6.1

Hasse diagram for the domain {(0), (−), (+), (±)} . . . . . . . . . . . . . 111

6.2

Hasse diagram for the domain ℘({Nil, Two, Three}) . . . . . . . . . . . . . 115

6.3

Call graph with non-singleton SCCs marked in grey . . . . . . . . . . . . . 130

8.1

Internal sharing graph for Example 8.2a after each line of analysis . . . . . 162

8.2

Internal sharing graph for if_then_else after selected lines of analysis . . . 164

8.3

Internal sharing graph for pair_const after each line of analysis . . . . . . . 166

8.4

Internal sharing graph for array_const after each line of analysis . . . . . . 167

8.5

Internal sharing graph for component_alias after selected lines of analysis

8.6

Internal sharing graph for build_either after selected lines of analysis . . . . 169

8.7

Internal sharing graph for append after selected lines of analysis (first iteration) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 171

8.8

Internal sharing graph for append (second iteration) . . . . . . . . . . . . . 171

8.9

Internal sharing graph for my_array_add after selected lines of analysis . . 173

168

8.10 Internal sharing graph for modify_a after selected lines of analysis . . . . . 174
8.11 Examples of concrete states, showing which abstract names (in italics) name
which concrete cells (via dotted arrows). Constructor names are omitted. . 175
ix


8.12 Hasse diagram for the domain Sharing with Name = \{a, b, c\}  .......... 178

9.1 Sharing graph for example call to const and its result ................. 201
9.2 Internal sharing graph for problem after selected lines of analysis ... 203
9.3 Internal sharing graph for apply after each line of analysis .......... 212
9.4 Internal sharing graph for compose due to both lines, and the final state . 216

10.1 The use and def sets ................................................................. 244

11.1 Mandelbrot set benchmark ......................................................... 277
11.2 Hash table insertion benchmark (non-destructive version) ............. 278
11.3 Hash table insertion benchmark (destructive version) .................. 279
Chapter 1

Introduction

Like most programmers today, I learned to program by feeding the computer a sequence of instructions. To this day, I am most comfortable communicating with the computer in imperative languages: assigning and re-assigning variables, and defining loops and conditionals that update the computation state. When programming “in the small,” I like to think about what actions the machine will take, and in what order.

However, early in my undergraduate degree, I was introduced to Haskell, a profoundly different programming experience. In Haskell, a declarative programming language, one does not tell the computer how to do its job, one merely writes a set of mathematical function definitions and lets the compiler figure it out. In the small, I often found it (and still sometimes find it) rather frustrating, having to use recursion or figure out how to use one of the many arcane higher-order functions to simulate a loop, for instance. But “in the large,” Haskell was a deep and liberating experience.

The core idea of functional programming is that functions are values. But more importantly than that, I discovered, was that functions produce values — and nothing more. In Haskell, there is no need to worry about what else might happen when you call a function. There is no need to worry that, after the function returns, global variables might have changed, that the value you passed as an argument may have been modified, that text may have been printed to the console, or any other unexpected side-effect may have occurred. There is no need to worry about what order to call functions in, since a call to a function with the same arguments has the same value no matter when it is called. In short, if $f$ is a function and $x$ is a variable, $f\ x$ represents nothing more than a value.

And so I was torn: I found that programming “in the small” is a step-wise process: the benefits of being able to modify local variables outweigh the potential pitfalls. But programming “in the large” is too cumbersome when literally any function is allowed to do anything to the global computation state, and I very much preferred the neat semantics
of a declarative language.

This thesis is about the journey my supervisor and I undertook to design a programming language that fits this philosophy. The seemingly simple task of specifying a language that allows local state changes, but prohibits global side-effects was complicated by two matters: firstly, the theoretical matter of coming up with a definition of “declarative programming” that was general enough to apply to all language disciplines including imperative, yet strict enough to rule out all harmful effects, and secondly, the practical matter of incorporating an optimisation that would allow the efficient data structure updates that imperative languages typically afford programmers, within the bounds of pure programming.

1.1 A purely imperative programming language

To put the above dilemma into context, consider Example 1.1, a function for computing the median value of a list (array) in Python, a popular imperative programming language. This function works by first sorting the list, then choosing the middle value. The algorithm correctly gives the median, but there is something seriously wrong with it: it has a side-effect. Calling `vals.sort()` physically re-arranges the order of the elements in the list. This is exactly what we want inside the median function, but it is unlikely to be what the person who called median was intending. Worse, they may not be aware of the side-effect (after all, the name median hardly indicates that the list will be sorted). Worse still, whoever called median may have received the list from another piece of code, so the unintended side-effect might mess up a list in a function that was not even aware of the call to median.

Example 1.1 Computing the median value of a list in Python

```python
def median(vals):
    vals.sort()
    return vals[len(vals)//2]
```

Python, like many imperative languages, provides no way to specify that an argument will not be mutated. Even in C or C++, the `const` keyword does not provide this guarantee for pointers within a data structure.

Maybe this is not such a big deal: after all, it is clearly a bug, and bugs happen all the time. The correct solution is not to use the `sort` method at all, but instead to use a different

---

1 The result will be slightly off for even-length lists, choosing the higher of the two middle values, and will also fail for empty lists, but this does not matter for the purpose of our example.
1.1. A PURELY IMPERATIVE PROGRAMMING LANGUAGE

built-in function, sorted, which returns a new, sorted list, without modifying its input:

\[
\text{return sorted(vals)[len(vals)//2]}
\]

But that only resolves one bug. As language designers, we strive to solve entire classes of bugs at a time. The fact remains that every time you pass a list to a function in Python (or any other imperative language), it might be different afterwards. You may not even have a choice as to whether the list is mutated (imagine if Python had sort but not sorted). Compare this to the same function in Haskell, shown in Example 1.2.

**Example 1.2** Computing the median value of a list in Haskell

\[
\text{median } vals = (\text{sort } vals) !! (\text{length } vals \div 2)
\]

The Haskell code looks very much like the “fixed” Python code (sort in Haskell is the same as sorted in Python). The key difference is that in Haskell, it is impossible to make the mistake of mutating \(vals\) — a Haskell function cannot modify its inputs, only produce new outputs. Therefore, the caller of \(\text{median}\), and its caller, and so on, can be safe in the knowledge that the list will not be modified.

This should, for now, be sufficient motivation for a language with “pure semantics” such as Haskell (we further motivate this concept in Chapter 3). However, Haskell lacks the expressiveness of imperative languages, in particular, for writing iterative algorithms.

**Example 1.3** Factorial function in Haskell (simple and tail-recursive versions)

\[
\begin{align*}
\text{factorial} \ 0 &= 1 \\
\text{factorial} \ n &= n \times \text{factorial} \ (n - 1) \\
\text{factorial}' \ n &= \text{factorial}' \ 1 \ n \\
\text{where} \\
\text{factorial}' \ acc \ 0 &= acc \\
\text{factorial}' \ acc \ n &= \text{seq} \ acc \ $ \\
&\quad \text{factorial}' \ (acc \times n) \ (n - 1)
\end{align*}
\]

Example 1.3 shows two implementations of factorial in Haskell. On the left is a straightforward recursive version, nearly identical to the mathematical definition of factorial. It is a testament to the brevity of Haskell and its ties to pure mathematics, but this version is famously non-tail-recursive. This means that that the implementation must expand out an expression \(n\) terms deep before it can begin to multiply the numbers together. Such an algorithm consumes \(O(n)\) stack space, which can cause a stack overflow for large input values. The solution is to use accumulator recursion, as shown in the version on the right. This version keeps an extra variable, \(acc\), which is initially 1. It multiplies \(acc\) by each number before recursing, and since the recursive call is the last thing to be done, the algorithm is said to be tail recursive, and therefore can be executed in \(O(1)\) space.

While we ended up with an efficient solution, the result is significantly less readable.
CHAPTER 1. INTRODUCTION

Not only does it require two functions (one to pass an accumulator, and the other to start the process), but it also requires imperative thinking to understand. The version on the left can be understood declaratively: “the factorial of \( n \) is \( n \) times the factorial of \( n - 1 \).” The version on the right can really only be understood imperatively: “to calculate the factorial of \( n \), start with \( acc = 1 \), then set \( acc \) to \( acc \times n \), subtract 1 from \( n \), and repeat until \( n \) is 0.” Once a programmer is thinking this way, he or she may as well be writing an imperative algorithm. The programmer will be fighting against the programming language if it forbids iteration.\(^2\)

This is the reason we designed and implemented the Mars programming language. Mars is a simple high-level language designed from the ground up to be both imperative and declarative. Its syntax is based on that of Python, allowing local variable re-assignment, stateful conditionals and iteration. However, its function semantics are based on those of Haskell, forbidding side-effects across function boundaries. We believe this language offers the best of both worlds, and much of the thesis is dedicated to this argument. The factorial function in Mars is shown in Example 1.4, in both recursive (left) and iterative (right) styles. The iterative version on the right performs exactly the same algorithm as the tail-recursive Haskell version of Example 1.3, but it is expressed clearly as a sequence of steps.

\begin{example}
\textbf{Factorial function in Mars (recursive and iterative versions)}
\begin{verbatim}
def factorial(n :: Num) :: Num:
    if n == 0:
        return 1
    else:
        return n \times factorial(n - 1)
def factorial(n :: Num) :: Num:
    acc = 1
    while n \neq 0:
        acc = acc \times n
        n = n - 1
    return acc
\end{verbatim}
\end{example}

Mars is by no means an industrial-strength language, but it has a fully functioning implementation, and many high-level features, including a parametric Hindley-Milner type system [40, 71], functions as first-class values, algebraic data types and pattern matching. Chapter 4 is devoted to describing the language.

1.2 Automatic destructive update

An important goal for Mars is to allow algorithms to be implemented in the imperative-style, and this entails efficient operations on arrays and array-based data structures such as hash tables. As we shall see in this section, this is not something that is easily achieved\(^2\)

\(^2\)There are various ways to define iterative procedures in Haskell and other functional programming languages, but none are as simple as an imperative language. We discuss these possibilities in Chapter 2.
in a language with declarative semantics such as ours.

Consider Example 1.5, a typical imperative-style algorithm in Python, which takes two lists (arrays) and adds each corresponding pair of elements together, producing a new list. This can be accomplished with map, but then a similar algorithm would be needed to implement map.

**Example 1.5** Element-wise list sum in Python

```python
def add_lists(xs, ys):
    zs = []
    for i in range(min(len(xs), len(ys))):
        zs.append(xs[i] + ys[i])
    return zs
```

This algorithm causes no side-effects itself (it does not modify either of the input arrays `xs` or `ys`), but it does rely on side-effects for efficiency — the call to `zs.append` mutates the array `zs` on each loop iteration. As each call to `append` costs amortised $O(1)$ time, the whole algorithm runs in $O(n)$ time. In a language without side-effects, we would have to copy the whole array on each iteration, so each call to `append` would cost $O(n)$ time, and the whole algorithm would run in a generally unacceptable $O(n^2)$ time. In addition, such a language would have to waste a lot of time allocating and garbage collecting all of the intermediate arrays. Indeed, this is one of the most common arguments against declarative languages: side-effects are required for efficiency.

To demonstrate the shift in semantics, consider the same algorithm implemented in our language, Mars, shown in Example 1.6. While the variable `zs` can be re-assigned, there is no way to change the value of the array that it refers to. So, we call the built-in function `array_add`, which allocates a new array with all of the elements of `zs` and a new element at the end. The new array is assigned to the variable `zs`, so the old array can immediately be garbage collected.

**Example 1.6** Element-wise array sum in Mars

```bash
def add_arrays(xs :: Array(Num), ys :: Array(Num)) :: Array(Num):
    zs = []
    i = 0
    while i < min(array_length(xs), array_length(ys)):
        zs = array_add(zs, array_ref(xs, i) + array_ref(ys, i))
        i = i + 1
    return zs
```

Being forced to re-assign the variable `zs`, rather than have it implicitly updated by `array_add` is an important part of the design of Mars. But actually having to allocate
memory and copy array elements on each iteration is madness! In this particular case, and many others, we know for sure that nobody is going to look at the old value for $zs$ ever again, because we created it in the first place — it will be garbage collected as soon as possible. In this case, it would be safe to destructively update $zs$ on each iteration, but Mars does not allow this because it is not safe in general.

A solution to this problem is automatic destructive update (a form of compile-time garbage collection), a technique in which the compiler can, in certain situations, prove that the input to an allocate-and-copy operation will definitely become eligible for garbage collection after that operation, and emit code that destructively updates the original memory cell, without changing the semantics of the program. Example 1.7 shows a possible optimised version of the `array_add` function, in which the compiler has automatically replaced the call to `array_add` with a special destructive array operation, `array_add_d`, that is not directly available to Mars programmers. This version, like the Python version, runs in $O(n)$ time.

**Example 1.7 Optimised version of Example 1.6 with destructive update**

```python
def add_arrays(xs :: Array(Num), ys :: Array(Num)) :: Array(Num):
    zs = []
    i = 0
    while i < min(array_length(xs), array_length(ys)):
        array_add_d(zs, array_ref(xs, i) + array_ref(ys, i))
        i = i + 1
    return zs
```

Because it is automated, this technique can even offer performance improvements over hand-written imperative code. Consider again the median function in Python from Example 1.1. We fixed the destructive version by calling `sorted` (non-destructive) rather than `sort`. However, note that the "fixed" version is actually slower, because `sorted` allocates new memory whereas `sort` operates in-place. The allocation is necessary for correctness, because we do not want to disturb somebody else’s array, but what if the code that called `median` is not going to use the array again? Then in that special case, it would actually be valid, and more efficient, to use the original, destructive version of `median`.

**Example 1.8 Computing the median value of a list in Mars**

```python
def median(vals :: Array(Num)) :: Array(Num):
    return array_ref(sort(vals), array_length(vals) // 2)
```

The Mars version of `median`, shown in Example 1.8 is almost identical to the non-destructive Python version or the Haskell version. It does not have any side-effects, but because the Mars compiler performs automatic destructive update, the compiler may
1.3. CONTRIBUTIONS

produce a special version of median called $\text{median}_d$ that calls a special version of sort called $\text{sort}_d$ that sorts the input array in-place. The compiler automatically changes calls to median into $\text{median}_d$ only if the array is definitely not aliased, and will definitely not be used again. Mars will avoid making a copy of the array if it is safe to mutate the existing one, and only copy it if necessary.

This reasoning can be helpful in fairly common cases. For example, in most modern programming languages, strings are immutable, for the precise reasons we argue in this thesis: it is safer when there is no possibility that passing a string to a function could result in it being mutated. But this is also a source of inefficiency, because strings need to be copied in order to be modified. In Mars, strings are immutable like all other objects, but as with other arrays, the compiler is free to destructively update strings where it is safe to do so. The result is the best of both worlds: objects have immutable value semantics, but can be efficiently updated in many cases.

The concept of compile-time garbage collection and automatic destructive update is decades old [18]. In this thesis, we survey existing techniques, and describe in detail the system implemented in the Mars compiler, which is the most precise we have seen. In particular, it is novel that our aliasing analysis system precisely handles higher-order functions.

1.3 Contributions

This thesis makes three primary contributions to the fields of language design and static analysis.

The first contribution is the concept of interface integrity, a scheme that classifies programs and programming languages that satisfy a certain notion of “purity.” Our definition is not only more widely applicable than existing definitions (being applicable to imperative and logic languages as well as functional languages), but is also directly motivated by software engineering concerns; namely, the property that encourages more modular programming.

The second contribution is the design and implementation of a new programming language, Mars. It was important to complement the theory of interface integrity with a full programming language to demonstrate the viability of using a fully imperative language with interface integrity. It was also important to have a complete language to apply our static analysis to, instead of a toy language missing a lot of features. Mars is a complete implementation of a simple higher-order imperative/declarative hybrid

---

3Python, Java and .NET, for example.
language with a powerful static analysis, available to the public as free software\textsuperscript{4} and usable as the basis for further research into language design or static analysis.

The third contribution is a highly precise aliasing analysis and automatic destructive update optimisation for pure, eager programming languages. The aliasing analysis presented in this thesis is context insensitive, interprocedural, understands structure information, and produces precise results for higher-order functions, parameterised by the function parameters. This last feature, support for higher-order programs, is novel in an analysis as precise as ours.

1.4 Conventions in this thesis

Throughout this thesis, we adopt several typographical conventions in text, algorithms and source code examples.

Program identifiers are formatted in both text and source code depending on the kind of identifier. Type names and data constructor names are formatted in a serif font (e.g., List, Cons). Function names are formatted in a sans-serif font (e.g., map). Parameter and variable names are formatted in italics (e.g., total), and usually with a single letter. In some languages, such as Haskell, there is no syntactic or semantic distinction between functions and variables, but we typographically distinguish them anyway, for consistency. Keywords are formatted in bold (e.g., while).

We adopt the same conventions for the mathematical notations used to describe algorithms and source code examples from various languages. Algorithms and source code examples are presented in a style that resembles Python’s syntax (blocks begin with a colon and are indented; a block ends when the indentation does), as this is the syntax used in our programming language, Mars, although when source code is presented in another language, we follow the syntax conventions of that language. Regardless of the language, source code is embellished with minor typographic niceties that do not alter the meaning of the code (for example, in Haskell code, we write “λ” instead of “\" “×” instead of “∗,” “→” instead of “→,” and “\_0” instead of “\_0”).

When it is appropriate to compare two code examples side-by-side, we will place them in a single example box, as shown in Example 1.9. When referring to the individual examples, the code on the left will be given the suffix “a,” and the code on the right, the suffix “b.” For example, plus is in Example 1.9a, and times is in Example 1.9b. Examples have line numbers only when we need to mention specific lines in the prose.

\textsuperscript{4}Mars is licensed under the GNU General Public License version 3, and can be downloaded from http://mars-lang.appspot.com.
1.5. TERMINOLOGY

We introduce terminology for dealing with memory allocation and studying the liveness of both variables and memory-allocated objects. In an imperative program, a variable is live at a particular program point if it may be statically read at some subsequent program point before being written; a variable that is not live is dead. We consider the computer to be an abstract machine, and do not usually concern ourselves with the details of the physical memory layout, memory allocation strategies, or pointers. The heap is an abstract large pool of memory from which individual objects may be allocated. A memory cell is a part of the heap, of any size, which is allocated in a single allocation operation. Some memory cells are divided into non-evenly sized components called fields. A reference (used interchangeably with pointer) is an opaque handle that refers to a memory cell. If a variable, cell or field holds a reference to a value that is stored in another memory cell, that value is said to be boxed; if a variable, cell or field holds a value directly, that value is said to be unboxed. A memory cell is said to be reachable (used interchangeably with accessible) if it is referenced by any live variable or reachable memory cell.

If references $x$ and $y$ both refer to the same memory cell, $x$ and $y$ are said to be aliased (used interchangeably with shared) to one another. If reference $x$ refers to a memory cell that is reachable via a variable other than $x$, then $x$ is said to be aliased; otherwise it is said to be unique.

We will often speak about linked lists, in the context of many programming languages. Abstractly, a linked list is a sequence of elements (values of any type). Concretely, a list is a collection of cons cells, each cell containing two fields: the head (which contains or references a list element) and tail (which references the subsequent cons cell in the list, or contains the special value Nil to indicate that it is the last cons cell in the list). The elements of a list are the heads of each cons cell in the list. The first cell of a list is the first cons cell in the list. The head and tail of a list are the head and tail, respectively, of its first cons cell. The spine of a list is the sequence of all of the cons cells in the list. These terms are shown visually in Figure 1.1.
1.6 Preliminaries

In this thesis, we will be interacting with mathematical expressions at two levels: in computer language source code (the source language) and in pure mathematical abstraction (the meta-language). Both languages are written in the same font and notation, and which we are talking about should be obvious from context. This section gives notations and semantics for the meta-language, which will be used consistently throughout the thesis. In addition to the concepts presented here, we assume basic mathematical notation for arithmetic and logic.

Quine corners [84], such as \([e]\), are used to quote source language constructs, distinguishing them from expressions in the meta-language — they are essentially quotes that denote syntactic structure. For example “\([v = e]\)” represents an assignment statement of the expression \(e\) to the variable \(v\), whereas “\(v = e\)” represents the meta-language truth value of whether \(v\) equals \(e\).

1.6.1 Sets

The numeric sets \(\mathbb{N}\), \(\mathbb{Z}\) and \(\mathbb{Q}\) denote the natural numbers, integers and rational numbers, respectively, and are defined in the usual way.

The usual set notation applies: \(\{x_1, \ldots, x_n\}\) denotes a set with elements \(x_1\) through to \(x_n\); set-builder notation \(\{\mathcal{F}(x) : \mathcal{P}(x)\}\)\(^5\) denotes the set of \(\mathcal{F}(x)\) for all values of \(x\) that

---

\(^5\)Note that \(\mathcal{F}(x)\) denotes an arbitrary expression with \(x\) as a free variable; likewise, \(\mathcal{P}(x)\) denotes an arbitrary truth condition in terms of \(x\).
satisfy predicate \( P(x) \). The empty set is denoted by \( \{\} \) or \( \emptyset \). Set operations union (\( \cup \)), intersection (\( \cap \)), difference (\( \setminus \)) and cardinality (\( |S| \)), and set predicates strict subset (\( \subset \)), subset or equal (\( \subseteq \)) and membership (\( \in \)) have the usual meaning. Two sets \( x \) and \( y \) are disjoint if \( x \cap y = \emptyset \).

The notation \( S \perp \) denotes the set \( S \cup \{\perp\} \), where \( \perp \) is a special value denoting a missing or erroneous value. \( \wp(S) \) gives the power set of \( S \):

\[
\wp(S) \equiv \{ s : s \subseteq S \}
\]

\( X \times Y \) denotes the Cartesian product of sets \( X \) and \( Y \). This generalises to \( n \)-tuples:

\[
S_1 \times \cdots \times S_n \equiv \{ (x_1, \ldots, x_n) : x_1 \in S_1 \land \cdots \land x_n \in S_n \}
\]

\( S^n \) denotes the \( n \)th Cartesian power of set \( S \):

\[
S^n \equiv S \times \cdots \times S \equiv \{ (x_1, \ldots, x_n) : x_1 \in S \land \cdots \land x_n \in S \}
\]

\( S^* \) denotes the set of all \( n \)-tuples of elements of \( S \):

\[
S^* \equiv \bigcup_{n \in \mathbb{N}} S^n \equiv \bigcup_{n \in \mathbb{N}} \{ (x_1, \ldots, x_n) : x_1 \in S \land \cdots \land x_n \in S \}
\]

\( X \otimes Y \) denotes the unordered Cartesian product of sets \( X \) and \( Y \), which is a set of unordered pairs instead of ordered pairs. This also generalises to \( n \)-tuples:

\[
S_1 \otimes \cdots \otimes S_n = \{ \{x_1, \ldots, x_n\} : x_1 \in S_1 \land \cdots \land x_n \in S_n \land x_1 \neq \cdots \neq x_n \}
\]

### 1.6.2 Functions

A function is a set of pairs \((x, y)\), where each value \( x \) appears as the first element of at most one pair, and \( y \) is the output corresponding to the input \( x \). A total function over the domain \( X \) has exactly one pair \((x, y)\) for each \( x \in X \); a partial function over the same domain has at most one such pair.

For the most part, functions are treated opaquely, ignoring the sets-of-pairs notation. \( X \rightarrow Y \) denotes the set of all total functions mapping elements of \( X \) to elements of \( Y \):

\[
X \rightarrow Y \equiv \{ (x, y_x) : x \in X \} : \forall x \in X. y_x \in Y \} \]
CHAPTER 1. INTRODUCTION

$X \rightarrow Y$ denotes the set of all partial functions mapping elements of $X$ to elements of $Y$:

$$X \rightarrow Y \equiv \{(x, y_x) : x \in X' : X' \subseteq X \land \forall x \in X', y_x \in Y\}$$

$\lambda x. \mathcal{F}(x)$ denotes a function that maps input value $x$ to the value of $\mathcal{F}(x)$:

$$\lambda x. \mathcal{F}(x) \equiv \{(x, \mathcal{F}(x)) : x \in X\}$$

The form $\lambda x_1 \ldots x_n. \mathcal{F}(x_1, \ldots, x_n)$ is short-hand for $\lambda x_1. \ldots \lambda x_n. \mathcal{F}(x_1, \ldots, x_n)$.\(^6\)

A function definition of the form:

$$f x_1 \ldots x_n = \mathcal{F}(x_1, \ldots, x_n)$$

is equivalent to:

$$f = \lambda x_1 \ldots x_n. \mathcal{F}(x_1, \ldots, x_n)$$

Function definitions sometimes use Haskell-like pattern matching, where the first matching clause is evaluated. A function definition of the form:

$$f p_1(x_1, \ldots, x_m) = \mathcal{F}_1(x_1, \ldots, x_m) \ldots f p_n(x_1, \ldots, x_m) = \mathcal{F}_n(x_1, \ldots, x_m)$$

is equivalent to:

$$f q = \begin{cases} 
\mathcal{F}_1(x_1, \ldots, x_m), & \text{if } p_1(x_1, \ldots, x_m) = q \\
\ldots & \\
\mathcal{F}_n(x_1, \ldots, x_m), & \text{else if } p_n(x_1, \ldots, x_m) = q 
\end{cases}$$

The notation $f x$ denotes the application of function $f$ to value $x$:

$$f x = y \iff (x, y) \in f$$

Application is left-associative: $f x_1 \ldots x_n$ is equivalent to $(f x_1) \ldots x_n$. $f x$ is defined if and only if $\exists y. (x, y) \in f$; otherwise it is undefined. Two functions $f$ and $g$ are disjoint if and only if:

$$\neg \exists x, y_1, y_2. (x, y_1) \in f \land (x, y_2) \in g$$

(i.e., they are never both defined for the same input). Only partial functions may be

---

\(^6\)This means that all meta-language functions are automatically curried, so if $f$ is a binary function, $f x$ is equivalent to $\lambda y. f x y$. 
undefined for inputs in their stated domain. A function is *injective* if and only if:

\[ \forall x, y. \ f \ x = f \ y \implies x = y \]

(i.e., each input maps to a unique output).

\( f \circ g \) denotes the composition of functions \( f \) and \( g \):
\[
    f \circ g \equiv \lambda x. \ f \ (g \ x)
\]

\( f^n \) for \( n \in \mathbb{N} \) denotes the \( n \)th composition of \( f \):
\[
    f^0 = \lambda x. \ x \\
    f^n = f \circ \cdots \circ f
\]

\( f^{-1} \) denotes the inverse of an injective function \( f \):
\[
    f^{-1} x = y \iff f y = x
\]

The notation \( \{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\} \), where all values \( k_i \) are distinct, constructs a partial function mapping keys \( k_1, \ldots, k_n \) to values \( v_1, \ldots, v_n \), respectively, being undefined for all other keys:
\[
    \{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\} \equiv \lambda x. \ \begin{cases} v_1, & \text{if } x = k_1 \\ \cdots \\ v_n, & \text{if } x = k_n \end{cases}
\]

The expression \( f[a \mapsto b] \) produces a mapping that maps \( a \) to \( b \), and that behaves like \( f \) on all other inputs:
\[
    f[a \mapsto b] \equiv \lambda x. \ \begin{cases} b, & \text{if } x = a \\ f \ x, & \text{otherwise} \end{cases}
\]

1.6.3 Ordering and lattices

A partial order is a set \( S \) equipped with a binary predicate \( \sqsubseteq [19] \) (pronounced “less-than or equal”). The \( \sqsubseteq \) operation must be reflexive, transitive and anti-symmetrical.\(^7\) For any two values \( x, y \in S \), \( x \not\sqsubseteq y \land y \not\sqsubseteq x \) means that \( x \) and \( y \) are incomparable (neither is greater than the other). \( x \sqsubseteq y \) means that \( x \leq y \) and \( x \neq y \).

A partial order may be represented with a Hasse diagram [97]. A line between \( x \) and \( y \)

---

\(^7\)The terms reflexive, transitive and anti-symmetrical are not defined here, but carry their usual meaning.
means that \( x \sqsubseteq y \), where \( x \) is lower in the diagram; transitively inferable lines are omitted. For example, Figure 1.2 is a Hasse diagram for the partial order \( \wp(\{0, 1\}) \), with \( \sqsubseteq = \subseteq \).

![Hasse diagram](image)

Figure 1.2: Hasse diagram for the partial order \( \wp(\{0, 1\}) \)

A function \( f : X \to Y \) for partial orders \( X \) and \( Y \) is said to be monotonic if:

\[
\forall x, y \in X. \ x \sqsubseteq y \implies f\ x \sqsubseteq f\ y
\]

If \( y \) is greater than or equal to every element of a set \( X \), \( y \) is an upper bound of \( X \), written as \( X \sqsubseteq y \). The least upper bound (lub) of \( X \), written \( \sqcup X \), is the least element that is greater than or equal to every element of \( X \):

\[
X \sqsubseteq y \iff \forall x \in X. \ x \subseteq y
X \subseteq \sqcup X \land (\forall y \in S. \ X \subseteq y \implies \sqcup X \subseteq y)
\]

Similarly, if \( y \) is less than or equal to every element of \( X \), \( y \) is a lower bound of \( X \), written as \( y \sqsubseteq X \). The greatest lower bound (glb) of \( X \), written \( \sqcap X \) is the greatest element that is less than or equal to every element of \( X \):

\[
y \sqsubseteq X \iff \forall x \in X. \ y \subseteq x
\sqcap X \subseteq X \land (\forall y \in S. \ y \subseteq X \implies y \subseteq \sqcap X)
\]

For convenience, these are also defined as binary operators:

\[
x \sqcup y = \sqcup\{x, y\}
x \sqcap y = \sqcap\{x, y\}
\]

The top \( \top \) of a partial order is the least upper bound of all elements in the set (the greatest element of the set):

\[
\top \equiv \sqcup S
\]
Similarly, the bottom \( \bot \) of a partial order is the greatest lower bound of all elements in the set (the least element of the set):

\[
\bot \equiv \cap S
\]

A partial order need not have a lub or glb defined for any two elements, nor does there need to be a \( \top \) or \( \bot \) element.

A **chain** is a set \( c \subseteq S \) such that all pairs of values in \( c \) are comparable; that is:

\[
\forall x, y \in c. \ x \subseteq y \lor y \subseteq x
\]

A **complete partial order** is one in which every chain has a lub.

A **lattice** is a partial order in which lub and glb are defined for all pairs of elements in the set. A **complete lattice** has a lub and glb defined for all subsets, and therefore, has both a \( \top \) and \( \bot \) element [19]. The **height** of a lattice is the cardinality of the largest chain in \( S \); for a complete lattice, this chain spans from \( \bot \) to \( \top \).

All total orders form a lattice where \( \sqcup = \max \), \( \sqcap = \min \), \( \top \) and \( \bot \) are the maximal and minimal elements of the set, respectively, and the height is the cardinality of the set.

All power sets \( \wp(X) \) form a lattice with \( \subseteq = \subseteq \). In these cases, \( \sqcup = \sqcup \), \( \sqcap = \cap \), \( \top = X \), \( \bot = \emptyset \) and height = \( |X| + 1 \). For example, in the set of sets of integers \( (\wp(Z)) \), \( \{1, 3\} \subseteq \{1, 3, 5\}, \{1, 3\} \sqsupseteq \{3\} \) and \( \{1, 3\} \) is incomparable to \( \{1, 2\} \). \( \{1, 3\} \sqcup \{1, 5\} = \{1, 3, 5\} \) and \( \{1, 3\} \sqcap \{1, 5\} = \{1\} \). This lattice has an **infinite height**, because \( |Z| \) is infinity, but it is complete, because \( \sqcup \) and \( \sqcap \) are defined for all subsets of \( \wp(Z) \).

Unless otherwise stated, a set of \( n \)-tuples \( L_1 \times \cdots \times L_n \), with each \( L_i \) being a partial order, is ordered element-wise:

\[
(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) \iff x_1 \subseteq y_1 \land \cdots \land x_n \subseteq y_n
\]

If each \( L_i \) is a lattice, then the set is itself a lattice. The \( \sqcup \) and \( \sqcap \) are element-wise, the \( \top \) and \( \bot \) are the tuples with \( \top \) and \( \bot \) for each element, and height = \( \sum_{i=1}^{n} \) (height \( L_i \) - 1) + 1.8

As a special case, the height of the lattice \( L^n \) is \( n \times (\text{height } L - 1) + 1 \).

Unless otherwise stated, a set of total or partial functions \( X \to Y \) or \( X \rightharpoonup Y \) with a partial order on \( Y \) is ordered pointwise:

\[
f \sqsubseteq g \iff \forall x. f \ x \text{ is undefined} \lor (g \ x \text{ is defined} \land f \ x \sqsubseteq g \ x)
\]

---

8A proof sketch for this claim follows: for a tuple lattice \( L_1 \times \cdots \times L_n \), the longest path from \( \bot \) to \( \top \) for each lattice \( L_i \) is height \( L_i - 1 \) steps. The longest path for the whole lattice involves all of those steps for each element lattice in any order, which is \( \sum_{i=1}^{n} \) (height \( L_i \) - 1) steps. The height is the number of steps plus 1.
By this definition, an undefined result is considered less than any other value. This means that if a partial function \( f \) is defined for a subset of inputs that \( g \) is defined for, but produces lesser or equal results to \( g \) wherever it is defined, \( f \sqsubseteq g \). For total functions, the rules about undefined results can be ignored.

If \( Y \) is a lattice, then the set of functions is itself a lattice. The \( \sqcup \) and \( \sqcap \) are pointwise:

\[
(f \sqcup g) \ x = \begin{cases} 
  g \ x, & \text{if } f \ x \text{ is undefined} \\
  f \ x, & \text{if } g \ x \text{ is undefined} \\
  f \ x \sqcup g \ x, & \text{otherwise}
\end{cases}
\]

\[
(f \sqcap g) \ x = f \ x \sqcap g \ x
\]

\[
\top \ x = \top
\]

\[
\bot \ x = \bot \quad \text{(for total functions)}
\]

\[
\bot = \emptyset \quad \text{(for partial functions)}
\]

The total function \( \bot \) in the set \( X \to Y \) gives the \( \bot \) value in \( Y \) for all inputs (it is equivalent to \( \lambda \ x. \bot \)). The partial function \( \bot \) in the set \( X \mapsto Y \) is undefined for all inputs.

The height of \( X \to Y \) is \(|X| \times (\text{height } Y - 1) + 1\).

The height of \( X \mapsto Y \) is \(|X| \times (\text{height } Y') + 1\).

The lattice only has a finite height if \( X \) is a finite set and \( Y \) has a finite height. As a special case, if functions \( f \) and \( g \) are disjoint mapping functions, \( f \sqcup g \) is the union of their key/value pairs.

For a total function \( f : S \to S \), \( x \) is a fixed point of \( f \) if and only if \( f \ x = x \). The least fixed point of \( f \) is the unique least value that is a fixed point. If \( S \) is a complete partial order, and \( f \) is continuous [19] (implying monotonicity), the least fixed point of \( f \) can be defined by Kleene’s fixed point characterisation (from [81]):

\[
\text{lfp} \ f \equiv \bigsqcup_{i \geq 0} (f^i \ \bot)
\]

\[\text{lfp} \ f \text{ can be computed in at most height } S \text{ steps with Algorithm 1.1, which calls } f \ \bot,\text{ and then continually applies } f \text{ to the result until the output is the same as the input. Because } f \text{ is monotonic, each step will either terminate, or move in a consistent direction through the lattice (since we begin with } \bot, \text{ it must be upwards). Therefore, the algorithm is guaranteed to terminate in at most height steps.}\]
1.7. OUTLINE OF THE THESIS

Algorithm 1.1 Computation of the least fixed point by ascending the Kleene chain

\begin{verbatim}
function lfp(f):
    x = ⊥
    repeat:
        x\_prev = x
        x = f x
    until x = x\_prev
    return x
\end{verbatim}

1.7 Outline of the thesis

In this thesis, we will address two major questions:

Can a programming language be designed to prohibit global side-effects without discarding conventional imperative language constructs?

and,

Can the implementation of such a language allow efficient update of data structures, in particular, arrays?

The thesis is therefore divided into two major parts, one for each of the above questions.

The first part of the thesis is concerned with language design. In Chapter 2, we explore a number of past language proposals and implementations that in some way blend together imperative and functional or declarative programming. Over a number of decades, we have seen languages that add functional features such as first-class functions to an imperative base, imperative languages that add some optional purity guarantees, declarative languages with some simple imperative programming constructs, and hybrid “effect” languages with controlled side-effects. All of these types of language are explored in detail.

In Chapter 3, we come to the key question: what is purity? More importantly: what aspects of pure programming languages are actually beneficial for software engineering? We define a concept that we call interface integrity, which essentially describes languages that do not allow global side-effects. We argue that interface integrity gives us the major software engineering benefits of purely declarative programming languages, but does not over-constrain the language designer, allowing true imperative programming.

In Chapter 4, we put this theory into practice, outlining the design for a new (relatively simple) programming language, Mars. The philosophy behind the language and some details about its implementation are discussed.
In Chapter 5, we formally define the intermediate language that Mars compiles into. A denotational semantics for MIR (essentially the critical subset of Mars with all syntactic sugar removed) is given. This chapter also gives a second denotational semantics for MIR, formally specifying the allocation and sharing of memory cells. This second semantics is used as the basis for compile-time garbage collection analysis in subsequent chapters.

The second part of the thesis is concerned with static analysis for automatic destructive update. In Chapter 6, we give an overview of abstract interpretation, a standard method for performing static program analysis. This chapter serves as a background on common analysis techniques, but as it works through a complete static analysis for MIR, also allows the reader to become comfortable with analysis of our framework in a relatively trivial example.

In Chapter 7, we review the literature on automatic destructive update and compile-time garbage collection. We find decades of theoretical and practical analyses for a wide variety of languages, pure and impure. However, while there are simple analyses for higher-order languages, we have found no analyses that handle higher-order programming in a highly precise manner. Thus, our goal for the remainder of the thesis is to build one.

To simplify the description of our higher-order compile-time garbage collection system, we split the analysis part into two chapters. In Chapter 8, we begin to describe the analysis, just focusing on first-order programs. We describe a highly precise context-insensitive aliasing analysis that suffers a huge precision loss when applied to higher-order functions, and is therefore fairly similar to existing work.

In Chapter 9, we augment the aliasing analysis with the ability to describe higher-order functions with the same precision as we describe first-order functions. Our final aliasing analysis is context-insensitive, producing precise information about the aliasing that each function will produce, parameterised by any function parameters passed to the function. This is a stronger result than any known analysis.

In Chapter 10, we formally describe how the system actually uses the aliasing information to produce safe destructive update instructions at compile time. We explore two methods for doing so: using destruction information at runtime, and compiling separate versions of certain procedures at compile time.

Finally, Chapter 11 concludes the thesis with summaries of each chapter and a discussion of avenues for future work.
Chapter 2

Survey of hybrid imperative/declarative languages

2.1 Introduction

This thesis is primarily an exploration of the relationship between imperative and declarative programming languages. While these paradigms are often considered to be mutually exclusive, there are many programming languages, both existing and theoretical, that have blurred the line of the imperative/declarative dichotomy. The language developed alongside this thesis, Mars, is one that we consider to be both imperative and declarative. In this chapter, we review past languages that conflate the two paradigms in one way or another.

We begin this chapter by exploring the perceived dichotomy between the imperative and declarative paradigms. In one corner, we have the imperative paradigm, which covers most of the programming languages in history, from assembly languages, to the venerable and low-level C, to modern and highly abstract languages such as Java, C#, Python and JavaScript. Imperative languages are primarily characterised by featuring statement sequences, sequences of instructions, each of which has some effect on the state of the computation [52].

In the other corner, we have the declarative paradigm, which generally refers to functional, logic and constraint languages, such as Lisp, ML, Haskell, Prolog and Mercury. It can also refer to certain domain-specific languages, such as the SELECT query in SQL, or formulas in spreadsheets. Declarative languages are primarily characterised by programs consisting of statements about the results to be computed; such programs specify what to compute, but not how to compute it [61] (for example, the order of evaluation is...
typically not important).

**Example 2.1** Appending a linked list, in Python (a) and Prolog (b)

```python
def append(x, y):
    if x is None:
        return y
    else:
        t = append(x.tail, y)
        return List(x.head, t)
```

```
append([], Y, Y).
append([H|T0], Y, [H|T]) :-
    append(T0, Y, T).
```

Consider Example 2.1, a linked list `append` operation written in the imperative language Python and the declarative logic language Prolog.\(^1\) The Python version can be read as a sequence of instructions, executed one after the other. The Prolog version, however, is best read as a set of rules. In this example, the rules are:

1. An empty list appended to $Y$ is equal to $Y$.
2. A non-empty list (having head $H$ and tail $T_0$) appended to $Y$ is equal to a non-empty list (having head $H$ and tail $T$), where $T$ is $T_0$ appended to $Y$.

The declarative version does not contain any instruction to read the head and tail out of a list, nor to construct a new list; it simply gives a set of rules by which the `append` predicate holds. This is what gives Prolog its famous ability to solve goals in different directions. For example, if the above predicate is called with “`append(X, Y, [1, 2, 3])`,” Prolog will find all values for $X$ and $Y$ that satisfy the rule. Not all declarative languages have this ability (and this thesis will not spend much time on logic languages), but it is an example of the key difference between imperative and declarative programming.

It is easy to see these paradigms as mutually exclusive: declarative programs do not specify how to compute the result, and hence by definition, are not imperative. However, as we shall see, there are many language designs that blur the line between these two schools of thought. It is also unclear precisely what these terms mean. For example, if functional programming is considered to be declarative programming, then we might consider ML to be a declarative language: like Haskell, ML functions are not allowed to re-assign variables. However, ML is *impure* in that it allows the creation of `reference` objects which may be mutated.

In Example 2.2, the `inc` function takes a reference variable, increments it (mutating the reference), and then returns its new value. This mutation means that we cannot reason about ML programs as sets of declarative statements — for example, when describing

\(^1\)These were chosen for comparison because they are both high-level dynamically typed languages. The Python version assumes a class List with a constructor that sets the `head` and `tail` attributes.
2.1. INTRODUCTION

Example 2.2 Impure code in ML

```ml
fun inc counter = (counter := !counter + 1; !counter)
fun two_counters () =
  let val counter = ref 0
  in inc counter - inc counter
  end
```

the expression `inc counter - inc counter`, we must consider the update of the reference as well as the value of each subexpression, which means that unlike an ordinary mathematical expression, we must consider the order in which the left and right operands are evaluated. Thus, ML cannot be said to have truly declarative semantics.\footnote{This is by no means limited to ML. Many declarative languages, including most Lisp dialects, allow arbitrary mutation, dating back to the original Lisp [69], which featured the `rplaca` and `rplacd` functions for destructively updating the head and tail, respectively, of a list.} It is a strange mix of imperative and declarative: it has the mutation of references and other side-effects found in imperative languages, but does not allow local variable update or looping. Even Haskell, one of the few truly pure functional programming languages, contains the imperative-style `do` notation, allowing a sequence of statements that conceptually update some state (although the state update is controlled by a monad, so this is syntactic sugar for pure programming, as opposed to actual side-effects).

These hazy overlaps between the two apparently mutually exclusive paradigms can be explained by there being two related definitions of each. We can define imperative programming as either of the following:

1. Programming with a sequence of statements, each of which can change the state of the computation. This definition includes Haskell’s `do` notation and the monadic style in general [52].

2. Programming that relies on side-effects such as mutation of data structures and input/output [25]. This definition includes ML’s `ref` objects and Lisp’s `cons` cell mutation functions.

The two concepts tend to be conflated (indeed, either definition can be applied to most imperative languages), but we see a subtle distinction: the former is a matter of style, while the latter is a matter of semantics. The former definition regards imperative programming as largely a syntactic mechanism for describing a computation; the latter regards functions in imperative programs as fundamentally something other than pure mathematical functions.

Likewise, we can define declarative programming as either of the following:
1. Programming by defining a set of rules or equations for the implementation to solve, without specifying how they are to be solved [61]. This definition includes ML, Prolog, and other impure functional and logic languages.

2. Programming without side-effects, where each function or predicate has no effect on its callers beside the obvious result(s) that it returns. This definition includes only pure languages, such as Haskell and Mercury. Many authors use the term “purely declarative” to distinguish this from the former [83].

Again, the former definition is a matter of style, while the latter is a matter of semantics. The former describes the manner in which the programmer communicates with the compiler, and thus, is ultimately a syntactic property; the latter regards functions in declarative programs as semantically the same as pure mathematical functions.

The reason we have taken such pains to describe the different ways to define these paradigms is that, upon considering these subtly different definitions, we can see that they actually form not one, but two dichotomies: one of style; the other of semantics. The imperative style seems (somewhat) incompatible with the declarative style, and the imperative semantics seems to be incompatible with the declarative semantics. But it seems quite plausible to have a language with declarative style and imperative semantics — this would include all of the ML-like languages that are expressed functionally, but are impure. It seems equally plausible to have a language with declarative semantics (not permitting side-effects) and imperative style. However, there are very few examples of this approach. Our language, Mars, is one such example, and it will be described in detail in Chapter 4. For the remainder of this chapter, we describe other ways in which languages combine aspects from these two paradigms.

It is historically interesting to note that the imperative/declarative dichotomy goes back further than the advent of general-purpose programmable computer hardware. Two of the most important (and equivalent) theoretical models for computation: the universal Turing machine [91] and the lambda calculus [13] were formalised in 1936. The universal Turing machine is a hypothetical machine which executes a “program,” represented as a set of instructions, each of which alters the state of the machine and then moves to another instruction — essentially a theoretical model for imperative programming. Likewise, the lambda calculus is a system of rules for expressing and evaluating mathematical expressions, including the ability to define and apply pure mathematical functions — essentially a theoretical model for declarative programming. Nearly all historical and modern computers (and many of the higher-level languages that followed) are fundamentally imperative, like the Turing machine. Similarly, declarative programming languages such as Lisp [67] are essentially implementations of the lambda calculus.
2.2 Imperative languages with functional features

The first type of imperative/declarative cross-over which we shall explore is imperative languages with functional language features — in particular, first-class functions and lambda expressions. In this thesis, we are primarily concerned with the incorporation of declarative semantics (i.e., purity) into imperative programming, so these languages, which incorporate declarative style (e.g., higher-order functions) are largely orthogonal to this concept. Nevertheless, this idea is usually the first to come up when we think about blending imperative and declarative languages, so it is worth addressing up front.

The two defining characteristics of functional programming languages (compared to languages with simple function pointers, like C) are lambda expressions (or anonymous functions) and closures. Lambda expressions allow the creation of function values inside an expression, without requiring a name or full declaration. For example, this Haskell expression creates a plus function:

\[ \lambda x y \rightarrow x + y \]

Closures allow nested functions (anonymous or otherwise) to refer to variables in the scope enclosing the function. For example, this Haskell function creates a function object (known as a closure) that remembers and uses the value of the argument \( x \):

\[ \text{add}_x = \lambda y \rightarrow x + y \]

While such features were once found primarily within the domain of functional languages such as Lisp, ML and Haskell, they appear to have "gone mainstream" in the past few decades, as nearly all modern languages fully support both closures and anonymous functions. For example, the imperative language Python treats all functions as first-class values, and \( \text{add}_x \) can be written in Python as easily as in Haskell.\(^4\) Other imperative languages that support these features include C#, C++, Java, JavaScript, Go, PHP, Ruby, and Scala. We might consider all of these languages to be functional, but it is important to note that this does not make them declarative, as they still allow arbitrary side-effects.

Other features traditionally associated with functional programming are also making their way into imperative programming. The object oriented language Scala \([75]\) contains many advanced functional programming features, including short-hand notation

---

\(^3\)Recall that in this thesis, we stylise language syntax with mathematical notation; in real Haskell code, one would use "/" instead of "\( \lambda \)" and "\( \rightarrow \)" instead of "\( \rightarrow \)."

\(^4\)However, in most non-declarative languages such as Python, surprising behaviour can result from a variable being captured by a closure, and subsequently updated; the programmer may expect the closure to retain the original value of the variable. This distinction is discussed in Section 3.3.7.
for writing curried-style functions, algebraic data types (called “case classes”) and pattern matching. These are welcome features to a functional programming audience, and may be said to contribute towards a declarative style, but we are primarily concerned with declarative semantics.

Obtaining purely declarative semantics is not a matter of adding new abilities, but rather, taking abilities away: chiefly, the ability for any function to perform arbitrary side-effects.

### 2.3 Imperative languages with purity constraints

At the extreme end of declarative programming is Haskell, which strictly prevents modification of data structures. However, imperative languages often give the programmer the ability to restrict the mutation of certain objects, for the purpose of writing more robust software. This could be considered a step towards declarative programming.

In C and C++, the `const` type qualifier, applied to a pointer or reference to an object, prevents the mutation of that object. For example, if a function accepts an argument of type `const char*`, it typically expects a pointer to an immutable string. The `const` is helpful in communicating that the function will not modify the string. This has a few obvious drawbacks: it is not strictly a guarantee, as the function could cast the `const` away, and the `const` only applies to the top-level data — a `const` pointer to an object with non-`const` pointers allows the subsequent objects to be mutated (`const` is not transitive).

There is also no way to guarantee that the function will not have other side-effects, such as performing input/output or accessing static or global variables.

Fortran 95 introduced a `pure` keyword that allows functions and subroutines to be guaranteed free of side-effects [44]. Pure functions and subroutines are prohibited from performing input/output, mutating their input arguments, modifying global variables, or calling non-pure procedures. This feature allows the enforcement of declarative semantics on selected parts of a program. As with C’s `const` feature, it is off by default.

However, Fortran has a handful of “leaks” which allow side-effects to seep into `pure`-annotated procedures. While pure functions are not allowed to `write` global variables, they are allowed to `read` globals, as in Example 2.3. The `get_counter` function is supposedly pure, yet its result is not determined by its arguments. While it does not `cause` side-effects, it can `observe` side-effects, such as having the `counter` variable incremented.

A further “leak” involves pure subroutines. Fortran subroutines are allowed to spec-

---

5However, in C++, there is a very strong convention that `const` objects transitively enforce their own immutability.
2.3. IMPERATIVE LANGUAGES WITH PURITY CONSTRAINTS

Example 2.3 Reading a global variable in a Fortran pure function

```fortran
integer :: counter
pure function get_counter() result(r)
  integer :: r
  r = counter
end function
```

ify each parameter with an `intent` of `in`, `out` or `inout`. Pure subroutines may modify parameters with the `out` or `inout` intent. This by itself is not impure (rather than being considered a side-effect, setting `out` variables may be considered part of the result of the subroutine), but leads to side-effects when arguments are aliased to one another.

Consider Example 2.4, which contains a subroutine `arg_update` that is intended to return two values: `b`, with the value 42, and `c`, with the value of the input `a`. This is usually the case, except that in our program, we pass the same variable `n` to both parameters `a` and `b`, which can behave unexpectedly, depending on the compiler. If the compiler passes arguments by reference, when `b` is assigned the value 42, so is `a`, and the program prints 42 instead of the intended 9. This supposedly pure subroutine’s results are not just based on the values of its inputs, but also on whether or not they alias one another, and hence it cannot be considered equivalent to a pure mathematical function.

Example 2.4 Aliased arguments to a Fortran pure subroutine

```fortran
pure subroutine arg_update(a, b, c)
  integer, intent(in) :: a
  integer, intent(out) :: b
  integer, intent(out) :: c
  b = 42
  c = a
end subroutine
```

```fortran
program arg_alias
  integer :: n, m
  n = 9
  call arg_update(n, n, m)
  print*, m
end program
```

The Eiffel programming language encourages a discipline known as `command-query separation`, a concept proposed by its designer, Meyer [70]. The principle is that every procedure should be either a `command`, performing side-effects but not returning a value, or a `query`, returning a value as determined by the arguments and the state of the com-

---

6Indeed, this behaviour is considered undefined by the language specification [44], but the `gfortran` compiler accepts it and gives it reference semantics. The fact that the specification calls this behaviour “undefined” does not help us write a correct program if the compiler does not enforce it!
putation, but not performing any side-effects. This principle is best demonstrated by Example 2.5, which shows a usage of Eiffel’s unusual line input library.

**Example 2.5** Reading a line from standard input in Eiffel

```
io.read_line
s := io.last_string
```

This example shows two procedure calls: one to the command `read_line`, and one to the query `last_string`. The command-query separation principle states that `read_line`, which performs I/O, cannot also return a value. Therefore, it instead stores the value in a variable inside the `io` object (effectively, in a global variable). Then, we can use the `last_string` query to retrieve the variable.

This gives the guarantee that querying for a value will not change the state of the computation, and hence is a move towards purity in imperative programming. However, note that queries provide weaker purity than the pure mathematical functions of declarative language: as with Fortran’s pure procedures, Eiffel queries may inspect the state of the computation, whereas a pure function would be dependent only on its arguments. Furthermore, it is important to note that this principle is not enforced by the language’s compiler; it is merely recommended programming style in this language.

### 2.4 Declarative languages with imperative constructs

In the previous section, we discussed languages with imperative style and semantics, but with features moving in the direction of declarative semantics. In this section, we look at a similar concept from the other side: languages with declarative style and semantics, but with features moving in the direction of imperative style.

The language ISWIM was proposed by Landin in 1966 [55]. The language borrowed concepts from ALGOL and Lisp, and introduced numerous new concepts, such as `where` clauses. A key goal of ISWIM was combining imperative concepts onto a purely functional language, which Landin argued was better than the other way around:

> [This paper] discussed adding ‘where’ to a conventional programming language. Theory and experiment both support the opposite approach, that taken in LISP, of adding imperative features to a basically nonimperative language. One big advantage is that the resulting language will have a nonimperative subset. The special claim of ISWIM is that it grafts procedural notions onto a purely functional base without disturbing many of the desirable properties.

This appears to have the same intentions as our research, but it appears that Landin’s
“desirable properties” were that the language had a purely declarative subset, as opposed to the entire language being pure.

The two imperative features that were added were assignment statements, and a goto-like feature known as “program points.” Landin only gives the syntax, not the semantics, for his language, so the scope of these features is unclear, but it is assumed that they violated referential transparency. The paper gives axioms along the lines of referential transparency: that the expression “\(L\) where \(x = M\)” be equivalent to \(L\) with all occurrences of \(x\) substituted by \(M\), and then states:

This rule is the most important, but it has only limited validity, namely, within the “purely functional” subset of ISWIM that results from not using the program-point feature or assignment.

It should also be pointed out that ISWIM does not add any kind of iteration operator. Hence, ISWIM can be seen as a mostly purely functional language, but with a few imperative constructs that make the language impure.

The ECLiPSe programming language\(^7\) is a constraint logic language that is a superset of Prolog. It adds several new features to Prolog (such as a constraint solver), but what is of interest to us is that it adds an iteration construct. Example 2.6\(^8\) demonstrates the computation of the sum of a list of numbers using the do loop. The syntax is quite general, with the do loop being given a set of “iteration specifiers” such as foreach and fromto. Each of the built-in iteration specifiers can bind variables within the body of the loop to a new value on each iteration, and possibly cause the loop to terminate. In this example, the foreach specifier binds the variable \(X\) inside the loop, and also terminates the loop when the list runs out; the fromto specifier binds \(In\) to 0 on the first iteration, and to the value of \(Out\) on each subsequent iteration, finally binding \(Sum\) to \(Out\) when the loop terminates.

---

**Example 2.6 Iteration in ECLiPSe**

```
sum(L, Sum) :-
   (foreach(X, L),
    fromto(0, In, Out, Sum)
    do Out is In + X).
```

This convenient notation allows algorithms to be coded in a more imperative style,

---

\(^7\)Not to be confused with the Eclipse software development environment.

\(^8\)Adapted from [http://eclipseclp.org/doc/tutorial/tutorial025.html](http://eclipseclp.org/doc/tutorial/tutorial025.html).
without any weakening of the declarative semantics of Prolog. However, this feature allows for, at best, a poor approximation of the conventional imperative style. The variables used and updated inside the loop must be carefully controlled by the iteration specifiers (there is no straightforward way to simply re-assign a variable with a new value in the loop, as one would in a conventional imperative language), and there is no equivalent of a while loop (although we show how to emulate one in Example 3.4).

In a similar manner, Clojure [39] has a built-in loop construct, while Sisal [24] is a purely declarative language with for and while loops. In both languages, like ECLiPSe, the looping construct is syntactic sugar for recursion: it permits only specially designated loop variables to be updated in a loop, not arbitrary local effects as in imperative code.

On the other hand, the Sisal-inspired SAC (Single Assignment C) [29] incorporates C’s local imperative style into mostly pure declarative semantics. Unlike the languages mentioned earlier, SAC allows local variables to be arbitrarily updated, including in conditionals and loops, without creating a new scope (for example, an array sum function can be written as in C), but eschews C’s mutable data (through pointers) and global variables. SAC functions are typically written imperatively, but are mathematically pure, other than the ability to perform arbitrary I/O side-effects. This closely matches our goals, other than providing I/O purity guarantees, and other functional programming constructs such as closures.

In Haskell, the do notation provides some access to both imperative style (a sequence of effectful statements executed in order) and imperative semantics (functions that perform side-effects) in a controlled manner. For these reasons, Haskell was called by its author “the world’s finest imperative language” [48].

The most famous usage of the do notation is for performing I/O, as shown in Example 2.7. This example shows a sequence of three “statements,” each of which has an effect: the first and third produce program output, while the second consumes standard input and also binds a variable.

Example 2.7 Simple imperative I/O in Haskell

```haskell
main = do
    putStrLn "Enter your name:"
    name ← getLine
    putStrLn ("Hello, " ++ name)
```

It is important to note that this is not really imperative code: the bound variables are

---

9Prolog and ECLiPSe are in no way purely declarative, due to their support for arbitrary I/O and the infamous assert and retract commands; however, this has no bearing on ECLiPSe’s do notation, which could be equally applied to a pure version of Prolog.
really lambda-bound, so are scoped like normal Haskell variables, and thus if they are bound within some nested scope (such as the then or else branch of an if statement), will not be available afterwards. Similarly, there are no looping constructs, and while higher-order functions can be used to simulate iteration, this will be subject to the same constraints on variable binding. In fact, the do notation is merely syntactic sugar for a series of “bind” operations (represented by the >>= operator). The same code is shown desugared in Example 2.8. Nevertheless, this allows some degree of imperative programming (the paper that introduced this concept was titled Imperative functional programming [52]).

**Example 2.8** Desugared version of Example 2.7

```haskell
main = putStrLn "Enter your name:" >>= λ_ →
    getline >>= λ name →
    putStrLn ("Hello, " ++ name)
```

It is also important to note that this does not allow side-effects in Haskell functions that appear to be pure (unlike Prolog and ML, for instance). The type of putStrLn is String → IO (), and the type of getline is IO String, while the type of the whole main function is, in this case, IO (). A function with an ordinary type such as String cannot perform I/O effects. This means that all Haskell programs are clearly separated into the “I/O” part and the “pure” part.

The generalisation of this concept is known as a monad [52]. Any data type that defines a sequencing operator ( >>= ), and has a few other properties, can be thought of as a monad — IO is one such monad. The do notation can be used with any monadic type. This makes it possible to define monads that store various types of state, to emulate imperative programming in various ways. For example, the ST (state transformer) monad’s state is an environment mapping mutable references to values. In this monad, variables can have an STRef or STArray type, and “statements” can modify the values associated with the reference without re-binding the variable. Unlike regular variable bindings, this allows reference updates to last beyond the scope of the update instruction. As with IO, these are not really side-effects, because they are just syntactic sugar for passing around and updating a dictionary, and the reference updates cannot affect code that does not have an ST return type.

Example 2.9 shows the implementation of a set data type in Haskell, using an STArray as a hash table. Functions empty, insert and member (implementation not shown), respectively, create, update and test membership in a hash table. This code assumes a type class Hash with method hash :: Hash a ⇒ a → Int. This is a simplified implementation with several flaws: it allows duplicates, does not resize when near capacity, and resolves collisions with linear probing. Nevertheless, it illustrates the use of a mutable array.
Example 2.9 Hash table implementation and usage in Haskell

```haskell
empty :: Int → ST s (STArray s Int (Maybe a))
empty capacity = newArray (0, capacity - 1) Nothing

insert :: Hash a ⇒ STArray s Int (Maybe a) → a → ST s ()
insert table value = do
  (_, hi) ← getBounds table
  let insert' h = do
        v ← readArray table h
        case v of
          Nothing → writeArray table h (Just value)
          Just _ → insert' ((h + 1) `mod` (hi + 1))
        insert' (hash value `mod` (hi + 1))

member :: (Eq a, Hash a) ⇒ STArray s Int (Maybe a) → a → ST s Bool

intersection :: (Eq a, Hash a) ⇒ [a] → [a] → [a]
intersection xs ys = runST $ do
  table ← empty (length xs × 2)
  mapM_ (insert table) xs
  filterM (member table) ys
```

An important reason to use the ST monad is performance. The `writeArray` function can update an STArray in constant time, whereas updating a regular, pure array takes linear time. The function `intersection` computes the elements common to two lists by creating a hash table, inserting the first list (`xs`) into the hash table, then testing whether each member of the second list (`ys`) is a member of the table. Where `m` and `n` are the lengths of `xs` and `ys`, respectively, this example runs in $O(m + n)$ time.\(^\text{10}\)

However, this efficiency comes at a price: the imperative performance characteristics come with some degree of imperative semantics. Users of the hash table operations (such as `intersection`) must be careful about aliasing the hash table object, because any updates to the reference would affect all aliases, just like in an imperative language. This is still technically pure, as all of the side-effects are captured by the ST monad, but it is still largely missing the point of declarative programming, at least with respect to the `table` array. Any code that deals with the array is allowed to mutate it, and that mutation affects all references to the array. Calling any function inside the ST monad could mean the array is modified, despite not mentioning the array as an output. Note, for example, the expression “`insert’ ((h + 1) `mod` (hi + 1))” causes the array referenced by `table` to change, despite not mentioning the variable by name. Avoiding these un-annotated side-effects

\(^{10}\)Assuming a uniform hash function and data with no duplicates.
is our primary motivation for using a language like Haskell in the first place.

As with the IO example, use of the ST monad tends to divide Haskell code into “imperative” and “pure” parts, with access to the imperative state being prohibited from the pure parts of the program. The table constructed by this code is not accessible outside of the call to runST, so any code that needs access to the hash table, such as intersection, must become “infected” by the ST monad, exposing the imperative semantics and associated risks to the calling code. An important aspect of the languages proposed in this thesis will be the ability to have efficient destructive update without the problems associated with aliasing.

This section has shown a few examples of imperative-like constructs in declarative programming languages. The wider subject of this thesis is the incorporation of imperative programming styles with declarative semantics, and it may seem, at this point, unnecessary, since languages like ECLiPSe and Haskell already allow imperative programming. However, this just serves to show that there are varying degrees of clarity and expressiveness with which programming languages allow imperative algorithms to be expressed: merely supporting sequence notation does not imply that these languages support natural and conventional imperative programming. We will address these concerns further in Chapter 3.

2.5 Languages with explicit effect annotations

Another, rather more direct, approach to integrating imperative and declarative programming is an effect system. Much like a type system tracks the values that may be stored in particular variables, an effect system tracks the side-effects that may be produced by particular functions, allowing the programmer to reason about mutation and side-effects more formally than in a traditional imperative language.

Whereas Haskell bottles up side-effects in semantically pure objects, giving the whole program declarative semantics, languages with effect systems simply let the programmer denote certain procedures as having imperative semantics. More powerful effect systems allow tighter control over what effects are allowed. Such a language can be seen as having separate declarative and imperative “sublanguages;” these have been called fluent languages by Gifford and Lucassen in their paper, Integrating functional and imperative programming [25].

That work introduces a simple “fluent language” derived from Lisp, with each expression (and, by extension, every procedure) belonging to one of four effect classes: PROCEDURE, OBSERVER, FUNCTION or PURE.
• A Procedure may have full side-effects: it may read or write any memory location, and may allocate memory.

• An Observer may not write to any memory, but it may read from any memory location, and allocate (and initialise) memory.

• A Function may not write to memory, nor may it read from any mutable memory location, but it may allocate (and initialise) memory. For example, a Function could not read an element from an array, but it could read the length of an array, because the length of an array cannot be changed once allocated, and it could allocate a fresh array.

• A Pure may not write, read or allocate memory.

By default, the language would automatically infer the effect class of each expression, but the programmer can also explicitly assert an expression’s effect class. Of course, expressions of a given effect class cannot contain expressions of a more permissive effect class (e.g., a Function cannot call an Observer). Such a language cannot be considered to have declarative semantics, since Procedure and Observer can, respectively, cause and observe arbitrary effects. Furthermore, because the language automatically infers effect classes, it is impure by default. Purity is enforced only by explicit annotation, which places such languages closer to Fortran 95, with an explicit pure annotation that is off by default (discussed in Section 2.3).

Disciple [60] is a strict-by-default dialect of Haskell which implements the effect system of Talpin and Jouvelot [87], a much more expressive effect system than that described above. In Disciple, the programmer can annotate variables with regions, giving an algebraic name for the memory cell(s) occupied by the variable, and may specify constraints on the region such as Mutable (for a mutable region). In addition, the programmer can add effect annotations on functions to describe the effects that are allowed to take place during the execution of that function. For example, a function that destructively updates an integer value may have the type ([60] §2.3.1):

\[
\text{inc} \quad :: \forall r_1. \, \text{Int } r_1 \overset{e_1}{\rightarrow} ()
\]

\[
\overset{\triangleright}{e_1 = \text{Read } r_1 \lor \text{Write } r_1}
\]

\[
\overset{,}{\text{Mutable } r_1}
\]

This is a function which would ordinarily have the type Int \( \rightarrow () \), augmented with a region annotation (“the Int parameter has region \( r_1 \), which must be mutable”) and an

\footnote{The pure keyword in Fortran is closer to Gifford and Lucassen’s Observer effect class, since it allows allocation and the observation of side-effects, as shown in Section 2.3.}
2.6 Conclusion

In this chapter, we have looked at the history of programming languages, with an interest in languages that in some way or another blur the line between imperative and effect annotation ("the function inc has effect \( e_1 \), which means that it is allowed to read and write to any object in region \( r_1 \))."

It is important to note that Disciple will infer region and effect information just as Haskell infers type information. Furthermore, it is possible to declare a function with type information (e.g., \( \text{inc} :: \text{Int} \rightarrow () \)) and have the Disciple compiler infer the region and effect information. This is a welcome feature, given the complexity of region and effect annotations, but we note that it further weakens the guarantees that can be made about programs. Unless the programmer explicitly annotates a procedure as "Pure," any effect that is not annotated will be inferred and automatically propagated up the static call chain. The result is that any function is allowed to produce arbitrary side-effects unless the programmer explicitly annotates it otherwise.\(^\text{12}\)

We consider a Disciple version of the hash table of Example 2.9. The insert function in the Disciple version is shown in Example 2.10. Aside from minor syntactic differences, the main distinction from the Haskell version is the lack of the ST monad: in Disciple, normal arrays may be mutated without a monad.\(^\text{13}\) The effect annotation \( e_0 = \text{Read} r_0 \lor \text{Write} r_0 \) indicates that the array (at region \( r_0 \)) may be mutated, but note that if this annotation is omitted, the program will still compile, as the annotation will be inferred. Example 2.10 Hash table insert function in Disciple

```
insert :: Array \( r_0 \) (Maybe String) \rightarrow\text{String} \overset{e_0}{\rightarrow} ()
> e_0 = \text{Read} r_0 \lor \text{Write} r_0, \text{Mutable} r_0

insert table value =
let insert' h = do
    v = table.(h)
    case v of
        Nothing \rightarrow table#(h) \#= Just value
        Just _ \rightarrow insert' ((h + 1) \mod table.size)
    in insert' (hash value \mod table.size)
```

\(^\text{12}\)This does not condemn us entirely to the chaos of impure languages: it is possible to explicitly require that certain functions have a pure interface, and this is important for lazy evaluation. Disciple is strict and effectful by default, but the programmer can explicitly request lazy evaluation of a function call, and the language will automatically require that the function is pure ([60] §2.3.9).

\(^\text{13}\)Also, note that the element type had to be changed from \( \text{Hash} \ a \Rightarrow \ a \) to String, because Disciple does not yet support type constraints.
CHAPTER 2. SURVEY OF HYBRID IMPERATIVE/DECLARATIVE LANGUAGES

declarative programming.

Several definitions for each of “imperative” and “declarative” were discussed. We argue that there are two largely orthogonal notions: whether the style is imperative or declarative, and whether the semantics are imperative or declarative. This leaves us with ample space to explore various combinations of the two. Due to this confusion, we have opted not to formally define either imperative or declarative — in the next chapter, we describe a new formal definition for pure programming.

In the rest of this chapter, we explored a number of ways in which imperative and declarative programming have been combined. Many modern imperative languages such as Python and Scala incorporate functional language concepts, such as higher-order programming (first-class functions), partial application, algebraic data types, and pattern matching. However, having imperative roots, these languages are invariably impure. Nevertheless, several imperative languages, including C and Fortran, include features that guarantee that certain data is immutable, or that certain functions are free of side-effects.

From the other side of the coin, languages with a purely declarative base have added imperative features. In some cases, such as the proposed language ISWIM, these have caused the language to become impure (much like other declarative-style languages such as Lisp, ML and Prolog). In ECLiPSe, Clojure, Sisal and SAC, we see the addition of a looping construct that does not introduce any side-effect semantics. Similarly, Haskell adds a rich imperative sub-language (monads) that allow a limited imperative style of programming without compromising the language’s purity. Finally, a number of declarative languages add controlled impure side-effects using effect systems. These “fluent languages” allow side-effects, but also allow the programmer to strongly guarantee that functions will be free from effects, or in more advanced systems such as Disciple, to control precisely which effects may occur.

Our ultimate goal in this thesis is the development of a language that allows (but does not force) the programmer to use imperative style, but which is absolutely free from functions causing or observing semantic side-effects. Such a language has not been seen in our language survey. In the next chapter, we give the motivation for building such a language, formally describe the condition by which we judge a language to be “pure,” and explore the effects of such a design on various programming constructs that imperative programmers take for granted.
Chapter 3

Interface integrity

3.1 Introduction

It is important to begin any discussion of the design of a programming language with a view to the desired characteristics of the programs written in the language. Two of the most important characteristics of good programs are maintainability and reusability, so we expect a good programming language to ensure, or at least encourage, these traits. To achieve maintainability, it is important for changes to one part of the program to cause few defects elsewhere, and for any new defects to be easily discovered and corrected. Thus a program should be decomposable into modular units with few and predictable interactions with one another. Conversely, reusability requires that modular units be composable in flexible ways with predictable results. Modularity, then, is important to ensure maintainability and reusability.

The interface of a procedure is crucial for ensuring modularity by defining the communication channel between the procedure (the callee) and its callers. Ideally, the interface should specify everything the caller and callee need to know about each other. Software becomes unmanageable when callers must consider all of the implementation details of the procedures they call, so having a well-defined interface means the caller and callee can evolve independently without having to consider each other’s private concerns. This was noted as early as 1974 by Stevens et al. [86], who wrote:

*Simplicity can be enhanced by dividing the system into separate pieces in such a way that pieces can be considered, implemented, fixed, and changed with minimal consideration or effect on the other pieces of the system.*

This is a general argument in favour of low coupling, a term coined in the same paper, but it can be used as an argument against side-effects. The interface of any procedure
with a side-effect does not specify everything the caller needs to know about it. For example, any global variable used by a procedure may affect its callers, despite not being mentioned in its declaration. There may be comments describing the side-effects, but without compiler checking, these are likely to be incomplete, informal, out-of-date, or not applied recursively (for example, if \(a\) calls \(b\), and \(b\) has a side-effect, it may be documented in \(b\), but not \(a\)).

Therefore, programmers have long been advised to avoid using global variables, or other variables with global lifetime (such as \texttt{static} variables in C). Stevens \textit{et al.} continue:

\begin{quote}
A module interfacing with a common environment for some of its input or output data is, on the average, more difficult to use \ldots than is a module with communication restricted to parameters in calling sequences.
\end{quote}

In effect, the “common environment” (global state) is both input and output of every procedure, because it can be used or modified anywhere. Any mutable data structure passed between procedures, and any object somehow accessible from one, must be considered to be both an input and an output, since it may be modified by the callee. For example, a procedure with local reference variables \(x\) and \(y\) may pass only \(x\) to a procedure, and nevertheless find that \(y\)’s object has changed because it was accessible via \(x\). This sort of defect is very difficult to find, because the developer has no way, in general, to know whether or not one variable refers to another. All of these possibilities greatly complicate the understanding of a procedure’s behaviour.

This has led many in the software industry, such as Bloch [8] and Marx [64], to recommend that immutable objects (those that cannot be modified once created) be preferred over mutable ones. Mutable objects can break encapsulation unless programmers are careful. For example, it is common practice to protect objects from arbitrary modification by making fields private and using a “getter” to expose their values in a read-only fashion. However, if the getter returns a mutable object, the caller will be able to indirectly mutate the containing object;\footnote{For example, in version 1.1 of the Java standard library, the method \texttt{Class.getSigners} returned a pointer to a mutable internal array, creating a security vulnerability that allowed malicious code to modify Java’s internal list of signers [5].} the programmer needs to make a “defensive copy” [76] of all such fields to avoid this.

While making objects immutable is commendable, even a single mutable type in a program may still allow most of the program to have side-effects. It is not enough to minimise the number of immutable \textit{types}: one must minimise the amount of \textit{code} that, directly or indirectly, mutates objects. Meyer [70] recommends that:
3.1. INTRODUCTION

If any two modules communicate at all, they should exchange as little information as possible. ... Whenever two modules A and B communicate, this must be obvious from the text of A or B or both.

In other words, interfaces should be small (exchanging as little information as possible), and comprehensive (the interfaces should reflect all the information exchanged). Note that writing comprehensive interfaces encourages writing small interfaces: if programmers are forced to explicitly mention all information flow, they are less likely to create unnecessary interactions, and will instead find more modular ways around problems. Therefore, we consider comprehensive interfaces to be very important in software construction. We call this concept interface integrity, and formally define it in Section 3.2.

History shows us that merely warning of the dangers of bad programming practices does not prevent them. For example, unchecked array indexing and pointer arithmetic in languages like C have given us untold numbers of buffer overrun vulnerabilities. Modern languages such as Java and Python have been very successful in completely preventing buffer overruns and other memory safety defects through careful language design. Similarly, a language can be designed to ensure that all interfaces are comprehensive.

Making all interfaces comprehensive gives programmers the information they need to predict the result of composing functions in different ways. It also guarantees that functions are functions in the mathematical sense: their inputs uniquely determine their outputs. Purely declarative programming languages, such as Haskell and Mercury, prohibit all side-effects. These are general-purpose programming languages used to write real software; for example, Haskell has been used to write 3D games, web servers and frameworks, database systems and compilers [35]. The modularity ensured by these languages has been called “the key to the power of functional languages” [42].

There are other practical benefits of languages that ensure comprehensive interfaces. Software transactional memory is a promising approach to handling conflicts between concurrent processes by rolling back and retrying any computation with a conflict. While it has been implemented in many programming languages, Harris et al. [33] argue that “a purely-declarative language is a perfect setting for transactional memory,” since it can prevent computations from performing operations that cannot be rolled back. Similarly, implicit parallelism, where the compiler automatically makes sequential code run in parallel, becomes possible if the compiler is aware of all effects produced by a computation [34].

Rules enforced by a language are also much more likely to be enshrined in the lan-

---

2With the exception of interfacing to code written in another language, and assuming no implementation bugs.
guage’s ecosystem. The disciplined programmer in a language that does not enforce comprehensive interfaces will inevitably have to deal with the global effects caused by language built-ins and third-party libraries. It would be extremely difficult to write even a moderately sized pure program in C, although it is theoretically possible. Such a task would be immeasurably easier in a language like Haskell, all of whose primitives and libraries are pure.

Despite their modularity, however, declarative languages are not currently very popular, with no declarative language in the top 10 most popular languages according to either of two language popularity surveys [56, 90]. This is hardly surprising: these languages uniformly lack features expected by users of conventional programming languages, such as variable assignment and iteration. This raises an interesting question: can a language with the comforting familiarity of a conventional programming language also provide the software modularity benefits of a declarative language? What conventional language features must one give up to ensure that a language’s interfaces are comprehensive? We show that, perhaps surprisingly, one need not give up destructive update or looping constructs, nor enforce immutability or defensive copying. We provide a framework for evaluating the modularity of programming languages, and show that while declarative languages naturally provide comprehensive interfaces, imperative languages can also be designed to achieve the same level of modularity.

The remainder of this chapter is organised as follows. In Section 3.2, we formally define interface integrity, our yardstick for evaluating whether languages enforce comprehensive interfaces. In Section 3.3, we discuss in detail which common programming language features are consistent with interface integrity and which are not, and Section 3.4 relates the concept of interface integrity to previous work. We then demonstrate how to apply our definition in Section 3.5, providing the denotational semantics of an example imperative language exhibiting interface integrity. Finally, Section 3.6 concludes.

### 3.2 Interface integrity

We use the word “effect” to describe a change to, or sensitivity to, the state of the computation. The key observation of this chapter is that not all effects are equally troublesome. Procedures that read and update global variables, modify data structures shared between different parts of the computation, or perform input/output, can lead to unexpected consequences for callers. However, the same cannot be said of modifications to local variables, which are invisible to callers.

Therefore, we distinguish between local effects, which cannot affect other procedures,
and global effects, which can. We see the question of whether to allow local effects as largely a matter of style, but the question of whether to allow global effects as a matter of software modularity and robustness. We perceive that the key benefit of declarative languages is the ability to call procedures without causing or observing global effects. This might be called “purity” or “referential transparency,” but we have not found existing definitions satisfactory (see Section 3.4), so we define a new term, “interface integrity,” to capture precisely this notion.

We first define a generalisation of functions that also captures procedures, subroutines, methods, predicates, etc., and define the dual concepts of an effective interface and an apparent interface. This permits our definitions to apply to a wide variety of languages and paradigms with varying abstractions and terminology.

**Definition 3.1.** An execution unit is a part of a program that may be invoked by other program code. It comprises a public declaration and a private implementation, including executable program code. The semantics of an execution unit is defined in terms of the semantics of the execution units and primitives it invokes.

Note that, for the sake of generality, we have not defined exactly what form the declaration of an execution unit must take, but it is intended to mean the “procedure header”: the part of the execution unit that instructs developers on how to invoke it. Conventionally, it would comprise the parameters and return value of a function, but it could also be the parameters of a predicate in a logic language, where the interface does not specify which are inputs and which are outputs. Importantly, we do not consider the declaration to include the text of comments, as these are not able to be checked by a compiler or tool.

**Definition 3.2.** An execution unit’s effective interface is all of the information that:

1. is created outside of the invocation of the execution unit and is usable inside of that invocation (“input”), or
2. is created inside of the invocation of the execution unit and is usable outside of that invocation (“output”).

The effective interface, by definition, captures all of the information that flows into and out of the execution unit in the semantics, whether it is visible in the language syntax or not. For example, any global state read or written by an execution unit is considered part of its effective interface. In a denotational semantics, an execution unit may be expressed as a relation of only the inputs and outputs in its effective interface, with the caveat that the semantics of an execution unit that fatally terminates the program or does not halt is ⊥, and this is not considered part of the interface.
\texttt{\textbf{def}} \texttt{name}(param_1, \ldots, \texttt{param}_n): \\
\texttt{result} = \texttt{name}(\texttt{arg}_1, \ldots, \texttt{arg}_n)

Figure 3.1: Example function declaration and call syntax

\textbf{Definition 3.3.} An execution unit’s \textit{apparent interface} is the subset of its effective interface that is explicitly listed in:

1. the execution unit’s declaration, or

2. (a) (for inputs): the execution unit’s invocation syntax, or
(b) (for outputs): the syntax that handles the return of control from the execution unit.

Clause 2b captures both the normal return of control from the execution unit (e.g., assigning the result to a variable or using it inside a larger expression), and exceptional return of control (e.g., a \texttt{catch} statement).

Figure 3.1 shows the function declaration and call syntax used by examples in this chapter. The apparent interface for execution units in this hypothetical language is underlined. In such a language, the parameters are explicit in both the declaration \textit{and} invocation syntax. The result is not part of the declaration, but if it is not ignored,\(^3\) it is explicitly assigned to a variable or used by the caller in some way.

The concept of “effective interface” represents \textit{all information flow}, while the concept of “apparent interface” represents only the information that is explicit in the language syntax. As observed by Stevens \textit{et al.} \cite{86}, information communicated between caller and callee through a common environment is harder to understand than information passed as parameters. We argue this concept should be extended to cover \textit{any} information flow that circumvents an execution unit’s apparent interface. One very important example of this not discussed by Stevens \textit{et al.} is when a data structure is modified through one pointer, and referenced through others. This can lead to data structures that change for no apparent reason, and is the basis of advice to make objects immutable and to make defensive copies. Data flow through apparent interfaces is simply easier to understand and control than data flow that bypasses them. Our key definition rests on this distinction:

\textbf{Definition 3.4.} An execution unit exhibits \textit{interface integrity} if and only if all inputs and outputs in its effective interface are also in its apparent interface.

This definition excludes all execution units that cause or are affected by global effects. For example, a function that accesses or modifies a global variable (without explicitly

\(^3\)If the result is ignored on all code paths, it is not part of the effective or apparent interface.
3.2. INTERFACE INTEGRITY

declaring so) does not exhibit interface integrity, as the global variable is not part of the function’s apparent interface. More examples are given in Section 3.3.

Definition 3.5. A programming language exhibits interface integrity if and only if:

1. the language allows the definition of execution units, and
2. all execution units definable in the language exhibit interface integrity.

A language with interface integrity guarantees that the programmer is aware of all information flow into and out of each execution unit, which means there can be no surprising communication between two execution units. We therefore assert that this new terminology, along with its definition, describes a desirable property of robust software, and therefore, programming languages.

3.2.1 Applying the definition

Consider a simple purely functional language, where a function’s definition has the form $f(a_1, \ldots, a_n) = e$. In this language, the execution unit is the function. We may model the data flow of functions in this language as in Figure 3.2. The effective interface of any function $f$ is its parameters and its result — this corresponds precisely with the function’s apparent interface, so therefore, this language exhibits interface integrity. We present a formal semantics for such a language in Section 3.5.

Now consider a language such as ML, with shared ref variables [32]. Because use of a ref involves global effects, the effective interface for each function must conceptually include a heap, which maps references to their current values, as an input and output. Expressions that modify a ref update the heap, affecting all aliased references. We may model the data flow of a function in this language as in Figure 3.3. As the heap is not specified in the function definitions or at the invocation site, $H$ is not part of the apparent interface, and so such a language cannot be said to exhibit interface integrity. For similar reasons, the definition of interface integrity also excludes languages with modifiable global variables or arbitrary input/output, as discussed in Section 3.3.

![Figure 3.2: Effective interface (pure)](image-url)
Recall from Section 2.3 that Fortran 95 features a pure keyword that prohibits functions from performing side-effects. However, when we put this keyword to the interface integrity test, it fails: to model a pure function, we must still pass in a read-only heap (for reading mutable variables) and global variable map (for reading global variables). These are part of the function’s effective interface, but not its actual interface. Applying this definition confirms our intuition from Chapter 2 that pure functions do not actually meet the requirements of purely declarative programming.

Similarly, a language with an effect system (for example, Disciple, which allows functions to be annotated with details about which arguments may be mutated, as discussed in Section 2.5) does not qualify for interface integrity. Such a language requires a heap to be passed through the effective interface of all functions, because mutating an argument can potentially affect the value of any other variable. As with Fortran, this scheme cannot be considered purely declarative, because the heap is not mentioned in the actual interfaces of pure functions.

### 3.2.2 Edge cases

Whether or not a language itself exhibits interface integrity is somewhat muddied by the necessity for practical languages to provide a mechanism for calling code written in other languages, to make low-level calls, and take advantage of the wide range of existing software libraries. A language implementation cannot prevent such foreign interface facilities from being used to create execution units without interface integrity.

For example, Haskell’s unsafePerformIO [49], combined with IORef, allows arbitrary mutation of aliased data structures, the same side-effect caused by ML’s ref mechanism [32] (for which we considered ML not to have interface integrity in section 3.2.1). This means that technically, Haskell is no more pure than ML. There is no rigorous objective definition we can apply to distinguish the two languages, because ultimately, execution units in both languages can wreak the same amount of havoc. Yet Haskell and its ilk, with their great modularity benefits, ought to be recognised as more “pure” than languages that freely permit side-effects, in the same way that we recognise Java as a language that provides memory safety, despite containing a mechanism allowing C code to be called.

![Effective interface (with ref)](image)

Figure 3.3: Effective interface (with ref)
Therefore, while we would like to be able to look at a language semantics, apply our definition, and judge whether or not a language has interface integrity, in reality, we must exercise some degree of subjectivity. Hence, the following algorithm should be used to determine whether a language exhibits interface integrity:

1. Apply Definition 3.5 to a given language, to determine whether or not that language exhibits interface integrity. For all non-trivial languages, the answer will be “no,” due at least to foreign function interface features. If the answer is “yes,” stop.

2. Determine the set of aspects of the language that caused it to fail the test of interface integrity. For very pure languages, such as Haskell, this should be just a handful of obscure features, such as `unsafePerformIO` and other foreign function interface features.

3. If all of the aspects in the above are unusual, then the language may be considered to exhibit interface integrity “for all practical purposes.”

For the purposes of this test, unusual refers to language features that are not intended to be used on a regular basis — typically these are features that can be used to break interface integrity, but that are provided for experts with special needs, such as linking to code written in a foreign language. The onus is on users to ensure that all publicly visible execution units written with the feature still exhibit interface integrity. There is an expectation that such features be used responsibly, in harmony with the language semantics and not in spite of it. These features often carry “scary” names (such as “unsafe”), and are often either undocumented, or carry documentation that warns that they should not be used to break the language semantics, or require onerous declarations (e.g., Mercury’s impurity system [38]) designed to discourage careless use.

A language with such a feature cannot pass a formal test of interface integrity, but it can be considered to exhibit interface integrity for the purpose of discussion, modularity and code analysis. Hence, we feel it is appropriate to say that while ML (for example) does not exhibit interface integrity, Haskell (for example), does exhibit interface integrity for all practical purposes.

3.3 Interface integrity and language features

This section presents a tour of a number of key features of programming languages both with and without interface integrity. Many features of conventional programming languages, such as destructive update and by-reference argument passing, may not be used
in languages with interface integrity, but modified versions of these features may be. As we show in this section, and more formally in Section 3.5, while only a small percentage of existing languages fit the definition of interface integrity, the definition does allow for a very broad class of languages, much wider than those which would be classified as “declarative.”

### 3.3.1 Assignment and iteration

One of the most important benefits of guiding language design with interface integrity is that it enables the use of imperative-style constructs, which would not be found in a traditional declarative language. While much of the literature regarding functional programming [25, 48, 52, 61] uses the word “imperative” to mean programming with global effects — a paradigm incompatible with interface integrity — we find the imperative style appealing due to local variable assignment (a *local effect*) and flexible looping constructs.

**Example 3.1** Newton’s method in Haskell (recursive)

```haskell
sqrt n = sqrt' n n
sqrt' n x =
  if abs (x * x - n) > ε
  then sqrt' n (0.5 * (x + n/x))
  else x
```

**Example 3.2** Newton’s method in iterative style

```python
def sqrt(n):
x = n
while abs(x * x - n) > ε:
x = 0.5 * (x + n/x)
return x
```

Example 3.1 is a recursive implementation of Newton’s method for calculating square roots [98] in Haskell. Compare this to Example 3.2, an iterative implementation of the same algorithm in a typical imperative language. This naturally iterative algorithm is expressed clearly in Example 3.2, whereas Example 3.1 essentially uses accumulator recursion to emulate iteration, and requires an additional function to initialise the accumulator. The imperative style is a natural fit for such algorithms, so it is important to us that a language permits this style. Other algorithms can more naturally be written recursively; we wish to promote languages that keep programmers’ options open.

While functional languages often provide looping constructs as higher-order functions, we would argue that they do not allow for true imperative expression of algorithms. We use Haskell for a case study, as it has the “do notation,” which allows a se-
quence of monadic actions that bind variables and update some state. There is a function `forM`, which applies a monadic action to each element in a list. Unfortunately, there is no while loop mechanism, but we shall assume a function `whileM :: Monad m ⇒ m Bool → m a → m ()`, which runs its first argument (the condition), stopping if the condition returns false, and otherwise, running its second argument (the body) and repeating.

We find that, even after adding our own monadic looping construct, Haskell’s `do` notation is a poor substitute for real imperative semantics. Example 3.3 shows our best attempt to implement the imperative version of `sqrt` using Haskell’s ST monad. Note that even though Haskell allows variables to be sequentially bound using `let`, the bindings are forgotten between iterations. Thus, `x_ref` is not an ordinary variable, but an STRef, which represents a reference to a mutable memory location that is updated on each iteration. The fact that we must deal with these special “imperative variables” means the imperative style could not simply be added on as syntactic sugar; it requires a rethink of the language semantics. Faced with this alternative, a Haskell programmer would undoubtedly choose the recursive version, despite being less clear to our eyes than the iterative version in Example 3.2.

Example 3.3 Newton’s method in Haskell (iterative)

```haskell
sqrt n = runST $ do
  x_ref ← newSTRef n
  whileM (
    (readSTRef x_ref >>= λ x → return $ abs (x × x − n) > ϵ)
    (modifySTRef x_ref $ λ x → 0.5 × (x + n/x))
    readSTRef x_ref
  )
```

For comparison, we also implemented this algorithm in Sisal, which was introduced in Section 2.4, and features interface integrity. Example 3.4 shows Newton’s method written in an iterative style in that language, taking advantage of Sisal’s `for initial` sequential loop operator.

Example 3.4 Newton’s method in Sisal (iterative)

```sisal
function sqrt(n : real returns real)
  for initial
    x := n
  while abs(x × x − n) > ϵ repeat
    x := 0.5 × (old x + n/old x)
  returns value of x
end for
end function
```

This version feels a lot more natural than Haskell’s Example 3.3. In fact it is almost
the same as the typical imperative version in Example 3.2; Sisal is a good example of adding imperative style elements to a language without eschewing interface integrity. However, it does not push the imperative style as far as we would like: note that the for initial loop is an expression (not a statement), so the programmer is not able to directly update existing variables inside the body of the loop (only variables initialised in the initial section). It also does not support more advanced control flow constructs such as break or continue, or being able to return from inside a loop, as imperative programmers are used to.

With interface integrity guiding the language design, we can have true imperative languages that provide the same global guarantees as declarative ones.

3.3.2 Field update

As previously discussed, a procedure that modifies (mutates) a data structure, affecting all aliases of that structure cannot be said to have interface integrity. This appears to rule out the convenient “field update” syntax \(x.f = v\) present in most imperative languages. It is possible, however, to support an alternative semantics for the same syntax which, while less familiar to users of most languages, can be considered to be more sensible.

First, let us introduce a non-destructive field replacement operator: \(x.f := v\), which constructs a new object that is a clone of \(x\), but with the field \(f\) replaced with the value \(v\). It does not mutate or overwrite \(x\); essentially this is the functional field update operator found in Haskell [49], or Abadi and Cardelli’s object calculus [2].

Now, for convenience, we introduce the syntactic sugar \(x.f = v\), equivalent to \(x = x.f := v\). This allows us to write natural-looking imperative code that “updates” \(x\), whilst remaining consistent with interface integrity (as only the variable is being changed, not the object it points to). Of course, this does not have the “aliased update” semantics we have come to expect in a conventional imperative language, but that is rather the point of creating a language with interface integrity. The imperative language SETL provides update operators with exactly this semantics: assigning to an element of a tuple, for example, leaves all other aliases of that tuple with their original value intact [22].

Consider Example 3.5, a function in a video game, called when the player fires a rocket. The goal is to create a Rocket object (with the constructor arguments position and velocity, respectively) at the same position as the player, but able to move independently of the player. The rocket’s position is initially copied from the player’s position, but is subsequently updated by re-assigning its \(x\) and \(y\) fields.

Importantly, the object that rocket points to is not being mutated; rather, the local vari-
Example 3.5 Local updates to an object’s fields

```python
def fire_rocket(player, velocity):
    return Rocket(player.position, velocity)

def update_rocket(rocket):
    rocket.position.x = rocket.position.x + rocket.velocity.x
    rocket.position.y = rocket.position.y + rocket.velocity.y
    return rocket
```

able `rocket` is being reassigned. While this semantics may be unfamiliar to some programmers, it is simpler than the semantics we are used to, as it treats all values as values in the mathematical sense. In fact, the above code would have a subtle bug in most imperative languages: the rocket is initialised with the player’s position; updating the rocket’s position would also move the player, since their positions are aliased, as shown in Figure 3.4. With the proposed semantics, the position is just a value, and assigning fields of `rocket.position` updates only the variable `rocket`, and no other.

It should be noted that the above semantics is also potentially much less efficient to implement than simply modifying an object, as typically a copy of the object will be made when it is updated — this is especially troublesome for arrays. However, an implementation can use reference counting to perform a destructive update if an object has no other aliases, improving performance without changing the language semantics. An advanced compiler could leverage the great deal of existing work in the field of aliasing analysis [41, 47, 66] to decide when to perform destructive update at compile time. These techniques are discussed from Chapter 7 onwards.

3.3.3 Object identity

There is a subtle but fundamental difference between languages with interface integrity and languages without: in a conventional language, an object is said to have an identity. This means it is possible to tell whether two equal objects are actually aliased (modifying one causes the other to change), or whether they are separate, but identical, objects.

In a language with interface integrity, the program cannot learn the identity of an object, or whether two objects share an identity: if this were possible, it would mean that

![Diagram](image-url)

Figure 3.4: Unintended aliasing of the player and rocket position
the constructor functions are relying on some hidden state (such as a heap).

Sometimes, it is desirable for objects to have identities. For example, if a Rocket is heat-seeking a Player in a game, and the player moves, the rocket should seek towards the player’s new location — it should remember the player’s identity, not the value of the player’s original position. In a conventional language, the Rocket might simply store a pointer to the Player. In a language with interface integrity, such a pointer would point to a stale version of the Player that never updates. The solution is to assign explicit identities to player objects. Every Player is assigned some kind of ID (such as a unique integer), and the program stores a mapping from ID to Player. The Rocket stores a target player ID, and always moves towards the position of the player corresponding to that ID. This solution is more complex, but it means that identities are explicit and well-understood when required, and objects are simple identity-free values in the default case.

### 3.3.4 Subroutines and call-by-value-result

In a language without destructive update, programmers may face friction when trying to abstract code into reusable functions. In Example 3.5, consider that the caller of `update_rocket` must explicitly store the resulting value back into the game state (otherwise, the update would have no effect). Conversely, in a language allowing mutation, `update_rocket` would simply take a Rocket object (passed by reference) and mutate it.

To achieve the “feel” of being able to pass an object to be updated, without violating interface integrity, we introduce a new kind of execution unit, the subroutine, which has all of its arguments labelled either in or out at both the definition and call site. Subroutines do not return a value; their only effect is to bind out arguments, much like Fortran subroutines, or deterministic predicates in logic languages, such as Prolog and Mercury.

We then add the syntactic sugar of inout arguments, which are equivalent to an in and an out argument of the same name that are always both given the same argument in a call. This allows the subroutine to have one or more arguments whose values conceptually change as a result of execution, but that are not actually mutated — this evaluation strategy is known as call-by-value-result [16]. Hence, subroutines allow a more imperative style by giving the illusion of updating arguments directly. Example 3.6 shows the rocket update subroutine able to conceptually modify the position of a rocket. As with field update, if a rocket passed to `update_rocket` was aliased, the aliases would not be affected.

In an object-oriented language, the same approach can be applied to methods: the Rocket class could have an update method that does not mutate the object, but constructs a new modified object and assigns it back to the variable it was applied to. An inter-
Example 3.6 Subroutine with call-by-value-result parameter

```plaintext
sub update_rocket(inout rocket):
    rocket.position.x = rocket.position.x + rocket.velocity.x
    rocket.position.y = rocket.position.y + rocket.velocity.y
```

esting consequence of a field update semantics that does not actually mutate an object is that mutable objects are no longer taboo. For example, in many languages, strings are immutable to avoid accidental aliased destructive update, but in a language with interface integrity, strings may have mutator methods such as append that update the string without affecting its aliases.

### 3.3.5 Global variables

We have already seen that global variables are forbidden by interface integrity (in accordance with decades of sound advice [86]). Recall, however, that interface integrity merely requires that all information flow be explicit in the syntax. Therefore, a language with interface integrity might allow procedures to read and modify global variables with a special syntax for annotating their use. Example 3.7 shows a procedure, rand, which reads and updates a global variable — such reading and writing is part of the procedure’s effective interface, but thanks to the annotation “uses rand_state,” it is also part of the procedure’s apparent interface, so the procedure has interface integrity.

Importantly, these annotations must be applied recursively. Even though dice_roll does not directly mention rand_state, it calls a procedure that does, so it must also be annotated “uses rand_state.” This should make it clear why annotated global variables do not hurt the modularity of the program: they are similar to inout arguments. They do not need to be explicitly mentioned in the call, but the caller does still need to explicitly acknowledge them. This establishes a contract between the caller and callee — the caller allows the callee to use and update the variable rand_state, but no other global variable. If the language allows concurrency, updates must be thread-local. In this example, if another thread modified rand_state at the same time as rand was executing, there would be data flow that is not part of the procedure’s interface, so this clearly cannot be allowed.

### 3.3.6 Input/output

If an execution unit interacts with the operating system, or calls an execution unit that does, that input and/or output must be considered part of the execution unit’s effective interface. Therefore, a language that allows functions to perform I/O without declaring so in their interfaces does not exhibit interface integrity. Peyton Jones [48] describes
Example 3.7 Procedures annotating use of global variables

```
var rand_state = 123456
def rand(range) uses rand_state:
    rand_state = next_state(rand_state)
    return rand_state mod range

def dice_roll() uses rand_state:
    return rand(6) + rand(6) + 2
```

several approaches to providing controlled I/O in purely declarative languages, including using monads (in Haskell) and state-of-the-world threading (in Mercury and Clean). Merely requiring an annotation, e.g., “uses io,” applied to every procedure that performs input/output directly or indirectly, would also be sufficient to satisfy the definition of interface integrity.

3.3.7 Closures

In Section 3.3.1, it was shown that a language may allow re-assignment of local variables without violating interface integrity. However, this feature, combined with closures, nested procedures that can refer to variables from their enclosing scope, can break interface integrity, depending on their semantics.

Example 3.8 Example of closures

```
def closure_maker():
    x = 7
    f = \( y \rightarrow x + y \)
    x = 12
    return f
def caller():
    g = closure_maker()
    return g(0)
```

In most languages that allow variable re-assignment, notably Scheme [85], but also Python, Scala, and many others, closures contain a reference to the local variable environment of the enclosing procedure, so we call these reference closures. If the environment is modified, a reference closure sees the updated value. Hence, in Example 3.8, even though the closure \( f \) is created while the value of \( x \) is 7, by the end of the procedure, \( x \) has been changed to 12. When the closure \( g \) is subsequently called, it returns the value 12. This does not satisfy interface integrity, as it would allow local variable assignments in the enclosing function to mutate the closure object.\(^4\)

\(^4\)More formally, the current values of the variables in the environment are not stored in the closure object,
The alternative semantics is **value closures**, in which a copy of the necessary variables in the enclosing procedure is taken when the closure is created. Value closures are not sensitive to changes to the variables, and so in Example 3.8, calling the closure $g$ results in 7, the value of $x$ at the time the closure was created. Languages that do not allow variable re-assignment (only re-definition), such as Haskell and ML, naturally support value closures, and so must any language that claims to exhibit interface integrity. The recently added lambda expressions in C++ allow the programmer to choose between reference and value closures [45] (see §5.1.2). An imperative language designer could require a compiler error if a variable is modified after being used in a closure, to make the reference and value semantics equivalent.\(^5\)

### 3.3.8 Exceptions

The fact that a function that throws an exception may be considered to have interface integrity might be somewhat controversial: after all, the act of throwing an unchecked exception constitutes information flow that is not apparent in either the caller or callee. Indeed, the denotational semantics for a language with exceptions typically represents a function’s result as either a value or an exception [48, 51], so clearly, an exception thrown by an execution unit is part of that execution unit’s effective interface.

Perhaps more subtly, it is also considered part of the execution unit’s apparent interface. Recall by Definition 3.3 that the apparent interface includes information in “the syntax that handles the return of control from the execution unit.” In the case of a caught exception, control returns to the **catch** clause, where the function’s output (the exception) is explicitly handled. An uncaught exception, which fatally terminates the program, or a non-terminating execution, does not represent communication between modules and therefore is not relevant to a discussion about interface integrity. This reasoning justifies the exception features in pure languages Haskell [51] and Mercury [38] as exhibiting interface integrity.

But does this model still ensure the modularity benefits we argued for in the introduction? The problem with impure code arises when a caller and callee communicate implicitly. This is not the case with exceptions: when a callee throws an exception, it does not communicate with its caller at all. Hence, from the caller’s perspective, having an exception thrown by a callee is no different than having the callee fatally terminate the program, or enter an infinite loop.

\(^5\)This is the approach taken by Java’s closure-like **inner class** mechanism: any local variables referenced by an inner class must be **final** [28].
3.4 Comparison to related concepts

The term *interface integrity*, introduced by this chapter, has similarities with some existing terms, but other definitions do not quite describe the same concept. Søndergaard and Sestoft [84] define a language as “referentially transparent” if all of its operators are referentially transparent, where an operator is referentially transparent if all arguments’ expressions can be replaced by any other equal in value without changing the operation’s value. This is similar to the concept of interface integrity, but we avoid using this definition for two reasons. Firstly, this definition excludes any operator that depends on its argument’s form, rather than its value, even if the operator is free of effects. For example, Lisp’s *quote* operator, which returns the syntax tree of its argument expression, is not referentially transparent, as it gives different results even for expressions with the same value. While it may be confusing for code’s meaning to depend on its form, it does not compromise the modularity benefits we require for software engineering purposes. Secondly, referential transparency is only defined for languages built exclusively on expressions; it does not directly apply to languages with statements, or to relational languages.

Sabry [80] defines a “purely functional” language as one for which a function produces the same result regardless of whether the evaluation strategy is call-by-name, call-by-need, or call-by-value (if the function terminates successfully). This concept is restricted to functional programming languages, while interface integrity is applicable to any language with some concept of execution units and interfaces. Our definition can also be applied structurally to the semantics of a programming language, as illustrated in Section 3.5, whereas testing a language for purity with Sabry’s definition would require a more rigorous analysis.

The concept of “declarative” programming languages is much older. In a discussion at the end of Landin’s seminal 1966 paper [55], Strachey roughly defines declarative languages as those without assignment statements or jumps. More recently, Lloyd [61] defines declarative programs as logical theories, separating logic from control. There appears to be little agreement about exactly what a “declarative” programming language is. It seems clear that “purely declarative” refers to languages with, at least, the requirement of what we call interface integrity, and this is the interpretation we have used in this thesis. However, most discussions in the literature see imperative features as disqualifying a language from being considered purely declarative. Many consider “imperative” and “declarative” to form a dichotomy.
3.5 Language construction

In Section 3.2, we developed the concept of interface integrity to capture what is necessary to ensure that programs provide the software engineering benefits of maintainability and reusability, and in Section 3.3, argued that a language that provides most of the common features of imperative languages may nonetheless exhibit interface integrity. In this section, we show how our definitions in Section 3.2 can be applied to a denotational semantics to decide whether or not a language exhibits interface integrity. We develop an imperative language step by step, beginning with a purely functional language and extending it with imperative features, showing that each step does not compromise the language’s interface integrity.

We give the full semantics for assignment, iteration and subroutines. The language also includes incremental array operations [92], for more natural imperative programming. For brevity, we do not introduce the field update syntax, annotated global variables and I/O constructs, closures or exceptions — the rationale for including these features in a language with interface integrity is sketched out in Section 3.3. Many of these features are found in Mars, the subject of Chapter 4.

3.5.1 Basic language

We begin with a simple functional language, $L_0$, which has interface integrity, but no imperative features. The language is dynamically typed and, in the interest of simpler semantics, features non-strict evaluation. Execution units in this language are functions. Values in this language are integers ($\mathbb{Z}$) and arrays; functions are not first-class. The body of a function is a single expression, without effects. Primitive functions (most with special syntax) include operators for basic arithmetic ($+, -, \times$), comparison\(^6\) ($=, \neq, <, \leq, >, \geq$), conditions ($c ? t : f$), array construction ($\text{array}(\text{len}, \text{val})$), indexing ($a[i]$) and replacement\(^7\) ($a[i \mapsto v]$). All exhibit interface integrity. A program in $L_0$ is shown in Example 3.9.

**Example 3.9 Example of $L_0$**

```plaintext
\textbf{def factorial}(n):
\textbf{return} \ (n === 0) \ ? \ 1 : (n \times \text{factorial}(n - 1))
```

The denotational semantics for $L_0$ is given in Figure 3.5. A brief explanation follows. The value domain, Val, is ordered by information content; that is, $\bot$ is smaller than everything else in Val, arrays of equal length are ordered according to the order of their corresponding elements, and other unequal values are incomparable. Thus Val is a com-

\(^6\)Boolean values are simply 0 for false and 1 for true.

\(^7\)This produces a new array like $a$ except that $i$ is mapped to $v$, without modifying the array $a$. 
Abstract syntax
Let Var be the set of all variable names and FuncName be the set of all function names.

\[
\begin{align*}
\text{Lit} & \rightarrow \cdots | -2 | -1 | 0 | 1 | 2 | \cdots \\
\text{Expr} & \rightarrow \text{Lit} \mid \text{Var} \\
& \mid \text{FuncName(Expr*)} \\
\text{FuncDef} & \rightarrow \text{def FuncName(Var*): return Expr} \\
\text{Program} & \rightarrow \text{FuncDef*}
\end{align*}
\]

Domains

\[
\begin{align*}
\text{Val} & = \{\bot\} \cup \mathbb{Z} \cup \text{Val}^* \\
\text{Env} & = \text{Var} \rightarrow \text{Val} \\
\text{Den} & = \text{FuncName} \rightarrow (\text{Val}^* \rightarrow \text{Val})
\end{align*}
\]

Semantic functions

\[
\begin{align*}
\Theta & : \text{Den} \\
\mathcal{E} & : \text{Expr} \rightarrow \text{Den} \rightarrow \text{Env} \rightarrow \text{Val} \\
\mathcal{F} & : \text{FuncDef} \rightarrow \text{Den} \rightarrow \text{Den} \\
\mathcal{P} & : \text{Program} \rightarrow \text{Den}
\end{align*}
\]

\[
\begin{align*}
\Theta & = \{f \mapsto \text{semantics of } f : \\
& f \text{ is the name of a primitive function}\}
\end{align*}
\]

\[
\begin{align*}
\mathcal{E}[n] &\rho\sigma = n, \text{ if } n \in \text{Lit} \\
\mathcal{E}[v] &\rho\sigma = \sigma v, \text{ if } v \in \text{Var} \\
\mathcal{E}[f(a_1, \ldots, a_n)] &\rho\sigma = \mathfrak{p} f \left( \mathcal{E}[a_1] \rho\sigma', \ldots, \mathcal{E}[a_n] \rho\sigma' \right) \\
\mathcal{F}[f(a_1, \ldots, a_n) e] &\rho = \{ f \mapsto \lambda (x_1, \ldots, x_n). \} \\
& \begin{array}{l}
\text{let } \sigma = \{ a_1 \mapsto x_1, \ldots, a_n \mapsto x_n \} \\
\text{in } \mathcal{E}[e] \rho\sigma'
\end{array}
\end{align*}
\]

\[
\begin{align*}
\mathcal{P}[p] & = \text{lfp} \left( \bigcup_{f \in \mathfrak{p}} (\mathcal{F} f) \sqcup \lambda \rho. \Theta \right)
\end{align*}
\]

Figure 3.5: The denotational semantics for $L_0$
plete partial order. Functions are ordered pointwise, so the least upper bound (⊔) of two disjoint mapping functions is the union of their key/value pairs.

A Den is the semantics of a program, representing the semantics of the source language functions as mathematical functions. Θ is a Den containing the semantics of all of the primitive functions in \( L_0 \). The semantics of the program itself (defined by \( P \)) is the least fixed point (lfp) of the least upper bound of the semantics of all function definitions in the program. Since \( \bigsqcup_{f \in P}(F_f) \) is monotonic, its least fixed point is well-defined.

An Env is a mapping from variable names to values, readable by all expressions. Importantly, this is a local environment — it is not passed through function calls — this fact should hint at the language’s interface integrity. The notation \( \bar{\rho} \) means \( \rho \), but with any undefined values replaced with \( \bot \).

The semantics of the function evaluates the function’s body expression (with the evaluation function \( \mathcal{E} \)) with respect to the global Den and a new local Env mapping formal parameter names to actual parameter values.

We now apply each definition of Section 3.2 to \( L_0 \):

1. The execution unit for \( L_0 \) is the function. The semantics of an execution unit \( f \) is the image of \( f \) under \( F \). The declaration syntax is the first line of FuncDef: the function name and the name of its parameters.

2. The effective interface of a function is the value of all of the arguments passed to its semantics \( (x_1, \ldots, x_n) \) and the result of its semantics \( \mathcal{E}[e][\rho(\sigma)] \).

3. The apparent interface of a function is the underlined syntactic elements in Figure 3.1: the function’s arguments and its result.

4. For any function, the arguments and result of its effective interface are also part of its apparent interface. Hence, all functions exhibit interface integrity.

5. Therefore, \( L_0 \) exhibits interface integrity.

### 3.5.2 Sequencing and assignment

We now introduce statements to the language. The new language, \( L_1 \), features a single type of statement: assignment. Example 3.10 shows a program in \( L_1 \). Variables may be assigned and re-assigned. The return is always required after the last statement, to simplify the semantics.

The denotational semantics for \( L_1 \) is given in Figure 3.6. Although the semantics of a statement accepts and updates a local environment \( (\sigma) \), note that the semantics of a
Example 3.10 Example of $L_1$

```python
def f(x):
a = x
b = a + 2
a = b - 3
return a
```

Abstract syntax

| Stmt   | $\rightarrow$ | $\epsilon$
|--------|----------------|
|        |                | Stmt ; Stmt
|        |                | Var = Expr

| FuncDef | $\rightarrow$ | def FuncName(Var*):
|---------|----------------|
|         |                | Stmt
|         |                | return Expr

Semantic functions

$S: Stmt \rightarrow Den \rightarrow Env \rightarrow Env$

$S[\epsilon]_\rho = id$

$S[s_1 \; s_2]_\rho = (S[s_2]_\rho) \circ (S[s_1]_\rho)$

$S[v = e]_\rho = \lambda \sigma. \sigma[v \mapsto E[e]_\rho \sigma]$

$F[f(a_1, \ldots, a_n) \; s \; e]_\rho = \{ f \mapsto \lambda(x_1, \ldots, x_n).
\text{let } \sigma = \{a_1 \mapsto x_1, \ldots, a_n \mapsto x_n\}
\text{in } E[e]_\rho (S[s]_\rho \sigma)\}$

Figure 3.6: Additions to the denotational semantics for $L_1$

function (its effective interface) still has the same inputs and outputs as in $L_0$, so $L_1$ also exhibits interface integrity. This is intuitive, as statements in $L_1$ have only local effects.

While $L_1$ lacks iteration, as we shall see in $L_2$, it is quite simple to add it. This is because the semantics of $L_1$ treats variable assignment as an update to the local state, as opposed to the semantics of nested let bindings in ML, Haskell and Sisal, which create a nested scope that cannot outlive a statement block.

3.5.3 Selection and iteration

We now introduce language $L_2$, which adds selection (if-then and if-then-else statements), and iteration (while statements).

Unlike in a language where bindings create a new scope, assignments in $L_2$ remain

---

8The semantics of if $e$ then $s$ endif is not shown, but is equivalent to if $e$ then $s$ else $e$ endif.
3.5. LANGUAGE CONSTRUCTION

Example 3.11 Example of $L_2$

```python
def fibonacci(n):
    cur = 0; next = 1; i = 0
    while i < n:
        prev = cur
        cur = next
        next = prev + cur
        i = i + 1
    return cur
```

Abstract syntax

\[
\text{Stmt} \rightarrow \epsilon \\
| \text{Stmt} ; \text{Stmt} \\
| \text{Var} = \text{Expr} \\
| \text{if} \text{Expr} \text{Stmt} \text{else} : \text{Stmt} \\
| \text{while} \text{Expr} \text{Stmt}
\]

Semantic functions

\[
S[\text{if} \ e : s_{\text{then}} \text{else} : s_{\text{else}}] \rho = \\
\lambda \sigma. \begin{cases} 
S[s_{\text{then}}] \rho \sigma, & \text{if } E[e] \rho \sigma \neq 0 \\
S[s_{\text{else}}] \rho \sigma, & \text{otherwise} 
\end{cases}
\]

\[
S[\text{while} \ e : s] \rho = \\
\text{lfp } \lambda h. \lambda \sigma. \begin{cases} 
\text{h } (S[s] \rho \sigma), & \text{if } E[e] \rho \sigma \neq 0 \\
\sigma, & \text{otherwise} 
\end{cases}
\]

Figure 3.7: Additions to the denotational semantics for $L_2$

past the end of an if or while block, and persist across iterations of a while loop. This allows imperative algorithms to be easily expressed, as in Example 3.11. Note that this means that a variable can be assigned on some code paths and not others; ideally the compiler would ensure that a variable cannot be read from unless it is definitely assigned on all code paths.

The denotational semantics for $L_2$ is given in Figure 3.7. The function semantics ($F$) has not changed, so the language still exhibits interface integrity, yet it is also recognisable as an imperative language.

3.5.4 Subroutines and output arguments

We introduce the language $L_3$, with subroutines, introduced informally in Section 3.3.4. Recall that subroutine parameters are labelled either in, out or inout. A subroutine in $L_3$ that changes its argument is the handy array_update subroutine, shown in Example 3.12. As with the update_rocket subroutine shown in Example 3.6, this gives the illusion of
CHAPTER 3. INTERFACE INTEGRITY

Example 3.12 Example of $L_3$

\[
\text{sub array_update(inout } a, \text{ in } i, \text{ in } v):
\]
\[
a = a[i \mapsto v]
\]

Abstract syntax
Let SubName be the set of all subroutine names.

\[
\begin{align*}
\text{Stmt} & \rightarrow \text{SubName}((\text{in } \text{Expr})^*, (\text{out } \text{Var})^*) \\
\text{SubDef} & \rightarrow \text{sub SubName}((\text{in } \text{Var})^*, (\text{out } \text{Var})^*) \\
\text{Program} & \rightarrow (\text{FuncDef} | \text{SubDef})^*
\end{align*}
\]

Domains

\[
\text{Den} = (\text{FuncName} \cup \text{SubName}) \rightarrow (\text{Val}^* \rightarrow \text{Val})
\]

Semantic functions

\[
\begin{align*}
B : \text{SubDef} & \rightarrow \text{Den} \rightarrow \text{Den} \\
\mathcal{S}[f(\text{in } i_1, \ldots, \text{in } i_m, \text{out } o_1, \ldots, \text{out } o_n)]\rho\sigma = \\
\text{let } r = \overline{p} f (\mathcal{E}[\text{in } i_1]\rho\sigma, \ldots, \mathcal{E}[\text{in } i_m]\rho\sigma) \\
\text{in } \sigma[o_i \mapsto r \text{ for } 1 \leq i \leq n]
\end{align*}
\]

\[
B[f(\text{in } i_1, \ldots, \text{in } i_m, \text{out } o_1, \ldots, \text{out } o_n) s]\rho = \\
\{ f \mapsto \lambda x_1 \ldots x_m. \\
\text{let } \sigma = (\mathcal{S}[s] \rho \{ i_1 \mapsto x_1, \ldots, i_m \mapsto x_m \}) \\
\text{in } \langle \sigma o_1, \ldots, \sigma o_n \rangle
\}
\]

\[
\mathcal{P}[p] = \text{lfp} \left( \bigcup_{f \in p} \left\{ \begin{array}{ll}
\mathcal{F} f, & \text{if } f \in \text{FuncDef} \\
B f, & \text{if } f \in \text{SubDef}
\end{array} \right\} \cup \lambda. \Theta \right)
\]

Figure 3.8: Additions to the denotational semantics for $L_3$

updating a data structure, but cannot make any changes the caller is unaware of.

The denotational semantics for $L_3$ is given in Figure 3.8. Note that \text{in} and \text{out} arguments can be freely interleaved; for notational convenience, the semantics assumes that all \text{ins} come before \text{outs}. The \text{inout} notation is omitted for brevity. We apply each definition of Section 3.2 to $L_3$’s subroutines:

1. The new execution unit for $L_3$ is the subroutine. The semantics of an execution unit $f$ is the function mapped from $f$ in the definition of $B$. The declaration syntax is the first line of SubDef, the subroutine name and the name and mode of its parameters.

2. The effective interface of a subroutine is the values of all of the arguments passed to
3.6. CONCLUSION

In this chapter, we argued that programs in which procedures communicate via explicit channels such as parameter passing will exhibit far greater modularity than languages that allow unfettered side-effects. This is an important prerequisite for maintainability and reusability. Given these advantages, languages should be designed that, like the purely functional Haskell, enforce this restricted communication, and design techniques for such languages should be refined.

We proposed a new term, “interface integrity,” which precisely captures the operations whose semantic information flow is reflected in their syntactic interface. These are the operations that can be composed without danger of unexpected interactions. This definition serves as a framework for evaluating language designs: by testing whether a language has interface integrity, we can predict how well it will encourage modular programs and a modular ecosystem.

Languages with interface integrity offer clear software engineering benefits: loose coupling and composable execution units, as well as more precise analysis and optimisation. An important contribution of this chapter is to show that key imperative constructs, which often provide a more familiar and convenient notation than recursion, are compatible with interface integrity, because interface integrity only forbids global effects, while these imperative constructs only require local effects. To prototype the ideas of this chapter, we have designed and implemented a new imperative language with interface integrity, Mars, which is described in detail in the next chapter.
Chapter 4

Mars: An imperative/declarative language

4.1 Introduction

In the previous chapter, the case was made for a class of programming languages that exhibit interface integrity (that is, do not permit execution units to cause or observe global effects), but still allow local imperative programming. To this end, we have designed and implemented a small prototype language, Mars. In this chapter, we describe the Mars language, and discuss the philosophy behind its design.

Mars is a very simple imperative programming language that exhibits interface integrity, as given in Definition 3.5. Mars has two types of execution unit: functions, and computable global constants, which are described later. Collectively, these are referred to as procedures. Within a procedure, programming is largely the same as a typical high-level imperative programming language, with the body being a sequence of statements including assignment statements, conditionals, and while loops. However, while Mars allows local variable assignment, it prohibits any sort of global effect. In particular, it does not allow:

- mutation of objects (updating an object creates a shallow copy);
- assignment to global variables (only global constants are allowed); and
- interaction with the operating system, except under controlled circumstances (input/output operations require a special annotation).

Mars also has some other nice features borrowed from functional programming: a
CHAPTER 4. MARS: AN IMPERATIVE/DECLARATIVE LANGUAGE

strong static type system, algebraic data types, pattern-matching switch statements, and
higher-order functions.

The Mars programming language is formally specified in The Mars Language Refer-
ence [27]. This chapter contains a summary of the language reference, tailored for the
version of Mars presented in this thesis.

We have fully implemented a compiler and interpreter for the language. The comp-
piler was implemented in Mercury, and its source code is publicly available. The toolkit
compiles Mars source code into an intermediate representation known as MIR, and then
has two options for executing the program: it can directly interpret the MIR code, or it
can compile MIR to LLVM bitcode [57], which can be heavily optimised and converted
into a native binary.

Mars source files are plain ASCII text files with the .mar extension. In this thesis,
we follow the conventions established in Section 1.4, formatting source code in mathe-
matical notation instead of plain text. As such, the syntax given in this chapter is also in
mathematical notation. For the actual ASCII syntax, see the official language reference.

The language includes a set of built-in types and functions, comprising only those
which could not be implemented in the language itself. In addition to the built-ins, a stan-
dard library has been written in pure Mars, and is available for all programs to use. The
main module of the standard library, comprising the most useful operations, is named
the "prelude," after the equivalent module in Haskell, and ALGOL before it.

The remainder of this chapter is organised as follows. In Section 4.2, we give a sum-
mary of the Mars Language Reference. In Section 4.3, we give a summary of the Mars
Library Reference. In Section 4.4, we present and discuss an example of a Mars program.
Finally, Section 4.5 concludes.

4.2 Overview

To begin our exploration of Mars, the two factorial programs from Chapter 1 are repeated
here, in Example 4.1. These examples demonstrate the basic Mars syntax, and highlight
that programs can be written in either the recursive or iterative styles. Again, note that
the actual syntax is slightly different from that presented here (e.g., in real Mars source
code, \# is written as `!=` and \# is written as `*`).

The Mars syntax is deliberately derived from that of Python [10], with procedure and
statement blocks being delimited by indentation, rather than symbolic delimiters such

\[1\]Mars is licensed under the GNU General Public License version 3, and can be downloaded from
4.2. OVERVIEW

Example 4.1 Factorial function in Mars (recursive and iterative versions)

```python
def factorial(n :: Num) :: Num:
    if n == 0:
        return 1
    else:
        return n × factorial(n − 1)
```

```python
def factorial(n :: Num) :: Num:
    acc = 1
    while n ≠ 0:
        acc = acc × n
        n = n − 1
    return acc
```

as { and }, or begin and end keywords. For example, the while block in Example 4.1b encompasses the acc = ... and n = ... statements, which are indented more than the while statement itself. The subsequent return statement is indented the same as the while statement, so it is not part of the loop body.

In terms of the languages described in Chapter 3, Mars is a more feature-complete version of L₂ (it lacks the subroutine feature of L₃). As with L₂, Mars features sequences of statements which may modify the local state of the procedure, primarily by writing to local variables such as acc and n. Control structures such as if, switch and while are used to determine which statements to execute, and how many times.

Mars distinguishes expressions from statements. Expressions cannot have any side-effects, which means that evaluation order within an expression is irrelevant,² and the condition of an if or while block cannot modify the local state. The body of a Mars procedure is a statement (as in C or Python), not an expression (as in ML or Haskell).

Mars is strongly and statically typed. All variables and expressions have a (possibly polymorphic) type at compile time. In a procedure definition, all arguments and the procedure’s return value must have an explicit type annotation. An annotation of the form x :: a means that x has type a; for example, the factorial function accepts and returns a value of type Num (a floating-point number).

Variables may optionally be declared with a type annotation. In Example 4.1b, it would be valid to state:

```python
var acc :: Num
```

It would then be a compile-time type error to assign a non-numeric value to acc. If a variable does not have an explicit type annotation, its static type is usually inferred from its first assignment statement. For example, the line acc = 1 gives acc the type Num, and it would be a type error to subsequently assign a differently typed value to that variable.

Mars has a full Hindley-Milner type inference algorithm, so if a type cannot be fully inferred from the first assignment (e.g., x = [] infers x :: Array(a), but does not know the

---

²As an exception discussed in detail later, expressions in an I/O context may have side-effects, and there are a number of “unofficial” impure features in the language which may exhibit side-effects. The evaluation order is important with respect to these features.
CHAPTER 4. MARS: AN IMPERATIVE/DECLARATIVE LANGUAGE

... element type), it may be inferred from the rest of the function definition.

Variable scoping is deliberately kept simple — all variables are scoped to the entire function body (as in Python). It is a compile-time error if a variable is read from without first being definitely assigned on all code paths.

4.2.1 Type system and algebraic data types

Mars is a strong statically typed language. The Mars type system is based on the parametric Hindley-Milner [40, 71] type system of ML. All variables and expressions have a known type at compile time. Mars requires explicit type declarations for function argument and return types, but can infer the types of local variables. As the type system is not the focus of this thesis, we do not give a formal description here; that can be found in the language reference [27].

Each type in Mars is either a data type or a type constructor. A data type is a type that represents a set of values; variables, arguments, and return values must have data types. A type constructor is a function at the type level; it accepts one or more data types as arguments, producing a data type. The kind of a type describes whether a type is a data type or type constructor, and how many arguments it accepts. For example, the built-in type Num is a data type, having kind \( \ast \). The built-in type Array is a type constructor, having kind \( \ast \rightarrow \ast \). This means that it accepts one type argument, for example, Array(Num), which is a data type representing the set of arrays of numbers.

Mars features three built-in types:\(^3\)

- Num (having kind \( \ast \)) — real numbers (represented by double-precision binary floats).

- Array (having kind \( \ast \rightarrow \ast \)) — contiguous sequences of homogeneously typed Mars values. Values of type Array\( a \) hold zero or more values of type \( a \), and offer constant-time read access to individual elements. Because Mars prohibits mutation of objects, appending, removing or replacing an element costs linear time in a naïve implementation of Mars.\(^4\)

- \( \rightarrow \) (having kind \( \ast, \ast \ast \rightarrow \ast \)) — closures, or first-class function values. This type constructor takes one or more type arguments; the type \( (a, \ldots, z) \rightarrow r \) represents functions that accept arguments of types \( a, \ldots, z \) and return a value of type \( r \). (Note that in our notation, the kind \( \ast \ast \) represents a sequence of zero or more types.)

\(^3\)As a practical matter, we omitted types such as Int or Bool that were not strictly necessary, since Mars is intended as a prototype language, not for industrial use.

\(^4\)Achieving a constant-time implementation of array update operations is the subject of much of the later part of this thesis.
Mars features type polymorphism: whenever a data type is expected, a type variable may be given instead. Type variables, beginning with a lowercase letter, may be substituted for any concrete data type. This is analogous to generics in Java or type templates in C++. For example, the array_head function in Example 4.2 gets the first element of an array, regardless of the element type. The type variable \( a \) allows for values of any data type, but all such values must have the same type.

**Example 4.2** Polymorphic types in Mars

```python
def array_head(array :: Array(a)) :: a:
    return array_ref(array, 0)
```

Mars does not have type classes. However, all types support equality comparison and string representation, via the special built-in functions `eq :: a \to a \to \text{Num}` and `show :: a \to \text{Array(Num)}` (the string type in Mars). The implementation of equality and string representation is automatically generated for user-defined types, and cannot be overridden. Comparing function types for equality results in a runtime error.

It is possible for a program to introduce new types via type declarations. User-defined types are algebraic data types: each type consists of one or more constructors, with each constructor having a name and zero or more parameters. Each value of the type belongs to exactly one of the type's constructors, and has a member for each parameter of that constructor. Example 4.3 shows a user-defined type that allows for the specification of addition and subtraction expressions. This type has three constructors, each of which provides a function of the same name. For example, a call to `Literal :: Num \to Expr` produces a value representing a number literal expression.

**Example 4.3** An additive expression type

```python
type Expr:
    Literal(Num)
    Add(Expr, Expr)
    Subtract(Expr, Expr)
```

User-defined types may also be polymorphic. The standard prelude defines the List type, given in Example 4.4. This allows for the construction of homogeneous linked lists of any Mars type \( a \). A linked list is either Nil, the empty list, or Cons, with a head of type \( a \) and a tail of type \( \text{List}(a) \). This example also shows that constructor parameters may optionally have names, which may be used to retrieve the value of the named field.

The `switch` statement is used to both determine which constructor a particular value belongs to, and extract field values from a parent value. Each `switch` has a single control value, and one or more `case` statements. Each `case` statement has a pattern, which is
Example 4.4 The standard linked list type

```haskell
    type List(a):
    Cons(head :: a, tail :: List(a))
    Nil
```

an expression consisting of constructor names, numeric literals and variables. A control value matches a pattern if it unifies with the pattern expression. The switch statement chooses the first case statement that matches the control value, binds any variables in the pattern to the corresponding parts of the control value, then executes the body of the case statement. The variable binding allows fields to be extracted from a value. Example 4.5 defines a function that evaluates an Expr value representing an expression. Note that the variables \( n, e_1 \) and \( e_2 \) are bound by the case statements before entering the body.

Example 4.5 Expression evaluation using a switch statement

```haskell
    def eval(e :: Expr) :: Num:
        switch e:
            case Literal(n):
                return n
            case Add(e1, e2):
                return eval(e1) + eval(e2)
            case Subtract(e1, e2):
                return eval(e1) - eval(e2)
```

4.2.2 Field access operators

As shown in the previous section, fields may optionally be given names. The name of a field may be used for reading and updating without the use of switch statements. This feature is similar to Haskell’s field labels [49], but with one key difference: Mars field names do not generate top-level functions, so they do not have to have globally unique names. Field access uses the object.field syntax common to C-derived languages.

This is primarily designed for use with record types, which have just a single constructor. In Example 4.6, a Vector3 has one constructor, with fields representing the components of a three-dimensional vector.

Example 4.6 A three-dimensional vector type

```haskell
    type Vector3:
    Vector3(x :: Num, y :: Num, z :: Num)
```

A variable \( v :: \) Vector3 may have its fields read using the switch statement, but it is easier to use the field access syntax. For example, the expression \( v.x \) gets the value of
the vector’s X coordinate. This is known as the \textit{field-reference} operator. Similarly, fields of a value may be non-destructively updated: the expression \( v.x := k \) (equivalent to \texttt{Vector3}(k, v.y, v.z)) creates a new \texttt{Vector3} with the X coordinate \( k \) and the same Y and Z coordinates as \( v \). This is known as the \textit{field-replace} operator.\footnote{Unlike in Haskell, there is no syntax to update multiple fields simultaneously, although field-replace operators may be chained. \textit{e.g.}, \( (v.x := i).y := j \).} It is a runtime error to access a field if the constructor used to build a value does not contain a field of that name.

As in Haskell [49], a field name may be used in multiple constructors of the same type. It is a compiler error if all fields of the same name do not share the same type. This can be used to add a field that is independent of the constructor. Example 4.7 redefines the \texttt{Expr} type, with field names, and an extra \texttt{lineno} field added to each constructor, to record the line number that each expression appeared on in a source file. Now, for any given expression, \( e.lineno \) refers to the line number, and \( e.lineno := i \) updates the line number. Similarly, \( op_1 \) and \( op_2 \) can be used to access the operands of an addition or subtraction expression, but it is a runtime error to attempt to access \( op_1 \) or \( op_2 \) of a number literal. We refer to the concept of overloading field names onto multiple constructors as “smart field access.”

\begin{example}
The \texttt{Expr} type with line numbers
\begin{verbatim}
type Expr:
    Literal(lineno :: Num, value :: Num)
    Add(lineno :: Num, op_1 :: Expr, op_2 :: Expr)
    Subtract(lineno :: Num, op_1 :: Expr, op_2 :: Expr)
\end{verbatim}
\end{example}

4.2.3 Arrays and strings

Mars features arrays as a primitive type, with special array literal syntax, and several built-in functions to create and manipulate arrays (it is not possible to create arbitrary-length values with constant-time access using user-defined types in Mars). We felt that it was important to focus on arrays, as imperative programming often consists of array manipulation.

The array literal expression, \([e_1, \ldots, e_n]\), creates an array of length \( n \), from the values of the enclosed expressions. An array can also be created with the built-in function \texttt{array} :: (Num, a) \to Array(a), which creates an array of the given length, with all elements set to the given \( a \) value. The built-in function \texttt{array_ref} :: (Array(a), Num) \to a retrieves an element from an array. The built-in function \texttt{array_replace} :: (Array(a), Num, a) \to Array(a) copies the input array, producing a new array with a single element replaced with the value of the third argument. Similarly, \texttt{array_concat} :: (Array(a), Array(a)) \to
Array\(a\) produces a new array which is the concatenation of the two input arrays. The other array-related built-ins are array\_add, which non-destructively adds a single element to the end of an array, array\_remove, which non-destructively removes a given element from an array, and array\_length, which returns the length of an array.

Of course, using array\_replace and array\_concat to repeatedly update an array is extremely inefficient in a naïve implementation of the language. As we want imperative programmers to be able to iteratively update an array, we have spent a good deal of effort optimising these operations so that, in certain cases, they do actually mutate the input array, without allocating any memory or copying any values. This optimisation does not affect the semantics of the program; it is discussed from Chapter 8 onwards.

To avoid an abundance of built-in types, Mars does not have a primitive string or character type. Instead, strings are represented by the type Array(Num), where each number represents a byte value. The meaning of the byte values (specifically, the character encoding that maps byte sequences onto Unicode code points) is at the discretion of the program author. Mars itself is not aware of Unicode or any character encodings.

Mars provides string and character literal notations, allowing string and character constants to be specified in a readable manner. This notation is conventional, with character literals being surrounded in single quotes (e.g., ‘\’x\’) and string literals being surrounded in double quotes (e.g., "Hello, world!"). The usual C escape sequences, such as ‘\n’ for newlines and ‘\xnn’ for arbitrary byte values, is available. To avoid dealing with Unicode, Mars only allows ASCII characters in string and character literals; non-ASCII characters must be specified using byte values. For example, the text “π is approximately 3” can be represented in UTF-8 “\xcf\x80 is approximately 3”.

### 4.2.4 Closures and computable global constants

Functions in Mars are first-class values. A closure is a value that represents a callable function. A function name is a valid expression on its own; its value is a closure that will execute the named function when called. There is no special “reification” syntax for creating a closure from a function.

A variable with a function type may be called using conventional parenthesis syntax; for example, \(f(x)\) calls the function \(f\) with \(x\) as a parameter. This allows us to define functions that operate on other functions; these are known as higher-order functions. Example 4.8 shows the standard prelude implementation of the \(\text{foldl}\) function, which accepts a function \(f\) as a parameter and calls it numerous times.

The \(\text{foldl}\) function may be called with any function as a parameter. For example, the
Example 4.8 A higher-order function

```python
def foldl(f :: (a, b) → a, z :: a, l :: List(b)) :: a:
    switch l:
        case Nil:
            return z
        case Cons(x, xs):
            return foldl(f, f(z, x), xs)
```

expression \( \text{foldl}(\text{add}, 0, \text{nums}) \) computes the sum of the list \( \text{nums} \) by repeatedly calling \( \text{add} \) (a built-in function) on each of its elements. In this case, \( \text{add} \) is treated as a closure, instead of being called directly.

Closures may also hold other values. Mars does not currently feature nested functions or lambda expressions (which are the usual approach for storing data in a closure), but it does feature partial application as a language primitive. If a function call ends in an ellipsis ("\ldots"), it is considered to be a partial application — a new closure is created that holds all of the supplied arguments. When called, this closure accepts the remaining arguments and calls the original function with all of the arguments.\(^6\) For example, the expression \( \text{add}(1, \ldots) \) has type \( \text{Num} \rightarrow \text{Num} \); it is a function that accepts a single number and adds 1 to it. This may be used, for example, in the expression \( \text{map}(\text{add}(1, \ldots), \text{nums}) \), to increment every number in a list.

The explicit \("\ldots"\) notation was chosen for a specific software engineering reason: if a new parameter is added to the end of a function’s signature, all regular calls to that function become compiler errors, allowing the programmer to identify and correct those calls. Most other functional languages either treat all function definitions as being in curried form (e.g., Haskell and ML) or automatically perform partial application if fewer than the required number of arguments are passed to a function (e.g., Mercury). In these languages, all such calls become partial applications, resulting in confusion, and potentially, a program that is undesirably accepted by the compiler.

To further promote a functional style, Mars introduces a second type of procedure called a *computable global constant* (CGC). Although Mars does not allow global variables, it is still sound to allow global constants. Instead of strictly defining what sort of expression might constitute a “constant,” we opted to let the programmer give a name to the result of any valid Mars computation. As such, a computable global constant is a procedure with no parameters. Because Mars procedures cannot observe side-effects, a

\(^6\)Note that POP-2 features a similar explicit partial application operator, while both Python and C++ include library support for explicit partial application (\texttt{functools.partial} and std :: \texttt{bind}, respectively). Eiffel features an advanced version where each unsupplied parameter is explicitly left “open”, as opposed to partial application in Mars and the other languages, which leaves all remaining parameters open.
CGC always computes the same result, so the implementation only computes the result the first time it is requested, and caches it for subsequent use. An expression consisting of the name of a CGC is not a closure; it is the value of the CGC. CGCs are often used to define a function in point-free style, such as in Example 4.9. Note that sum is not a function: it is a CGC whose value is a partially applied closure.

**Example 4.9** A computable global constant

```haskell
def sum :: List(Num) \rightarrow Num:
    return foldl(add, 0, ...)
```

A computable global constant is syntactically and operationally different to a zero-arity function. Syntactically, a zero-arity function requires parentheses to compute the result (e.g., \( f() \)); mentioning the function name without parentheses gives a zero-arity closure (for example, \( f \) has type \( () \rightarrow a \), where \( a \) is the return type of \( f \)). Operationally, a zero-arity function is re-computed each time it is called, whereas a computable global constant is computed once and cached. Lastly, zero-arity functions may perform input/output, as described in the next section, whereas computable global constants may not.

### 4.2.5 Input/output

Every program needs to interact with the operating system in order to be useful. At the simplest level, this involves sending and receiving data from the standard input and output streams, but this can also involve reading and writing from the file system, creation and manipulation of user interfaces, network communications, and so on.

As with other declarative languages, a mechanism is required to prevent these inherently effectful operations from breaking the interface integrity of the language. Several mechanisms are discussed in Section 3.3.6, but Mars uses a very simple one: the keyword `io` in a function definition signals that the function is allowed to perform input/output side-effects. Example 4.10 shows a basic “Hello world” program, which declares itself to be `io` in order to display text to the user via the standard output stream.

As in C, the main function is the entry-point to a Mars application, and always has type \( () \rightarrow \text{io} \text{Num} \). In this case, the main function calls the `print_string` function, which also has the `io` annotation. The `print_string` function, which is included in the Mars prelude module, calls the built-in `put_char` function, which has type `Num \rightarrow \text{io} \text{Num}`, and performs a side-effect of writing a byte to standard output. It is a compiler error to call an `io` function from a non-`io` function, so therefore, `print_string` and `main` must have the `io` annotation. This rule ensures that any function without the `io` annotation does not exhibit side-effects.
Example 4.10 Hello world program in Mars, with print_string function

```python
def main() :: io Num:
    print_string("Hello, world!\n")
    return 0

def print_string(str :: Array(Num)) :: io Num:
    i = 0
    while i < array_length(str):
        put_char(array_ref(str, i))
        i = i + 1
    return 0
```

Semantically, input/output effects in Mars are viewed as passing a hidden “world” object as input to the function, modifying the world in some way (such as adding a byte to the standard output stream), and then returning the modified world object as a hidden output. This hidden input and output form part of the function’s effective interface, as specified in Definition 3.2. The `io` annotation is viewed as a syntactic short-hand for specifying the world object as both an input and output to the function, forming part of the function’s apparent interface, as specified in Definition 3.3. Therefore, by forcing all functions which directly or indirectly produce input/output effects to declare themselves as “`io`,” Mars satisfies Definition 3.5 for interface integrity.

This is the same basic justification as the IO monad in Haskell or the explicit `io` or `World` objects passed as an input and output in Mercury or Clean, respectively. However, in practice, the simpler Mars approach is more limited. For example, Haskell’s IO monad is more flexible than Mars’s explicit `io` keyword. In Haskell, the type `a → IO b` is an ordinary function that returns a value of type `IO b`, an input/output action. It is possible to use the regular `map` function to turn a list of `a` values into a list of `IO b` values, which can be executed at a later time. In Mars, by contrast, the type `a → io b` is a separate “I/O function” type, which is incompatible with the ordinary `a → b` type. It is not possible to use the ordinary `map` function with I/O functions. The decision to use the inflexible `io` keyword was made out of pragmatism, not idealism: to keep the Mars language simple.

Note that computable global constants may not have the `io` annotation; it does not make sense for a constant to perform I/O, because then it might have a different value every time it is called. By contrast, zero-arity functions, such as `main`, are perfectly suited for performing I/O, because the explicit parentheses indicate when the function is to be executed.
4.2.6 Modules

Mars has a very simple module system. Each module is defined in a .mar file. At the top level of a Mars module, an import statement can be used to include the declarations from another module, found on the Mars search path. For example, the following declaration is required to use functions from the standard prelude module:

\[
\text{import prelude}
\]

Modules do not provide namespaces, so importing a module is almost the same as simply pasting its contents into the source file.\(^7\) There is no separate compilation; all of the declarations from all modules in a program are compiled together.

4.2.7 Impure language features

In Section 3.2.2, we spoke of interface integrity with respect to impure language features, such as Haskell’s unsafePerformIO function, that allow a function to violate the language’s pure semantics, and its type safety. Mars has similar features, but (to use the terminology established in that section) these features are unusual, and so we say that Mars exhibits interface integrity for all practical purposes. The features described in this section can be used to circumvent interface integrity, and therefore, responsible software authors should use them only to produce functions that exhibit interface integrity, and generally strive to avoid using them at all.

The first impure feature in Mars is the “=!" destructive field-update operator. This may be used in statement form, as in:

\[
x.field =! y
\]

This statement destructively modifies \(x\), replacing \(field\) with \(y\). The documentation warns against the use of this feature, and the “=!" in the operator name is designed to suggest that it is unsafe.

The second impure feature is the set of impure built-in functions, available by importing the impure module: array_set, array_append, array_extend, array_delete, and is. The first four destructively manipulate arrays in various ways. The is function determines whether two objects have the same identity. While this may seem benign, it is actually a violation of interface integrity, as explained in Section 3.3.3. The name and

\(^7\) However, it is more sophisticated than a simple text include, since duplicate imports of a module are ignored, allowing for two modules to import the same module, or even for cyclic imports.
4.2. OVERVIEW

documentation for the module act as a warning against its use in normal programming.

The third impure feature is the foreign function interface. This feature is essential for serious programs, in order to make use of existing libraries written in other languages, and access operating system services such as window managers and network interfaces. However, it is necessarily impure, as it allows software authors to do anything that is possible in C or another low-level language.

A number of built-in types and functions are provided as part of the foreign-function interface: CChar, CShort, CInt, CFloat, CDouble and CPtr (equivalent of C’s char, short, int, float, double and void* types, respectively). The native module also makes various built-in functions available for converting Mars values to and from these types (e.g., num_cuint to convert from a Mars Num to a C unsigned int), and for manipulating pointers (e.g., malloc, cptr_add, cintptr_ref and cintptr_set). These are provided to allow data to be converted from Mars types to C-compatible types, including arrays, structs and strings. Mars also provides a native_import keyword, which may be applied to a procedure declaration without a body. For example, the C standard library function puts may be declared as follows:

\[
\text{def native_import puts(s :: CPtr) :: io CInt}
\]

The puts function may be called or treated as a closure, just like any other Mars function. However, a valid string pointer (in C, a const char*) must be supplied. A Mars string may be converted into a C string using malloc, cptr_add and ccharptr_set; a convenience function, string_cptr, is provided to do just that. Clearly, this feature may be used to wreak havoc, and the onus is on the programmer to provide pure abstractions over foreign functions (e.g., using io when declaring foreign functions with legitimate side-effects).

Unfortunately, due to its Hindley-Milner type system and value caching for computable global constants, Mars’s impure mutation functions make it possible to violate type safety, even without the use of foreign language code. Example 4.11 shows a Mars implementation of an unsafe (bit-level) type cast. This works because the CGC array_a may be considered an array with elements of any type a. Because CGC types are polymorphic, the a may be a different type each time it is mentioned. In the core language (without mutation), this is perfectly safe, because the only valid value of type Array(a) is []. However, this code uses the impure array_append to destructively modify the array, adding the value x, of type a. The array is subsequently re-interpreted as having elements of type b, which is an invalid assumption. This could be used, for example, to store a Num in the array, then read it back out as an Array(Num), which could allow arbitrary memory access if the number is treated as a pointer.
CHAPTER 4. MARS: AN IMPERATIVE/DECLARATIVE LANGUAGE

Example 4.11 Unsafe type cast in Mars, using impure functions

```python
import impure
def array_a :: Array(a):
    return []
def unsafe_cast(x :: a :: b):
    var x_b :: b
    array_append(array_a, x)
    x_b = array_ref(array_a, 0)
    array_delete(array_a, 0)
    return x_b
```

This problem is not unique to Mars; indeed, the very same exploit is possible in Haskell by using `unsafePerformIO` with `IORef` [58]. This problem can be prevented with static analysis (ML prevents this via a typing rule known as the “value restriction”), but Mars takes the same stance as Haskell: that the impure features are not intended for general use, and therefore the type system should not be extended solely on their account.

It is important to note that any automatic code modification or optimisation is allowed to assume that functions exhibit interface integrity, so that we may develop a simplified semantics for code analysis. The impure features are available on a “use at your own risk” basis. Software authors should use them only to write functions that exhibit interface integrity; those that do not may exhibit undefined behaviour.

4.3 Standard libraries

This section briefly describes relevant functions and types available as part of the Mars standard library. This describes only the most useful functions and types, including all of the ones mentioned elsewhere in this thesis. For the complete set of functions, types and constants, consult The Mars Language Reference [27].

The built-in functions are implicitly available to all Mars programs. They are a small set of functions which could not be implemented in pure Mars, since they deal with built-in types, contain type-polymorphic behaviour, interact with the operating system, or interact with the Mars runtime. Some of the Mars built-ins have a syntactic sugar for convenience. A relevant subset of built-in functions is listed in Table 4.1. Some array operations have argument names and return types omitted, due to space reasons; they are described in detail in Section 4.2.3. There are also built-in operators \(\neq, <=, >\) and \(\geq\) which perform inequality tests using the built-in `eq` and `cmp` operators.

The Mars prelude is a library module which nearly all programs will use. It contains
4.4. EXAMPLE

<table>
<thead>
<tr>
<th>Function signature</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq(x :: a, y :: a) :: Num</td>
<td>$x == y$</td>
<td>1 if $x$ equals $y$; 0 otherwise</td>
</tr>
<tr>
<td>cmp(x :: Num, y :: Num) :: Num</td>
<td>-1 if $x &lt; y$; 0 if $x = y$; 1 if $x &gt; y$</td>
<td></td>
</tr>
<tr>
<td>add(x :: Num, y :: Num) :: Num</td>
<td>$x + y$</td>
<td>add $y$ to $x$</td>
</tr>
<tr>
<td>sub(x :: Num, y :: Num) :: Num</td>
<td>$x - y$</td>
<td>subtract $y$ from $x$</td>
</tr>
<tr>
<td>mul(x :: Num, y :: Num) :: Num</td>
<td>$x \times y$</td>
<td>multiply $x$ and $y$</td>
</tr>
<tr>
<td>fdiv(x :: Num, y :: Num) :: Num</td>
<td>$x \div y$</td>
<td>divide $x$ by $y$ (floating point)</td>
</tr>
<tr>
<td>div(x :: Num, y :: Num) :: Num</td>
<td>$x \div y$</td>
<td>divide and round towards $-\infty$</td>
</tr>
<tr>
<td>mod(x :: Num, y :: Num) :: Num</td>
<td>$x \mod y$</td>
<td>$x$ modulo $y$</td>
</tr>
<tr>
<td>sqrt(x :: Num) :: Num</td>
<td></td>
<td>square root of $x$</td>
</tr>
<tr>
<td>array(len :: Num, def :: a) :: Array(a)</td>
<td></td>
<td>array with $len$ copies of $def$</td>
</tr>
<tr>
<td>array_ref(array :: Array(a), i :: Num) :: a</td>
<td></td>
<td>element $i$ of array</td>
</tr>
<tr>
<td>array_length(array :: Array(a)) :: Num</td>
<td></td>
<td>length of array</td>
</tr>
<tr>
<td>array_replace(Array(a), Num, a)</td>
<td></td>
<td>array with element replaced</td>
</tr>
<tr>
<td>array_add(Array(a), a)</td>
<td></td>
<td>array with element appended</td>
</tr>
<tr>
<td>array_concat(Array(a), Array(a))</td>
<td></td>
<td>concatenation of two arrays</td>
</tr>
<tr>
<td>array_remove(Array(a), Num)</td>
<td></td>
<td>array with element removed</td>
</tr>
<tr>
<td>put_char(char :: Num) :: io Num</td>
<td></td>
<td>write $char$ to standard output</td>
</tr>
<tr>
<td>get_char() :: io Num</td>
<td></td>
<td>read $char$ from standard input</td>
</tr>
<tr>
<td>show(x :: a) :: Array(Num)</td>
<td></td>
<td>string representation of $x$</td>
</tr>
<tr>
<td>error(msg :: Array(Num)) :: a</td>
<td></td>
<td>halt program</td>
</tr>
</tbody>
</table>

Table 4.1: Subset of the Mars pure built-in functions

<table>
<thead>
<tr>
<th>Type signature</th>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pair(a, b)</td>
<td>Pair(a, b)</td>
</tr>
<tr>
<td>List(a)</td>
<td>Cons(head :: a, tail :: List(b)); Nil</td>
</tr>
<tr>
<td>Maybe(a)</td>
<td>Just(value :: a); Nothing</td>
</tr>
</tbody>
</table>

Table 4.2: Subset of the Mars prelude types

A large set of useful utility types and functions. A relevant subset of prelude types is listed in Table 4.2; a subset of prelude functions is listed in Table 4.3. Not shown are the higher-order array-based versions of the list functions, such as array_filter.

4.4 Example

The purpose of Mars is to be programmed imperatively, while providing strong guarantees to the programmer. While there is currently no community of Mars programmers, we are still able to identify “idioms” in our own code that we expect to represent the typical Mars style. In this section, we give an example of what we consider to be idiomatic Mars style.

In Section 2.4, we discussed imperative programming in Haskell. Example 2.9 gave our Haskell implementation of a hash table using the STArray as an underlying data
### CHAPTER 4. MARS: AN IMPERATIVE/DECLARATIVE LANGUAGE

<table>
<thead>
<tr>
<th>Function signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id(x :: a) :: a</td>
<td>returns x</td>
</tr>
<tr>
<td>const(k :: a) :: b → a</td>
<td>returns a function that returns k</td>
</tr>
<tr>
<td>apply(f :: a → r, x :: a) :: r</td>
<td>applies f to x</td>
</tr>
<tr>
<td>compose(f :: b → c, g :: a → b) :: a → c</td>
<td>the composition of f and g</td>
</tr>
<tr>
<td>not(x :: Num) :: Num</td>
<td>1 if x = 0; 0 otherwise</td>
</tr>
<tr>
<td>ne(x :: a, y :: a) :: Num</td>
<td>1 if x ≠ y; 0 otherwise</td>
</tr>
<tr>
<td>abs(x :: Num) :: Num</td>
<td>absolute value of x</td>
</tr>
<tr>
<td>min(x :: Num, y :: Num) :: Num</td>
<td>lesser of x and y</td>
</tr>
<tr>
<td>max(x :: Num, y :: Num) :: Num</td>
<td>greater of x and y</td>
</tr>
<tr>
<td>length(list :: List(a)) :: Num</td>
<td>length of list</td>
</tr>
<tr>
<td>append(a :: List(a), b :: List(a)) :: List(a)</td>
<td>a concatenated with b</td>
</tr>
<tr>
<td>map(f :: a → b, l :: List(a)) :: List(b)</td>
<td>f applied to each element of l</td>
</tr>
<tr>
<td>filter(f :: a → Num, l :: List(a)) :: List(a)</td>
<td>elements of l satisfying f</td>
</tr>
<tr>
<td>foldl(f :: (a, b) → a, z :: a, l :: List(b)) :: a</td>
<td>left-fold of f on z and l</td>
</tr>
<tr>
<td>foldr(f :: (a, b) → b, z :: b, l :: List(a)) :: b</td>
<td>right-fold of f on l and z</td>
</tr>
<tr>
<td>print_string(string :: Array(Num)) :: io Num</td>
<td>write string to standard output</td>
</tr>
<tr>
<td>get_line() :: io Array(Num)</td>
<td>read a line from standard input</td>
</tr>
</tbody>
</table>

Table 4.3: Subset of the Mars prelude functions

structure. Example 4.12 gives our Mars implementation of the same code. The null function determines whether a Maybe value is Nothing. The function hash :: a → Num computes a number from any given Mars value.

This demonstrates several Mars idioms. Firstly, the fact that we are using arrays is idiomatic. Whereas users of other declarative languages would generally prefer to use a tree-based representation which allows \(O(\log n)\) update time, Mars is designed to allow the use of array-based structures which have \(O(n)\) update time in the naïve implementation, but which can be updated in \(O(1)\) time if programmed carefully, and our destructive update optimisation is enabled.

Secondly, note the update to the table array in intersection:

\[
\text{table} = \text{insert}(\text{table}, \text{array_ref}(\text{xs}, i))
\]

The array is treated as an ordinary value, with the updated value being re-assigned to the variable on each loop iteration, just like \(i = i + 1\) on the next line. This explicit assignment (as opposed to implicit mutation of the array) would not be present in a traditional imperative language implementation of this function, nor is it present in the Haskell or Disciple version in this thesis (Examples 2.9 and 2.10, respectively). In the

---

8As noted in Section 2.4, this implementation is severely flawed: it allows duplicates, does not resize when near capacity, and resolves collisions with linear probing. It is simply an example of Mars code, and not a serious hash table implementation.
4.5. CONCLUSION

Example 4.12 Hash table implementation and usage in Mars

```python
def empty(capacity :: Num) :: Array(Maybe(a)):
    return array(capacity, Nothing)

def insert(table :: Array(Maybe(a)), value :: a) :: Array(Maybe(a)):
    h = hash(value) % array_length(table)
    while not(null(array_ref(table, h))):
        h = (h + 1) % array_length(table)
    return array_replace(table, h, Just(value))

def member(table :: Array(Maybe(a)), value :: a) :: Num:
    ...

def intersection(xs :: Array(a), ys :: Array(a)) :: Array(a):
    table = empty(array_length(xs) × 2)
    i = 0
    while i < array_length(xs):
        table = insert(table, array_ref(xs, i))
        i = i + 1
    return array_filter(member(table, ...), ys)
```

Mars version, it is the variable that is being updated through assignment, not the object being updated through side-effects; any aliases of the array will not be unexpectedly updated.

4.5 Conclusion

In this chapter, we gave an informal description of the experimental programming language Mars, which was developed alongside this thesis for the purpose of exploring a hybrid imperative/declarative programming language, and the automatic generation of destructive update operations. We have described, with examples, most of the syntax and semantics of the programming language, with the formal syntax and detailed semantics being described in the Mars Language Reference [27]. In subsequent chapters, the destructive update optimisation is described in detail.

Mars is an imperative programming language, with syntax and some of its semantics modelled after Python. Unlike Python, Mars obeys the rules of interface integrity as described in Chapter 3: it features statement sequences, selection and iteration operators, and local variable assignment, but individual procedures do not have any side-effects. The only effect of calling a procedure is receiving a value as the result. As an exception,
functions marked with the \texttt{io} keyword are allowed to interact with the operating system as a side-effect, which is justified in that the \texttt{io} keyword itself represents the “state of the world” as both an input and output, much like the IO monad in Haskell.

Mars is by no means a fully featured programming language. Among the features that it is missing which we consider necessary for mainstream usage are a \texttt{for} loop mechanism, structured exception handling, type classes, proper namespaces for modules, and additional syntactic sugar such as array indexing operators.

Aside from its lack of type classes, Mars features an advanced type system comparable to that of ML and Haskell. Mars types are polymorphic, allowing type variables which may be substituted for any type. It also allows the definition of algebraic data types, which may be matched and unpacked using the \texttt{switch} statement, or have fields read or updated by name.

Functions are first-class values in Mars, able to be stored in local variables or elements of a data structure, and passed as function arguments. Mars also supports partial application via the “…” operator. The standard higher-order functions, such as \texttt{map}, \texttt{filter} and \texttt{foldl} are implemented in the Mars standard prelude. Mars also features a special type of global constant which contains a sequence of statements that are evaluated the first time the constant’s value is requested, and cached for subsequent use.

The built-in array type allows for constant-time access and, with the appropriate optimisation, update, to individual array elements. Unlike other declarative languages, Mars users are encouraged to use arrays for list processing, allowing a more natural imperative style of programming. For efficiency, this will rely on an automatic destructive update optimisation to reduce array update operations from $O(n)$ to $O(1)$ time complexity. This optimisation is the subject of the remainder of this thesis.

In the next chapter, we delve deeper into the semantics of a subset of the Mars programming language, describing the compiler’s intermediate representation known as MIR. Our compiler strips away all of the syntactic sugar and several of the higher-level language features, compiling Mars code down into the simplified MIR language. This representation is the subject of our aliasing and destructive update analysis, and also serves as the input to the code generator.
Chapter 5

Code representation for analysis

5.1 Introduction

The major part of this thesis deals with static analysis of Mars programs to determine whether particular variables are aliased at certain program points, so that update operations can be made destructive if it can be shown that the variable being updated is not aliased. We do not perform static analysis on the Mars programming language itself. First, the code is transformed into a flattened representation, the “Mars Intermediate Representation” or MIR, which is easier to analyse. This chapter describes the MIR language,\(^1\) which will be used throughout the remainder of the thesis.

There are a number of reasons to perform the analysis on MIR, rather than Mars. Most importantly, because the order of evaluation is explicit in MIR, it has a clear operational semantics, whereas in Mars code, it is unclear in which order the expressions should be evaluated (in non-io functions, this does not affect the result,\(^2\) but it does affect the memory usage, which is important for the analysis). Secondly, it makes the analysis language-independent — any pure language which can be described in terms of MIR could have the analysis applied. Lastly, it allows us to ignore any high-level constructs which can be desugared down to the simpler intermediate representation.

MIR is based on static single assignment (SSA) [17]. This was chosen over functional-style code representations (such as A-normal form) because it is an ideal representation for analysis of imperative code, and also because it closely matches the LLVM bitcode [57] that MIR is transformed into. We opted not to perform the analysis directly on the LLVM

---

\(^1\)The real Mars compiler uses a somewhat different intermediate representation language, with a few implementation-level differences and a different notation; see Section 5.4. For the purpose of this thesis, the language described here is sufficiently similar to the one used in the compiler.

\(^2\)With the caveat that it is unspecified which error/non-termination result will be visible to the user.
code, because it lacks too much high-level information. While LLVM is type-aware, it does not support type polymorphism, tagged unions, closures, or partial application, so the Mars compiler discards this information and generates C-like memory allocation operations and pointer casts. The analysis would be much more complicated if the full generality of these features had to be taken into account, so we limit the scope of the analysis to the much higher-level MIR.

5.2 Converting Mars to MIR

We do not formally describe the process for converting Mars code to MIR. This is a rather long process, but its implementation should be fairly self-explanatory. This section describes the differences between Mars and MIR, and informally, how one might transform Mars code down to the MIR level.

5.2.1 Flattening expressions

MIR can be described as a “semi-flattened” form of code with a weak form of static single assignment (SSA) [17]. Normal Mars program code allows nested statements (for example, a block of statements may appear nested within an if statement), as well as nested expressions (for example, a function call expression can appear as an argument to another function call expression). MIR flattens expressions, but not statements. Hence, function calls and other complex expressions are not allowed inside other expressions, but statements may still appear nested within one another. For example, the Mars function quadratic_formula in Example 5.1a may be compiled to the MIR code shown in Example 5.1b, where all of the nested expressions are given an explicit order and converted into separate instructions, assigning their results to fresh temporary variables, but nested constructs such as if statements remain.

5.2.2 The reassignment rule

To simplify analysis, we introduce a restriction into MIR. The “reassignment rule,” given in Rule 5.1, prevents a variable from being assigned twice on the same code path.

Rule 5.1. For any statement sequence $s_1; s_2$, there shall be no variable assigned in both $s_1$ and $s_2$. No variable that is a parameter of a function shall be assigned in the body of that function.

3In MIR, the terms statement and instruction are used interchangeably.
Example 5.1 Quadratic formula function in source form (a) and MIR form (b)

```
def quadratic_formula(a, b, c):
    delta = b * b - 4 * a * c
    if delta ≥ 0:
        r1 = (-b + sqrt(delta)) / (2 × a)
        r2 = (-b - sqrt(delta)) / (2 × a)
        r = Pair(r1, r2)
    else:
        r = error("No real solutions")
    return r
```

```
def quadratic_formula(a, b, c):
    t0 = b * b
    t1 = 4 × a
    t2 = t1 × c
    delta = t0 - t2
    t3 = delta ≥ 0
    if t3:
        t4 = -b
        t5 = sqrt(delta)
        t6 = t4 + t5
        t7 = 2 × a
        r1 = t6 / t7
        t8 = t4 - t5
        r2 = t8 / t7
        r = Pair(r1, r2)
    else:
        t4 = "No real solutions"
        r = error(t4)
    return r
```

This rule allows us to assume that any variable assigned in a given statement is not already in scope. In our aliasing analysis covered in Chapter 8, this is crucial in ensuring that aliases of one variable do not “bleed” over into a new variable with the same name. This scheme was chosen over full SSA in order to keep code readable (with nested statements instead of goto).

This means that code of the form “x = f(x)” must be rewritten as “x1 = f(x),” with all subsequent references to x updated to refer to x1. Each time x is assigned, it must be given a new subscript. It also means that we need a special treatment for while loops, discussed in Section 5.2.5. Note that this does not only apply to two adjacent statements, but the entire function body (because statements are defined recursively; for example, a sequence of three statements is written as s1; (s2; s3)). However, unlike SSA, it does allow the same variable to be assigned on two alternative code paths (e.g., r in Example 5.1).

5.2.3 The unique arguments rule

There is another restriction introduced into MIR to simplify analysis. The “unique arguments rule,” given in Rule 5.2, prevents a variable from being passed as two or more parameters to a function or similar.

**Rule 5.2.** For any array literal [x₁, ..., xₙ], any constructor, closure or global function call \( f(x₁, ..., xₙ) \), or closure template reification, \( c\{x₁, ..., xₙ\} \), there must not be a variable \( v \) that
appears as two or more of \( x_1, \ldots, x_n \).

This means that code of the form “\( y = f(x, x) \)” must be rewritten as “\( t = x ; y = f(x, t) \)”. Code analysis may assume that all arguments to a function are either numbers or unique variable names. Closure template reification is introduced in Section 5.2.8.

### 5.2.4 Single exit point

MIR does not allow return statements at arbitrary places — there must be a single return statement at the end of every function, and none anywhere else. Thus, the compiler must rearrange any code with early exits such that it flows through to the end, without changing the semantics.\(^4\)

### 5.2.5 While loops

There is a while construct in MIR, but it has special properties to cope with the reassignment rule (Rule 5.1). Almost all while loop bodies reassign variables that were in scope before the body, which is a violation of the reassignment rule (see Example 5.2a). It is not possible to simply rename a variable updated in the body without breaking the feedback loop from one iteration to the next.

**Example 5.2** Imperative factorial function in Mars (a) and MIR (b)

```plaintext
def factorial(n):
    x = 1
    while n:
        x = x × n
        n = n - 1
    return x

def factorial(n):
    x = 1
    while n [x_1 = \phi(x, x'_1), n_1 = \phi(n, n'_1)]:
        x'_1 = x_1 × n_1
        n'_1 = n_1 - 1
    return x_1
```

To resolve this problem, the MIR while header includes a special “phi” syntax, reminiscent of the phi notation in SSA, for declaring variables that are updated in the loop body, as shown in Example 5.2b. The notation “\([x_1 = \phi(x, x'_1), n_1 = \phi(n, n'_1)]\)” has two implicit effects:

1. Before entering the loop, \( x_1 \) and \( n_1 \) take the values of \( x \) and \( n \), respectively.
2. At the end of each loop iteration, \( x_1 \) and \( n_1 \) take the values of \( x'_1 \) and \( n'_1 \), respectively.

\(^4\)This is a difficult transformation. The real Mars compiler avoids having to do this because in the compiler, MIR has branch instructions, which are omitted from this thesis for simplicity.
This allows a loop to iteratively update variables without violating the reassignment rule. Effectively, the `while` loop, taken as a single statement, reads \( x \) and \( n \) and assigns the new variables \( x_1 \) and \( n_1 \) (the intermediate variables \( x'_1 \) and \( n'_1 \) can be ignored).

We could have avoided having `while` loops in MIR entirely by converting loops into equivalent recursive functions. However, the decision was made early in the development of the compiler to keep loops in MIR, which makes analyses and optimisations, both in our compiler and LLVM, more straightforward (although our aliasing analysis itself preserves information across function calls).

### 5.2.6 Algebraic data types, switches and fields

MIR has a simplified syntax for dealing with algebraic data types (tagged unions). It does not have the following high-level Mars features:

- Variable binding in the cases of a `switch` statement. In MIR switch statements are purely control-flow constructs, and do not perform variable binding or recursive pattern matching.
- The numeric version of the `switch` statement. Such statements must be converted to nested `if-else` blocks.
- Default case on the `switch` statement. Default cases must be converted to a separate case for each constructor not explicitly covered by another case.
- “Smart” field access expressions. Mars allows a) runtime error detection if the constructor used to build the given object has no field of the given name, and b) when multiple constructors each have a field with the same name and type, accessing that field by name, and automatically selecting the appropriate constructor. By contrast, MIR code may only access the field of a data structure if it is statically known which constructor was used to build the value.

In MIR, fields are accessed by index, not by name. This permits access to fields without a name. For example, in Mars, the first and second elements of a Pair named \( x \) could be accessed via \( x.fst \) and \( x snd \), respectively, whereas in MIR, one would use \( x.Pair/0 \) and \( x.Pair/1 \), respectively. Fields are numbered, starting from 0 for the first field of each constructor. The field numbers are parameterised by constructor name. For example, the Either type has two constructors: Left and Right, with one field each. In MIR, these are accessed via \( x.Left/0 \) and \( x.Right/0 \), respectively. It is not legal to access a field of an object unless the constructor mentioned in the code is statically known to be the constructor.
used to build that object. In order to access the field of an Either, one must first `switch` on the constructor and then access the field, as shown in Example 5.3. A low-level code generator may subsequently optimise the switch away, if indeed all of the fields are at the same byte offset.

Example 5.3 Appropriate use of field access on an Either value in MIR

```plaintext
switch my_either:
    case Left:
        v = my_either.Left/0
    case Right:
        v = my_either.Right/0
```

Consider Example 5.4a, a `sum` function in Mars source form. The line "`case Cons(h, t)`" affects both control flow (it is branched to if and only if `nums` is a Cons value) and variable bindings (it binds the variables `h` and `t` from the fields of `nums`). MIR does not support variable bindings in the `case` statement, nor does it support nested pattern matching. This code is transformed into the MIR code shown in Example 5.4b. Note that the `case` statement only mentions the constructor name and not the fields; `h` and `t` are assigned by explicit field access statements.

Example 5.4 sum function in Mars (a) and MIR (b)

```plaintext
def sum(nums):
    switch nums:
        case Nil:
            r = 0
        case Cons(h, t):
            t' = sum(t)
            r = h + t'
    return r
```

An example of “smart” field access is the List1 type, which represents a non-empty reverse list (with the head at the end). It has two constructors, `Cons(tail, head)` and `Last(head)`. Importantly, the field name `head` appears in both constructors, and the Mars field-reference expression allows the programmer to easily access the head of a list, regardless of which constructor is present, as shown in Example 5.5a. In MIR, the fields are numbered, so the Cons fields `tail` and `head` are fields 0 and 1, respectively, while the Last

---

5 Note that while these bindings do escape the `case` scope, this is safe because a variable is not allowed to be used unless it is definitely assigned on all code paths.

6 The method for doing so is non-trivial, due to the possibility of partially overlapping patterns. It is described in the Mars documentation under the heading “switch factoring algorithm.” [27]
field \textit{head} is field 0. The equivalent MIR code, shown in Example 5.5b, is a \texttt{switch} statement. This example shows the need for the \texttt{switch}: the field named \textit{head} has a different index on each constructor.

\begin{example} \textbf{5.5} Getting the head of a \texttt{List1} in Mars (a) and MIR (b)
\begin{verbatim}
\texttt{h = list.head}
\end{verbatim}
\begin{verbatim}
\texttt{switch list:
  case Cons:
    \texttt{h = list.Cons/1}
  case Last:
    \texttt{h = list.Last/0}
}\end{verbatim}
\end{example}

\subsection{5.2.7 Field-replace operations and hinted constructor calls}

The Mars field-replace expression (of the form \texttt{x.f := v}) produces a new object which is the same as the algebraic data type object \texttt{x}, but with field \texttt{f} replaced by value \texttt{v}. As with field-reference expressions, the field name \texttt{f} may refer to a field present in several different constructors, so the expression must be transformed into an explicit \texttt{switch} if the constructor is not statically known, as shown in Example 5.5. In addition, MIR does not feature a field-replace operator, so these expressions must be transformed into code that explicitly extracts the remaining elements from the object and then calls the constructor to generate a new object. Example 5.6 shows this transformation applied to a field-replace on the \texttt{List1} type.

\begin{example} \textbf{5.6} Replacing the head of a \texttt{List1} in Mars (a) and MIR (b)
\begin{verbatim}
\texttt{newlist = list.head := x}
\end{verbatim}
\begin{verbatim}
\texttt{switch list:
  case Cons:
    \texttt{t = list.Cons/0}
    \texttt{newlist = Cons(t, x)}
  case Last:
    \texttt{newlist = Last(x)}
}\end{verbatim}
\end{example}

The resulting code is equivalent, yet it has lost a small amount of information: the constructor calls in the resulting code do not specify which variable held the original value. This information is unnecessary for executing the program, but in terms of a reuse analysis, it provides us with a potentially useful \textit{hint} as to which memory cell to attempt to reuse.

A major component of this research is that a call to a constructor, naïvely implemented as a memory allocation, may sometimes be optimised by reusing an existing dead memory cell. This optimisation is described in detail in Chapter 10. While a sophisticated heuristic could decide which dead cell to reuse, it is desirable to follow the advice of the
programmer when the source code explicitly requests that a particular variable be updated. Therefore, we do not throw away this information. It is present in MIR in the form of a hinted constructor call, a constructor call of the form:

\[
\text{constructor}(a_1, \ldots, a_n) \{ \text{var} \}
\]

This indicates that the compiler should attempt to reuse the memory allocated for \text{var} for the newly constructed value. Semantically, the hint has no meaning; it is solely provided for the reuse analyser. The reuse analyser may only reuse the hinted variable if it is safe to do so, and is still free to reuse other variables. The code of Example 5.6b with hinted constructor calls is shown in Example 5.7.

**Example 5.7** MIR for Example 5.6a with hinted constructor calls

```ml
switch list:
  case Cons:
    t = list.Cons/0
    newlist = Cons(t, x) \{ list \}
  case Last:
    newlist = Last(x) \{ list \}
```

### 5.2.8 Closure templates

Functions are first-class values in Mars, with the mention of a function name being a valid expression in its own right. For example, consider the notation “f(x_1, \ldots, x_n),” where f is a global function name. While it is quite common to apply a global function, there is nothing intrinsically special about the fact that a global function name is being used in an application — the function f could just as well have been assigned to a variable (e.g., “t = f”), or some other expression could have been applied (e.g., “\text{const}(4)(x)”)). However, in MIR, global functions may not be used as values; they are second-class citizens that must be explicitly converted into closures (Definition 5.1), which are first-class function values.

Furthermore, Mars allows explicit partial application of functions, using the notation “f(x_1, \ldots, x_n, \ldots).” This is not allowed in MIR, again requiring the explicit creation of a closure. This section describes that process.

**Definition 5.1.** In MIR, a closure is a function value that may be stored in local variables and used in contexts where values are expected. Closures may or may not contain embedded values.

---

7 The Mars compiler does not include hinted constructor calls, as destructive update of user-defined types is not yet implemented.

8 This notation may cause some confusion: the first ellipsis (\ldots) denotes the \(n - 2\) arguments in between \(x_1\) and \(x_n\), while the second ellipsis is literal punctuation in the Mars language, invoking partial application.
Definition 5.2. **Reification** is the process of converting a global function into a closure.

In general, when generating MIR, references to global functions must be converted into explicit reification operations (Definition 5.2) in order to produce values. However, the common case where a global function is directly applied (e.g., \( f(x_1, \ldots, x_n) \)) can be kept as-is, without the need to first reify the function, and then apply the resulting closure, but this is an optimisation and not a necessity. For all other uses of global functions (such as storing them in variables, passing them as arguments or as array elements, or partially applying them), the compiler must explicitly construct closure objects using *closure templates*.

A closure template is a top-level execution unit, much like a function, except that as well as its ordinary parameter list, it refers to one or more *free variables* which must be bound into a closure before it may be called. A closure template may be converted into a closure at runtime, by supplying a value for each of the template’s free variables, much like invoking a constructor. A closure is therefore a pair consisting of a closure template (code) and a vector of values for the free variables (data). The only way to construct a closure in MIR is by reifying a closure template.

Example 5.8a shows a partial application expression in Mars, \( \text{add}(3, \ldots) \).” To produce equivalent MIR code, the compiler must generate the \( \text{add}_1 \) closure template, shown in Example 5.8b (if it has not already done so). This template has one free variable, \( x \), and one ordinary function parameter, \( y \). It may be instantiated in MIR by supplying a value for \( x \). The statement \( f = \text{add}_1\{3\} \) reifies the closure template, creating a first-class closure value which takes one argument and adds 3 to it. This value can then be applied; for example, the expression \( f(5) \) has the value 8.

**Example 5.8** Partial function application in Mars, and equivalent closure template in MIR

\[
\begin{align*}
  f &= \text{add}(3, \ldots) \\
  \text{def } \text{add}_1\{x\}(y): \\
  &\quad \text{return } x + y \\
  f &= \text{add}_1\{3\}
\end{align*}
\]

The key difference between the partial application expression in Mars and the reification expression in MIR is that the reification requires a special piece of code to exist statically in the program that is purpose-built for partially applying precisely that number of arguments. This allows for an efficient implementation on many compiler back-ends. For example, when the Mars compiler generates LLVM code, it compiles each closure template into a function that takes \( \text{freevars} \), an array of free variables, as the first parameter. Reification statements compile into code that constructs a pair \( (\text{freevars}, fp) \), where \( fp \) is a function pointer to the closure template. Call statements compile into code that calls \( fp \), passing \( \text{freevars} \) as the first argument. It would not be possible to directly compile
the Mars partial application expression into LLVM, because some intermediate code — a closure template — is required that knows how many free variables will be stored in the closure.

Partial application of a closure is more complicated, but can also be implemented using closure templates. Consider Example 5.9a, converted to MIR in Example 5.9b. Since the function \( f \) is unknown, the compiler constructs a generic closure template, \( ct_{2:1} \), for partially applying functions of two arguments with one argument. Unlike the closure templates above, this one includes the partially applied closure as a free variable, \( g \). The reification stores \( f \) inside the new closure, so that the correct function will be called when the closure is applied.

**Example 5.9** Partially applying a closure in Mars (a) and MIR (b)

```python
def partial(f :: (a, b) -> c, x :: a) :: b -> c: return f(x, . . .)
def ct_{2:1}[g, u](v): return g(u, v)
def partial(f, x): return ct_{2:1}{f, x}
```

Closure templates are also required for simple reification — when a global function is mentioned by name, not in the context of a direct or partial application. It is not possible to simply take the address of a global function, due to an implementation requirement that all reified function objects be constructed via closure templates (to see why, consider the LLVM back-end described above — it expects the \( fp \) of all function objects to point to a function that takes a \( freevars \) array as its first argument, which is not true for global functions). This case is equivalent to a partial application of zero arguments ("\( f \)" is equivalent to "\( f(\ldots) \)"). A special closure template with zero free variables is constructed for the function being reified, as shown in Example 5.10.

**Example 5.10** Treating a global function as a value in Mars (a) and MIR (b)

```python
def succ_list(list):
    return map(succ, list)
def succ0[](x):
    return succ(x)
def succ_list(list):
    f = succ0{}
    return map(f, list)
```

Implementing closures by defining code with a sequence of required free variables is a common technique, for example, employed by Python [9] (via the `MAKE_CLOSURE` instruction). There does not appear to be any agreement on the name of the execution unit used to create the closure (Python calls it a “code object”). The name “closure template”

---

9In practice, it is more complex than this, since two unknown functions of different types will require two different closure templates.
5.3. CONVERTING OTHER LANGUAGES TO MIR

has been used by Thiemann [89] to describe the equivalent construct and an associated function closure-template in Scheme 48.\(^\text{10}\) We prefer this term as it is fairly descriptive of the concept.

5.2.9 Type dictionaries

Two built-in Mars functions, show (turn a value into a string representation) and eq (test whether two values are equal, a synonym for the “==” operator), are defined to work over all types. This poses an implementation problem: how can show (for example) know the type of its argument if values do not possess any runtime type information? The solution to this problem in Haskell and other functional languages is type classes. For example, the show function might have the type class constraint “Show,” which means it will only work for types that have an explicit operation for converting values of that type into strings. In the implementation, calling a function with a type class constraint causes the compiler to pass a “type dictionary” or “method dictionary” containing a pointer to the show implementation for the argument’s type [95].

In the Mars implementation, we implement a cut-down version of this scheme. Mars effectively provides a single type class “ShowEq” with both a show and eq method, and implicitly provides an implementation of the two methods for all types, built-in and user-defined. A simple static analysis annotates a type class constraint “ShowEq” for each function that requires either of these methods.

MIR does not support type dictionaries at all: the compiler makes all of this explicit in the code it emits. Each type’s “type dictionary” is simply a pair consisting of a show and eq function value. These dictionaries are passed as arguments to any function that requires them, and the relevant function value is extracted and called when required. An example of this transformation is shown in Example 5.11. This approach could easily be extended to handle a language with full type classes.

5.3 Converting other languages to MIR

It should also be possible to convert other high-level languages to MIR for analysis. This section describes how one might translate features of non-Mars languages (in particular Haskell) to MIR. While we do not intend to compile Haskell or other languages to MIR, this section may give some indication as to how the algorithms presented in the later chapters of this thesis may be applied to other languages, or future versions of Mars.

\(^\text{10}\)Oddly, Scheme 48 does not appear to have the relevant functions described by Thiemann, nor do any of its historical manuals mention any such concepts.
Example 5.11 Use of \texttt{eq} in Mars (a), implemented as type dictionaries in MIR (b)

\begin{verbatim}
def \texttt{ne}(x, y):
  t0 = \texttt{eq}(x, y)
  return not(t0)
def \texttt{not\_100}(n):
  return \texttt{ne}(n, 100)

\end{verbatim}

\begin{verbatim}
def \texttt{show}\texttt{\textunderscore}\texttt{Num}_0[](x):
  return \texttt{show}\texttt{\textunderscore}\texttt{Num}(x)  \triangleright Primitive function
def \texttt{eq}\texttt{\textunderscore}\texttt{Num}_0[](x, y):
  return \texttt{eq}\texttt{\textunderscore}\texttt{Num}(x, y)  \triangleright Primitive function
def \texttt{dict}\texttt{\textunderscore}\texttt{Num}:
  t0 = \texttt{show}\texttt{\textunderscore}\texttt{Num}_0[]{}
  t1 = \texttt{eq}\texttt{\textunderscore}\texttt{Num}_0[]{}
  return \texttt{Dict}(t0, t1)
def \texttt{ne}(d, x, y):
  eq = d.1  \triangleright Get eq from dictionary
  t0 = \texttt{eq}(x, y)
  return not(t0)
def \texttt{not\_100}(n):
  d = \texttt{dict}\texttt{\textunderscore}\texttt{Num}
  return \texttt{ne}(d, n, 100)
\end{verbatim}

5.3.1 Lambda expressions and nested functions

Mars does not feature lambda expressions or the ability to nest functions, but many other high-level languages do. Both of these language features typically allow the nested function to close over the local variables in the containing function, creating a closure. These can be translated to MIR using closure templates, in much the same way as partial application expressions.

Note that, because MIR does not support mutation, it only allows the implementation of \textit{value closures}, in which the enclosed variables’ values are copied into the closure, and subsequent re-assignment to those variables does not affect the closure. The distinction between value closures and the more common reference closures is discussed in Section 3.3.7.

Each nested function is converted to a closure template with a unique name and a free variable for each unbound variable mentioned inside the nested function. The lambda expression itself is replaced with a reification of the closure template, passing the current value of each required free variable. An example of this transformation applied to the code from Example 3.8 (written in a hypothetical version of Mars with lambda expressions) is shown in Example 5.12.
5.3. CONVERTING OTHER LANGUAGES TO MIR

Example 5.12 Hypothetical Mars extended with lambda expressions (a) and MIR (b)

```python
def closure_maker():
    x = 7
    f = \(y \rightarrow x + y\)
    x = 12
    return f

def closure_maker1(x)(y):
    return x + y

def closure_make():
    x = 7
    f = closure_make1(x)
    x = 12
    return f
```

5.3.2 Curried functions

Mars functions are uncurried by default — they take \(n\) arguments (where \(n \geq 0\)), which must all be supplied at the same time. The language allows partial application (supplying fewer than the expected number of arguments), but this must be explicit in the call syntax (via the “…” notation). Some other higher-order languages, notably Haskell and ML, are curried languages [62]. In such languages, all functions take precisely one argument. Anything that seems to take more than one argument actually (semantically, at least) takes only one argument, returning a function that takes the next argument, and so on. In other words, the following Haskell functions are equivalent:

\[
\text{add\_three} \quad x \ y \ z = x + y + z \\
\text{add\_three} = \lambda x. \lambda y. \lambda z. x + y + z
\]

Functions in MIR, like Mars, are uncurried. Nevertheless, a curried language may be translated to MIR by making the lambda expressions shown above explicit, and translating them to closure templates as in Section 5.3.1.

Example 5.13 Curried-style function in Haskell (a) and MIR (b)

```python
add\_three \ x \ y \ z = x + y + z
add\_three = \lambda x. \lambda y. \lambda z. x + y + z
```

```python
def add\_three\_2(x, y)(z):
    t0 = x + y
    return t0 + z

def add\_three\_1(x)(y):
    return add\_three\_2(x, y)

def add\_three\_0(){(x):
    return add\_three\_1{x}

def add\_three:
    return add\_three\_0{}
```

Example 5.13 shows the add\_three example translated from Haskell to MIR using this approach. The resulting code is quite inefficient, as calling the function requires

\[^{11}\text{Notwithstanding a handful of subtle differences in the type system, such as the monomorphism restriction.}\]
the construction of two intermediate closures. A more efficient approach would be to implement the function in uncurried style, and only produce the closures if the function is applied with fewer than three arguments — this is how the GHC (the Haskell compiler) works [62], but this approach would require changes to MIR’s back-end architecture. Our simple approach is sufficient for static analysis of curried functions.

A difficulty in dealing with curried languages is the added complexity of static analysis. Any analysis which does not take higher-order values into account will break (or over-approximate) on any function of more than one parameter. Thus, handling this curried style is one of the reasons why it is important that our reuse analysis be able to handle higher-order functions, as we shall explore in Chapter 9.

5.3.3 Type classes

Type classes were briefly discussed in Section 5.2.9. A language with type classes could be transformed into MIR using a very similar technique to the type dictionaries described in that section, for implementing the show and eq built-in functions in Mars. This would require a type dictionary value be constructed for each type class instance, and passed as an argument to each function type class constraint [95].

The fact that type classes may be transformed into ordinary MIR means that the higher-order aliasing analysis works for type class code without any modification. A further discussion on this appears in Section 9.9.1.

5.3.4 Lazy evaluation

Mars has strict semantics, which means that every variable holds a value. A non-strict language like Haskell may use a lazy evaluation strategy, whereby values remain uncomputed until they are required. An uncomputed value (or thunk) can be implemented in MIR using closure templates, but MIR lacks the machinery to memoize thunk values (cache their results so that they do not have to be repeatedly evaluated).

This could be added to MIR in the form of “thunk templates.” Thunk templates are essentially closure templates with zero arguments; reifying a thunk template produces a thunk value. The only other difference between a thunk and a closure is that when a thunk is applied, its internal function pointer is overwritten with the value of its result. This memoization behaviour does not affect the semantics of the program, so for the purposes of analysis, we can simply treat the thunk template as a closure template.
5.4 The implementation of MIR

There are a number of differences between MIR as described here (called “Thesis MIR”) and as implemented in the actual Mars compiler (called “Implementation MIR”). This section accounts for the major differences. The full Implementation MIR is given in the Mars documentation [27].

A major difference between the code representation in the actual Mars compiler and the MIR described in this thesis is that the implementation uses basic blocks and branching for control flow [57], whereas in this thesis, we operate on the source-level statements, such as if, switch and while. The difference between these two representations is demonstrated by Example 5.14.

The basic block form used by the real compiler much more closely resembles an assembly language, and it also enforces the static single assignment (SSA) restriction, which is common for code analysis tools [17]. Under SSA, each variable may be assigned at precisely one program point. While variables in SSA form may be assigned multiple times on any given run of the program, there is only one statement that is ever responsible for its assignment. To allow for the fact that branching code may need to access the value of a variable regardless of which code path caused it to be assigned, $\phi$ nodes are used to combine the values of two or more variables, depending on which one was assigned.

Example 5.14 MIR as described in this thesis (a) and as used in the implementation (b)

```
def abs(x):
    t = x < 0
    if t:
        r = -x
    else:
        r = x
    return r

def abs(x):
    block 0:
        t = x < 0
        if t goto 1 else goto 2
    block 1:
        r₁ = -x
        goto 3
    block 2:
        r₂ = x
        goto 3
    block 3:
        r₃ = $\phi$(1 : r₁, 2 : r₂)
    return r₃
```

Example 5.15 shows the factorial function from a previous example converted to Implementation MIR. Note how the $\phi$ syntax from Thesis MIR while loop construct maps onto $\phi$ nodes in the SSA form.

The rationale for the notational difference is that the basic block form was more appropriate for the implementation, but presented an unnecessarily messy semantics to work
with the purpose of describing the static analysis. Using basic blocks and SSA form made it easier to manipulate code in the compiler, and allowed more straightforward generation of LLVM bitcode (which is also in SSA form [57]). However, we found that code examples, and the semantics of the language, are much clearer and more concise in a form closer to the source code (contrast Example 5.14a with Example 5.14b).

However, it is important that the algorithm described here is equivalent to the one implemented, so we must roughly demonstrate the correspondence between the two representations. The specific differences between the two notations are:

- Thesis MIR does not have the SSA restriction (instead, it has the weaker constraint of Rule 5.1 — that no variable may be assigned twice on the same code path).

- Thesis MIR allows if, switch and while statements to have nested statement blocks. In Implementation MIR, statement blocks are explicitly numbered. The if and switch statements contain block numbers instead of nested statement blocks, and there is no while statement.

- Implementation MIR features switch statements over numeric values, performing exact matches on the number values. This allows for a more efficient implementation, but has exactly the same semantics as nested if-else statements, so it is omitted here for simplicity.

- Implementation MIR features \( \phi \) statements, which select a variable based on the block that was previously executing (necessary due to the SSA restriction). Thesis MIR only features a limited \( \phi \) syntax for while loops.

Example 5.15 The factorial function in Thesis MIR (a) and Implementation MIR (b)

<table>
<thead>
<tr>
<th>Thesis MIR</th>
<th>Implementation MIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>def factorial(n):</td>
<td>def factorial(n):</td>
</tr>
<tr>
<td>( x = 1 )</td>
<td>( \text{block 0:} )</td>
</tr>
<tr>
<td>while ( n_1 ) ( [x_1 = \phi(x, x'_1), n_1 = \phi(n, n'_1)] ):</td>
<td>( x = 1 )</td>
</tr>
<tr>
<td>( x'_1 = x_1 \times n_1 )</td>
<td>( \text{goto 1} )</td>
</tr>
<tr>
<td>( n'_1 = n_1 - 1 )</td>
<td>( \text{block 1:} )</td>
</tr>
<tr>
<td>return ( x_1 )</td>
<td>( x_1 = \phi(0 : x, 2 : x'_1) )</td>
</tr>
<tr>
<td></td>
<td>( n_1 = \phi(0 : n, 2 : n'_1) )</td>
</tr>
<tr>
<td></td>
<td>if ( n_1 ) goto 2 else goto 3</td>
</tr>
<tr>
<td></td>
<td>block 2:</td>
</tr>
<tr>
<td></td>
<td>( x'_1 = x_1 \times n_1 )</td>
</tr>
<tr>
<td></td>
<td>( n'_1 = n_1 - 1 )</td>
</tr>
<tr>
<td></td>
<td>goto 1</td>
</tr>
<tr>
<td></td>
<td>block 3:</td>
</tr>
<tr>
<td></td>
<td>return ( x_1 )</td>
</tr>
</tbody>
</table>
5.5. ABSTRACT SYNTAX

- Implementation MIR preserves the io annotation. In this thesis, we simply drop the annotation for simplicity.

- Implementation MIR includes support for the impure language features described in Section 4.2.7. This is irrelevant for analyses, because the compiler is allowed to assume functions have no side-effects.

With the two notations being roughly equivalent, it is a relatively simple matter to translate any algorithm from one to the other. Therefore, we use the Thesis MIR in all of the discussions to come.

5.5 Abstract syntax

What follows is the complete abstract syntax for MIR, with an explanation for each production.

\[
\begin{align*}
\text{Int} & \rightarrow [-]Digit^* \\
\text{Num} & \rightarrow \text{Int}[.-]Digit^* \\
\text{Atom} & \rightarrow \text{Num} | \text{Var} \\
\text{Expr} & \rightarrow \text{Atom} \quad \text{(number literal or variable)} \\
& \quad | \text{CGCName} \quad \text{(CGC reference)} \\
& \quad | \text{Var} \cdot \text{CtorName} / \text{Int} \quad \text{(field access)} \\
& \quad | \text{[Atom^*]} \quad \text{(array construction)} \\
& \quad | \text{CtorName(Atom^*)[\{Var\}]} \quad \text{(hinted constructor call)} \\
& \quad | \text{Var(Atom^*)} \quad \text{(closure call)} \\
& \quad | \text{FuncName(Atom^*)} \quad \text{(global function call)} \\
& \quad | \text{CTName\{Atom^\}} \quad \text{(closure template reification)} \\
\text{Stmt} & \rightarrow \epsilon \quad \text{(pass)} \\
& \quad | \text{Stmt ; Stmt} \quad \text{(statement sequence)} \\
& \quad | \text{Var = Expr} \quad \text{(assignment)} \\
& \quad | \text{if Atom: Stmt else: Stmt} \quad \text{(if statement)} \\
& \quad | \text{switch Var: (CtorName : Stmt)^*} \quad \text{(switch statement)} \\
& \quad | \text{while Var: [(Var = \phi(Var, Var))^*]} : \text{Stmt} \quad \text{(while loop)} \\
\text{FuncDef} & \rightarrow \text{def CGCName:} \quad \text{(CGC definition)} \\
& \quad | \text{Stmt} \\
& \quad | \text{return Expr} \\
& \quad | \text{def FuncName(Var^*):} \quad \text{(function definition)} \\
& \quad | \text{Stmt} \\
& \quad | \text{return Expr} \\
& \quad | \text{def CTName\{Var^*\}(Var^*):} \quad \text{(closure template definition)} \\
& \quad | \text{Stmt} \\
& \quad | \text{return Expr} \\
\text{Program} & \rightarrow \text{FuncDef^*}
\end{align*}
\]
Note that in examples, a statement sequence is usually denoted with a newline rather than a semicolon, and the else of an if statement is optional (if omitted, it defaults to “else: ε”).

### 5.6 Concrete value denotational semantics

This section gives the full denotational semantics for MIR. Such a semantics can be used as the basis for an interpreter (in fact, it can almost be treated as pseudo-code for an interpreter), but also as the basis for static analysis. It is called the “value semantics” because it specifies only the values to be produced, and is not concerned with memory allocation or aliasing (addressed later, in Section 5.7).

#### 5.6.1 Preliminaries

The value \( \bot \) represents an error condition or the result of a non-terminating function. \( S_\bot \) is short-hand for \( S \cup \{\bot\} \). \( f! \) denotes a version of a function \( f \) that is strict in all arguments: if \( f \) takes one argument, \( f! \) is defined as follows:

\[
f! x = \begin{cases} 
\bot, & \text{if } x = \bot \\
f x, & \text{otherwise}
\end{cases}
\]

The expression \( f[a \mapsto b] \) means the same as \( f[a \mapsto \bot] \), but is strict in all arguments (in particular, if \( b \) is \( \bot \), it replaces the entire map with the value \( \bot \) ). \( \overline{f} \) denotes a total version of a partial function \( f \) that results in \( \bot \) wherever \( f \) is undefined. It is defined as follows:

\[
\overline{f} x = \begin{cases} 
\bot, & \text{if } f x \text{ is undefined} \\
f x, & \text{otherwise}
\end{cases}
\]

The value domain, \( \text{Val} \), is ordered by information content; that is, \( \bot \) is smaller than everything else in \( \text{Val} \), arrays of equal length are ordered according to the order of their corresponding elements, and other unequal values are incomparable. Thus \( \text{Val} \) is a complete partial order.

#### 5.6.2 Domains

Let \( \text{Var} \), \( \text{CGCName} \), \( \text{FuncName} \), \( \text{CTName} \), and \( \text{CtorName} \) be the set of all variable, computable global constant, function, closure template and data constructor names, respectively. These sets are all disjoint, each containing only the finite set of names declared in each context for a given program.
5.6. CONCRETE VALUE DENOTATIONAL SEMANTICS

\[ \text{Val} = \mathbb{Q} \cup (\text{CtorName} \times \text{Val}^*) \cup [\text{Val}^*] \cup \text{Closure} \]
\[ \text{Closure} = \text{Val}^* \rightarrow \text{Val}_{\bot} \]
\[ \text{Env} = \text{Var} \rightarrow \text{Val} \]
\[ \text{CGC} = \text{Val}_{\bot} \]
\[ \text{Func} = \text{Val}^* \rightarrow \text{Val}_{\bot} \]
\[ \text{CT} = \text{Val}^* \rightarrow \text{Closure} \]
\[ \text{Den} = (\text{CGCName} \cup \text{FuncName} \cup \text{CTName}) \rightarrow (\text{CGC} \cup \text{Func} \cup \text{CT}) \]

The domain Val represents all values — numbers, constructed terms, arrays\(^{12}\) and closures. The domain Env represents a local variable environment — a mapping from variable names to values. The domain Den represents the semantics of a program. It is a combined mapping from CGC names to values, from function names to functions, and from closure template names to closure templates.

The domain CtorName\(_T\) is a subset of CtorName representing the set of constructor names for type \(T\).

5.6.3 Semantic functions

We first give a denotational semantics for the language. Here, we define \(\Theta\) (the semantics of the primitive functions), \(\mathcal{A}\) (atom semantics), \(\mathcal{E}\) (expression semantics), \(\mathcal{S}\) (statement semantics), \(\mathcal{F}\) (function semantics) and \(\mathcal{P}\) (program semantics). These functions assume that the program is well-formed (e.g., is type checked, only refers to procedures that exist and variables that are assigned, etc.). The definitions are explained in Section 5.6.5.

\[ \Theta : \text{Den} \]
\[ \mathcal{A} : \text{Atom} \rightarrow \text{Env} \rightarrow \text{Val} \]
\[ \mathcal{E} : \text{Expr} \rightarrow \text{Den} \rightarrow \text{Env} \rightarrow \text{Val}_{\bot} \]
\[ \mathcal{S} : \text{Stmt} \rightarrow \text{Den} \rightarrow \text{Env} \rightarrow \text{Env}_{\bot} \]
\[ \mathcal{F} : \text{FuncDef} \rightarrow \text{Den} \rightarrow \text{Den} \]
\[ \mathcal{P} : \text{Program} \rightarrow \text{Den} \]

\(^{12}\)[Val\(^*\)] refers to an array of values, semantically equivalent to a tuple (Val\(^*\)) but distinguished as it is treated differently in the analysis (tuples have a statically fixed width, while arrays are dynamically sized).
\[ \Theta = \{ f \mapsto \text{semantics of } f : \\
\text{f is the name of a primitive function} \} \]

\[ \mathcal{A}[n] \sigma = n, \text{ if } n \in \text{Num} \]
\[ \mathcal{A}[v] \sigma = \sigma v, \text{ if } v \in \text{Var} \]
\[ \mathcal{E}[a] \rho \sigma = \mathcal{A}[a] \sigma, \text{ if } a \in \text{Atom} \]
\[ \mathcal{E}[c] \rho \sigma = \overline{c}, \text{ if } c \in \text{CGCName} \]
\[ \mathcal{E}[s \cdot c / n] \rho \sigma = (\sigma s)_{(n+1)} \]
\[ \mathcal{E}[[a_1, \ldots, a_n]] \rho \sigma = (\mathcal{A}[a_1] \sigma, \ldots, \mathcal{A}[a_n] \sigma) \]
\[ \mathcal{E}[c(a_1, \ldots, a_n)] \text{ hint} \rho \sigma = (c, \mathcal{A}[a_1] \sigma, \ldots, \mathcal{A}[a_n] \sigma), \text{ if } c \in \text{CtorName} \]
\[ \mathcal{E}[v(a_1, \ldots, a_n)] \rho \sigma = (\sigma v)(\mathcal{A}[a_1] \sigma, \ldots, \mathcal{A}[a_n] \sigma), \text{ if } v \in \text{Var} \]
\[ \mathcal{E}[f(a_1, \ldots, a_n)] \rho \sigma = (\overline{p} f)(\mathcal{A}[a_1] \sigma, \ldots, \mathcal{A}[a_n] \sigma), \text{ if } f \in \text{FuncName} \]
\[ \mathcal{E}[f\{a_1, \ldots, a_n\}] \rho \sigma = (\overline{p} f)(\mathcal{A}[a_1] \sigma, \ldots, \mathcal{A}[a_n] \sigma) \]
\[ S[e] \rho = \text{id} \]
\[ S[s_1 ; s_2] \rho = (S[s_1] \rho) \circ (S[s_2] \rho) \]
\[ S[v = e] \rho = \lambda \sigma. \sigma[v \mapsto e] \sigma \rho \]
\[ S[\text{if } a: \text{then } s_1 \text{ else } s_2] \rho = \lambda \sigma. \begin{cases} \\
\bot, & \text{if } \mathcal{A}[a] \sigma = \bot \\
S[s_1] \rho \sigma, & \text{if } \mathcal{A}[a] \sigma \neq \bot \text{ and } S[s_2] \rho \sigma, \text{ otherwise} \\
\end{cases} \]
\[ S[\text{switch } v: (c_1 : s_1, \ldots, c_n : s_n)] \rho = \lambda \sigma. \begin{cases} \\
\bot, & \text{if } \mathcal{A}[a] \sigma = \bot \\
S[s_1] \rho \sigma, & \text{if } (\sigma v)_0 = c_1 \\
\vdots & \\
S[s_n] \rho \sigma, & \text{if } (\sigma v)_0 = c_n \\
\bot, & \text{otherwise} \\
\end{cases} \]
\[ S[\text{while } a [u_1 = \phi(v_1, w_1), \ldots]; s] \rho = \lambda \sigma. w \sigma[u_1 \mapsto v_1, \ldots] \]

where \( w = \text{lfp } \lambda h. \lambda \sigma. \begin{cases} \\
\bot, & \text{if } \mathcal{A}[a] \sigma = \bot \\
\text{let } \sigma' = S[s] \rho \sigma \text{ in } h \sigma'[u_1 \mapsto v_1, w_1, \ldots], & \text{if } \mathcal{A}[a] \sigma \neq \bot \\
\sigma, & \text{otherwise} \\
\end{cases} \]
\[ \mathcal{F}[f \, s \, e] \rho = \{ f \mapsto \mathcal{E}_1[e] \rho \, (S[s] \rho \Theta) \} \]
\[ \mathcal{F}[f(a_1, \ldots, a_n) \, s \, e] \rho = \{ f \mapsto \lambda(x_1, \ldots, x_n). \]
\[
\begin{aligned}
\text{let } \sigma &= \left\{ \begin{array}{l}
a_1 \mapsto x_1, \\
a_n \mapsto x_n \end{array} \right.
\end{aligned}
\]
\[
\text{in } \mathcal{E}_1[e] \rho \, (S[s] \rho \sigma)
\]
\[ \mathcal{F}[f\{v_1, \ldots, v_m\}(a_1, \ldots, a_n) \, s \, e] \rho = \{ f \mapsto \lambda(w_1, \ldots, w_m). \lambda(x_1, \ldots, x_n).
\begin{aligned}
\text{let } \sigma &= \left\{ \begin{array}{l}
v_1 \mapsto w_1, \\
v_m \mapsto w_m, \\
a_1 \mapsto x_1, \\
a_n \mapsto x_n \end{array} \right.
\end{aligned}
\]
\[
\text{in } \mathcal{E}_1[e] \rho \, (S[s] \rho \sigma)
\]
\[ \mathcal{P}[p] = \text{lfp} \left( \bigsqcup_{f \in p} (\mathcal{F} f) \sqcup \lambda \rho. \Theta \right) \]

### 5.6.4 Primitives

The semantic value \( \Theta \) denotes a mapping from function names to each primitive function. This simplifies the main semantics by offloading a lot of operations to separate functions. For example, basic arithmetic operations are considered to be primitives (we do not explicitly define these). It also leaves the language open for any desired primitive. Here we only discuss a few important ones.

\[ \text{add}(x, y) = x + y \]
\[ \text{eq:Num}(x, y) = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{otherwise} \end{cases} \]
\[ \text{array}(n, v) = (v, v, \ldots, v) \]
\[ \text{array_length}(a) = |a| \]
\[ \text{array_ref}(a, i) = a_i \]
\[ \text{array_replace}(a, i, v) = (a_0, \ldots, a_{i-1}, v, a_{i+1}, \ldots, a_{\|a\|}) \]
\[ \text{array_add}(a, v) = (a_0, \ldots, a_{\|a\|}, v) \]
\[ \text{error}(msg) = \bot \]

### 5.6.5 Discussion

An explanation of the value semantics follows.

The Val domain is the set of all values that a variable can take: the union of all rational numbers, terms (tuples whose first element is a constructor name, with zero or more subsequent values), arrays (tuples of values, distinguished from terms using the special
function symbol, “[ ]”), and functions (mappings from tuples of values onto values). The Val domain does not contain ⊥, so it is not legal for ⊥ to appear except in the augmented Val⊥ domain.

For the numbers, note that we use the symbol Q denoting the set of rationals, while the implementation uses a subset of Q: the IEEE 754 double-precision floating point numbers [1], excluding NaN and infinity values. We do not concern ourselves with the rounding involved, and for the semantics, consider the numeric values to be countably infinite.

The Env domain (members usually given the name σ) represents the current state of the local variables during the evaluation of a function. It maps each variable name onto a member of the Val domain. Env functions are partial, as not all variable names map onto values. However, its range does not include ⊥. As MIR is a strict language, any non-terminating functions or errors do not return at all, so variables cannot hold the value ⊥. Note that any time the σ map is updated, the special ↦→! operator is used, to prevent ⊥ values from entering the map.

The Den domain (members usually given the name ρ) represents the semantics of an entire program — a mapping from constant, function and closure template names to their corresponding semantics. The semantics of a computable global constant is simply its final value. The semantics of a function is a function in the meta-language. The semantics of a closure template is a meta-language function that takes a tuple of values (the free variables) and produces a function. As with Env, a Den is a partial function, but cannot produce ⊥ values. Θ is a Den containing the semantics of all primitive functions in Mars.

The rest of the semantics consists of semantic functions, named by script capital letters. The semantics of an atom (A) is a function of the atom and the environment (σ). For example, the semantics of a variable atom is found by looking up the variable name in the environment. Note that, unlike Env, A is not a partial function. This is because, as stated earlier, we assume it will only be used for well-formed programs, which means that the σ map lookup will always succeed. Also, A cannot produce a ⊥ result. The semantics of an expression (E) describes how to compute the result of any expression. E uses ρ to ensure that when functions are undefined on the first fixed-point iteration, we still obtain a well-defined result. Since expressions can call functions or refer to computable global constants, the result of evaluating an expression may be ⊥.

The semantics of a statement (S) describes how the environment is updated. Two rules enforce the strictness of the MIR language: the sequence rule takes the semantics of the first statement with S[S1], then passes that resulting environment to S[S2]. The strict version of S ensures that if the first statement is ⊥, the result of the sequence is also ⊥. Similarly, the assignment statement uses the strict version of the update operator to
ensure that if the expression is $\bot$, the entire environment becomes $\bot$ — hence, any error or non-termination in an expression causes the entire function to fail. The **while** loop semantics is computed by least fixed point. First, the loop variables $u_1, \ldots, u_n$ are given their initial values from $v_1, \ldots, v_n$. The $lfp$ computes the least function $h$ that satisfies the recursive definition: a function that checks whether the condition variable $a$ is non-zero, and if so, updates the environment according to the body $s$, updates the loop variables $u_1, \ldots, u_n$, then recurses on $h$. The least fixed point of this function is one that recurses indefinitely until the condition is false, having the value $\bot$ if the loop never terminates.

The semantics of a computable global constant, function or closure template ($F$) is a singleton map that maps the name onto the semantics. In all cases, the result is the semantics of the procedure’s statement passed to the semantics of the procedure’s result expression — note the use of $E$ to ensure that if the statement is $\bot$, the function’s result is also. The semantics of the program itself (defined by $P$) is the least fixed point ($lfp$) of the least upper bound of the semantics of all function definitions in the program. Since $\bigcup_{f \in P} (F f)$ is monotonic, its least fixed point is well-defined. Like the **while** loop, this allows the semantics of finite recursive functions to be well-defined. When evaluating the least fixed point via Kleene iteration (using Algorithm 1.1), all functions are initially evaluated with a denotation of $\rho = \bot$, so any function that calls any other function will fail. On subsequent passes, the program becomes successively more defined, until eventually it represents all finite recursion. Infinite recursion results in the semantics of $\bot$. Note that for computable global constants, this behaviour is *non-strict* (an erroneous CGC will not crash the program until it is evaluated for the first time).

It is important to note that MIR, like Mars, exhibits interface integrity, as given in Definition 3.5. This can be seen by the fact that the function semantics does not receive or emit any kind of hidden “heap”; the effective interface of a function is the same as its apparent interface.

### 5.6.6 Example

To fully explore the workings of the concrete value semantics, we work through a complete example: the imperative version of *factorial*, the source code given in Example 5.16.

The concrete semantics essentially defines a mapping from the abstract syntax of Example 5.16 to a mathematical function. We can begin by applying the semantic functions


**Example 5.16** Imperative factorial function in MIR

```python
def factorial(n):
    x = 1
    while n1 [x1 = φ(x, x'), n1 = φ(n, n')]:
        x' = x1 * n1
        n' = n1 - 1
    return x1
```

...to the individual statements to obtain the semantics of each statement:

\[
S[x = 1] = \lambda \rho \sigma. \sigma[[x] \mapsto 1]
S[x' = mul(x1, n1)] = \lambda \rho \sigma. \sigma[[x'] \mapsto \sigma[[x1]] \times \sigma[[n1]]]
S[n' = sub(n1, 1)] = \lambda \rho \sigma. \sigma[[n'] \mapsto \sigma[[n1]] - 1]
\]

The semantics for the entire while-loop body, which we shall abbreviate as `body`, is:

\[
body = \lambda \rho \sigma. \sigma[[x'] \mapsto \sigma[[x1]] \times \sigma[[n1]], [n'] \mapsto \sigma[[n1]] - 1]
\]

The semantics for the while loop itself, which we shall abbreviate as `loop`, is:

\[
\text{loop } = \lambda \rho \sigma. \text{w} \sigma[[x1] \mapsto \sigma[[x]], [n1] \mapsto \sigma[[n]]]
\]

where `w = lfp λh. λσ. \left( \begin{array}{l}
\text{let } \sigma' = \text{body } \rho \sigma \\
\text{in } \text{w} \sigma'[[x1] \mapsto \sigma'[[x1]], [n1] \mapsto \sigma'[[n1]]]
\end{array} \right), \text{ if } \sigma[[n1]] \neq 0
\]

The `lfp` (least fixed point) operator takes a function `f` whose first parameter is `h` and finds the least value `h` for which `f h = h`. Wherever the recursion would not terminate, the result is `⊥`. This means we can rewrite `w` as:

\[
w \sigma = \left( \begin{array}{l}
\text{let } \sigma' = \text{body } \rho \sigma \\
\text{in } \text{w} \sigma'[[x1] \mapsto \sigma'[[x1]], [n1] \mapsto \sigma'[[n1]]]
\end{array} \right), \text{ if } \sigma[[n1]] > 0
\]

\[
\text{⊥, if } \sigma[[n1]] = 0
\]

The semantics for the whole function can now be determined by composing the semantics for its two statements and return expression:

\[
F[[\text{factorial}(n) \ (x = 1; \text{while } n1 \ldots) \ x]] \rho = \{[[\text{factorial}] \mapsto \text{fac}]\}
\]
where \( \text{fac } n = \begin{cases} \text{let } \text{fac}' n' x = & \begin{cases} \text{fac}' (n' - 1) (x \times n'), & \text{if } n' > 0 \\ x, & \text{if } n' = 0 \\ \bot, & \text{otherwise} \end{cases} \\ \text{in } \text{fac}' n 1 \end{cases} \)

The resulting semantics for the factorial function is a clean mathematical definition of \( n! \), in accumulator recursion style and with an explicit case to map negative inputs to \( \bot \).

Finally, the program semantics is merely the union of the factorial semantics with the other primitives. Fixed-point iteration is generally required, but is not required here because there are no recursive calls, so the final program semantics is the mapping \( \{ [\text{factorial}] \mapsto \text{fac} \} \sqcup \Theta \).

### 5.7 Concrete memory denotational semantics

We now give a second complete semantics for MIR. This time, we explicitly track the memory usage of each procedure: which values are sharing the same memory cells and which values are independent of each other.

In a language without interface integrity (Definition 3.5), this information is vital in the core language semantics, since if two objects are aliased, the program may compute a different result than it would if they were not. However, since MIR does exhibit interface integrity, the value semantics presented above is sufficient to model the behaviour of programs in the language, despite the fact that it does not distinguish between aliased and non-aliased objects. That said, we need a semantic model that preserves aliasing information in order to describe an analysis of aliasing in programs.

For example, consider two variables \( x = [1, 2, 3] \) and \( y = [1, 2, 3] \). At runtime, \( x \) and \( y \) could be pointers to the same memory location, or they could be independent pointers to two separate memory locations, each with a separate copy of the array. Our value semantics does not distinguish between the two possibilities. To describe an aliasing analysis, we need a semantics that does.

The memory semantics needs to make some assumptions about the implementation that are not part of the language specification. For example, the specification does not state whether evaluating an array or string literal expression allocates a fresh array each time, or whether the literal always produces a reference to the same static memory cells. Nor does it need to specify this — the program result is the same regardless. However, the decision on this matter will have implications for our aliasing analysis (if array literals are fresh, we will be able to safely mutate a variable that was assigned a literal; if not,
we will have to make a copy). Similarly, the language does not state whether arrays and structures are copied by reference, or if a full deep copy is made, but that decision is quite important when discussing the memory model of the program. Thus, these implementation decisions must be encoded into the memory semantics, which will in turn affect the aliasing analysis. Note that the memory semantics, and hence the analysis, will not be appropriate for all MIR implementations, only those with the same allocation policy as ours. In our implementation, array literals cause a fresh allocation (and hence $x$ and $y$ in the above example are not aliased), but arrays and structures are always copied by reference, resulting in aliases.\footnote{Copies of data structures are only created when the structure is updated, e.g., by array\_replace.}

One assumption we make about the implementation is that numbers will be \textit{unboxed}, meaning variables of type Num directly contain their values, instead of a reference to a memory cell.\footnote{Because numbers are double-precision floating point numbers, and Mars variables can be polymorphic, the compiler stores all values in 64-bit words, even on 32-bit architectures.} This allows us to ignore memory sharing between numbers.

### 5.7.1 Domains

Let Ref be a denumerable set of opaque values. These could be natural numbers, memory addresses,\footnote{The compiler obviously has a finite number of memory addresses available. This semantic model assumes infinite memory; out-of-memory errors and garbage collection are outside the scope of the semantics.} or unique strings, for example. The semantic model of memory allocation relies on this denumerability (it works by selecting an arbitrary element of Ref that has never been selected before).

The domain Object contains all of the “boxed” values — under this model, these values can no longer be directly stored in variables, and must be pointed at by a Ref. These include constructed terms, arrays and closures. An Object may also be the special value \textit{unevaluated}, which represents a CGC that has not been evaluated. The domain Val now only contains the unboxed numbers, and Ref.

We introduce a Heap (commonly denoted by $\eta$) which contains a mapping from Ref to the Object that it points to. A Heap is required to access the contents of any non-numeric value.
5.7. CONCRETE MEMORY DENOTATIONAL SEMANTICS

\[
\begin{align*}
\text{Val} & = \mathcal{Q} \cup \text{Ref} \\
\text{Object} & = (\text{CtorName} \times \text{Val}^*) \cup [\text{Val}^*] \cup \text{Closure} \cup \{\text{unevaluated}\} \\
\text{Closure} & = \text{Ref} \times \text{CT} \\
\text{Heap} & = \text{Ref} \rightarrow \text{Object} \\
\text{Env} & = \text{Var} \rightarrow \text{Val} \\
\text{State} & = \text{Heap} \times \text{Env} \\
\text{CGC} & = \text{Heap} \rightarrow (\text{Heap} \times \text{Val})_\bot \\
\text{Func} & = \text{Heap} \rightarrow \text{Val}^* \rightarrow (\text{Heap} \times \text{Val})_\bot \\
\text{CT} & = \text{Heap} \rightarrow \text{Val}^* \rightarrow \text{Val}^* \rightarrow (\text{Heap} \times \text{Val})_\bot \\
\text{Den} & = (\text{CGCName} \cup \text{FuncName} \cup \text{CTName}) \rightarrow (\text{CGC} \cup \text{Func} \cup \text{CT})
\end{align*}
\]

5.7.2 Semantic functions

\[
\begin{align*}
\Theta & : \text{Den} \\
\mathcal{A} & : \text{Atom} \rightarrow \text{Env} \rightarrow \text{Val} \\
\mathcal{E} & : \text{Expr} \rightarrow \text{Den} \rightarrow \text{State} \rightarrow (\text{Heap} \times \text{Val})_\bot \\
\mathcal{S} & : \text{Stmt} \rightarrow \text{Den} \rightarrow \text{State} \rightarrow \text{State}_\bot \\
\mathcal{F} & : \text{FuncDef} \rightarrow \text{Den} \rightarrow \text{Den} \\
\mathcal{P} & : \text{Program} \rightarrow \text{Den}
\end{align*}
\]

The following helper functions are used by the semantics:

\[\text{alloc} : \text{Heap} \rightarrow \text{Object} \rightarrow \text{Heap} \times \text{Ref},\] given heap \(\eta\) and object \(o\), creates a new Ref, \(r\), such that \(r \notin \eta\), and returns \((\eta[r \mapsto o], r)\).

When evaluating the result of calling a function at the top level, we must supply a valid initial heap \((\eta_0)\). The initial heap must contain an existing reference for every computable global constant in the program, each mapped onto \text{unevaluated} — this provides a storage location for the result of the CGC, so that on subsequent evaluations the result can be returned without recomputation.

\[\text{cgcref} : \text{CGCName} \rightarrow \text{Ref},\] given computable global constant name \(n\), returns the opaque token \(r\) that is unique to the name \(n\), and guaranteed to map onto the value
unevaluated in the initial heap \( \eta_0 \).

\[
\Theta = \{ f \mapsto \text{semantics of } f : f \text{ is the name of a primitive function} \}
\]

\[
A[n] \sigma = n, \text{ if } n \in \text{Num}
\]

\[
A[v] \sigma = \sigma v, \text{ if } v \in \text{Var}
\]

\[
E[a \in \text{Atom}] \rho(\eta, \sigma) = (\eta, A[a] \sigma)
\]

\[
\text{let } x = \text{cgref } c
\]

\[
E[c \in \text{CGCName}] \rho(\eta, \sigma) = \begin{cases} 
(\eta', r) = (\overline{\rho} c) \eta & \text{if } \eta x = \text{unevaluated} \\
(\eta', [x \mapsto r], r), & \text{otherwise}
\end{cases}
\]

\[
E[s \cdot c / n] \rho(\eta, \sigma) = \text{let } v = \eta \circ \sigma s \text{ in } (\eta, v_{(n+1)})
\]

\[
E[[a_1, \ldots, a_n]] \rho(\eta, \sigma) = \text{alloc } \eta (A[a_1] \sigma, \ldots, A[a_n] \sigma)
\]

\[
E[[c(a_1, \ldots, a_n) \text{ hint}] \rho(\eta, \sigma) = \text{alloc } \eta (c, A[a_1] \sigma, \ldots, A[a_n] \sigma)
\]

\[
E[[v(a_1, \ldots, a_n)] \rho(\eta, \sigma) = \text{let } (c, f) = (\eta \circ \sigma v) \text{ in } f = \overline{\rho} f
\]

\[
E[[f \in \text{Var}(a_1, \ldots, a_n)] \rho(\eta, \sigma) = (\overline{\rho} f) \eta (A[a_1] \sigma, \ldots, A[a_n] \sigma)
\]

\[
E[[f \in \text{FuncName}(a_1, \ldots, a_n)] \rho(\eta, \sigma) = (\eta', c) = \text{alloc } \eta (A[a_1] \sigma, \ldots, A[a_n] \sigma)
\]

\[
S[e] \rho = \text{id}
\]

\[
S[s_1 ; s_2] \rho = (S[s_2] \rho) \circ (S[s_1] \rho)
\]

\[
S[v = e] \rho = \lambda(\eta, \sigma). \text{ in } (\eta', \sigma[v \mapsto r])
\]

\[
S[\text{if } a: \text{then } s_\text{then } \text{else } s_\text{else}] \rho = \lambda(\eta, \sigma). \begin{cases} 
\bot, & \text{if } A[a] \sigma = \bot \\
S[s_\text{then}] \rho(\eta, \sigma), & \text{if } A[a] \sigma \neq 0 \\
S[s_\text{else}] \rho(\eta, \sigma), & \text{otherwise}
\end{cases}
\]

\[
S[\text{switch } v: (c_1 : s_1, \ldots, c_n : s_n)] \rho = \lambda(\eta, \sigma). \begin{cases} 
\bot, & \text{if } A[a] \sigma = \bot \\
S[s_i] \rho(\eta, \sigma), & \text{if } (\eta \circ \sigma v)_{0} = c_i \\
\ldots & \text{otherwise} \\
S[s_n] \rho(\eta, \sigma), & \text{if } (\eta \circ \sigma v)_{0} = c_n \\
\bot, & \text{otherwise}
\end{cases}
\]

\[
S[\text{while } a [u_1 = \phi(v_1, w_1), \ldots] : s] \rho = \lambda(\eta, \sigma). w (\eta, \sigma[u_1 \mapsto v_1, \ldots])
\]

where \( w = \text{lfp } \lambda h. \lambda(\eta, \sigma). \begin{cases} 
\bot, & \text{if } A[a] \sigma = \bot \\
h (S[s] \rho(\eta, \sigma))[u_1 \mapsto v_1, \ldots], & \text{if } A[a] \sigma \neq 0 \\
(\eta, \sigma), & \text{otherwise}
\end{cases}
\]
\( \mathcal{F}[f \cdot e] \rho = \{ f \mapsto \lambda \eta. \mathcal{E}_1[e] \rho (S[s] \rho(\eta, \emptyset)) \} \)

\( \mathcal{F}[f(a_1, \ldots, a_n) \cdot e] \rho = \{ f \mapsto \lambda \eta. (x_1, \ldots, x_n). \)

let \( \sigma = \{ a_1 \mapsto x_1, \ldots, a_n \mapsto x_n \} \)

in \( \mathcal{E}_1[e] \rho (S[s] \rho(\eta, \sigma)) \} \)

\( \mathcal{F}[f\{v_1, \ldots, v_m\}(a_1, \ldots, a_n) \cdot e] \rho = \{ f \mapsto \lambda \eta. (w_1, \ldots, w_m) (x_1, \ldots, x_n). \)

let \( \sigma = \{ v_1 \mapsto w_1, \ldots, v_m \mapsto w_m, \}

a_1 \mapsto x_1, \ldots, a_n \mapsto x_n \} \}

in \( \mathcal{E}_1[e] \rho (S[s] \rho(\eta, \sigma)) \} \)

\( \mathcal{P}[p] = \text{lfp} \left( \bigcup_{f \in p} (\mathcal{F}f) \sqcup \lambda \rho. \Theta \right) \)

### 5.7.3 Primitives

The primitives in the concrete memory semantics are as follows:

- **add \( \eta \) \((x, y)\) = \((\eta, x + y)\)**
- **eq::Num \( \eta \) \((x, y)\) = let \( r = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{otherwise} \end{cases} \)

  \text{in } (\eta, r)\)**
- **array \( \eta \) \((n, v)\) = alloc \( \eta \) \((v, v, \ldots, v)\)\)**
- **array_length \( \eta \) \((a)\) = \((\eta, |(\eta a)|)\)**
- **array_ref \( \eta \) \((a, i)\) = \((\eta, (\eta a)_i)\)**
- **array_replace \( \eta \) \((a, i, v)\) = alloc \( \eta \) \((a'_0, \ldots, a'_{i-1}, v, a'_{i+1}, \ldots, a'_{|a'|})\)\)**
  where \( a' = \eta a \)
- **array_add \( \eta \) \((a, v)\) = alloc \( \eta \) \((a'_0, \ldots, a'_{|a'|}, v)\)\)**
  where \( a' = \eta a \)
- **error \( \eta \) \((msg)\) = \bot**

### 5.7.4 Discussion

The major difference between the value and memory semantics is that the memory semantics moves most of the members of the value semantics’ Val domain into the new Object domain. The Val domain now consists only of \( \mathbb{Q} \) (for the unboxed numbers) and Ref values (effectively pointers or memory addresses), while the Object domain represents the contents of memory cells. Everything in this semantics has an extra level of indirection, allowing us to distinguish between objects that are aliased (have equal refer-
ences) and those that are merely equal. We also introduce the Closure domain, distinct from the Func domain (which is now reserved for global functions). In this semantics, we represent closures faithfully to the implementation: as a pair of a pointer to a tuple of free variable values and a closure template. We do not rely on the variable capture features of the meta-language to make closures work; we manually allocate memory for them and pass the free variable tuple to the closure template when they are called.

To make all of this work, we introduce a final domain, Heap (members usually given the name \( \eta \)), which maps references onto objects. The heap simulates the memory contents of the computer the program is run on, and is threaded into and out of most of the semantics. For example, the semantics of an expression \((E)\) includes a heap, giving expressions the opportunity to update the heap as well as producing a value.

Computable global constants are carefully defined as “thunks” — the heap is pre-populated with a special value “unevaluated” for every CGC, which causes the CGC expression to fully evaluate the CGC and modify the heap to store the result. Subsequent evaluation of the same CGC will simply return the same result from the heap — this ensures that all results of a CGC are aliases of the same object. Array expressions, constructor call expressions, and closure template reification expressions explicitly allocate new memory, so the result is never initially aliased to any other object.

The remainder of the semantics is essentially the same as the value semantics, except for the need to thread the singleton heap object through nearly everything.

### 5.8 Conclusion

In this chapter, we described the full, formal semantics for MIR, the intermediate language which Mars source code is transformed into for analysis and code generation. MIR is a very high-level language, quite similar to Mars itself, with just a handful of differences that make it easier to analyse.

MIR does not allow nested expressions or return statements other than one at the end of each function. It features a simplified switch statement which is not capable of numeric matching, variable binding or advanced pattern matching; it is the duty of the front-end compiler to deal with simplifying that advanced language construct. Similarly, MIR does not allow global functions to be treated as first-class values; they must be explicitly transformed into “closure templates” which may then be reified as function values. We showed a number of high-level language features and sketched out how the front-end compiler would translate those features into MIR code. We also showed how several constructs in other pure programming languages might be translated into MIR.
The semantics for MIR were given in two distinct versions: the basic value semantics, which is the most straightforward denotational semantics for the language, and a more advanced memory semantics. The simpler value semantics is necessary as a basis for making correct program modifications: when the compiler automatically optimises a program, we can tell whether we have correctly preserved the program’s meaning by comparing the original and modified versions against the value semantics. The memory semantics has the added property of explicitly tracking the memory allocations and sharing of memory cells by the real implementation of MIR. This extra information is crucial for any analysis that examines the aliasing of variables in the program, and so it is the memory semantics from which we will derive the aliasing analysis over the coming chapters.
Chapter 6

Abstract interpretation

6.1 Introduction

Before presenting our aliasing analysis, we briefly digress to formally introduce the notion of abstract interpretation by way of a simple example. The purpose of this chapter is twofold: to introduce or refresh the concept of abstract interpretation to the reader, and to demonstrate abstract interpretation using our notations, and applied to MIR.

Abstract interpretation is a systematic approach to static analysis, suitable in a wide array of situations. The technique was introduced by Cousot and Cousot [14] in 1977 as a way of unifying existing “apparently unrelated program analysis techniques.” This approach revolves around selecting an abstract domain, a simplified representation of the state of a program that can be explored in finite time. Converting a concrete state to the abstract domain loses information, but with a well-chosen domain, we can preserve a good approximation of the information we care about (e.g., which variables are aliased).

To take a basic example from [14], we may only be interested in whether a number is positive or negative, so we choose an abstract domain \{(0), (−), (+), (±)\}, ordered by information content. This forms a lattice as shown in Figure 6.1.

```
(±)
  / \  
(−)  (+)
  /  /
(0)
```

Figure 6.1: Hasse diagram for the domain \{(0), (−), (+), (±)\}
In this domain, given a variable $x$, if we are certain that $x = 0$ or could not have a concrete value (because of a contradiction; for example if it must be negative and must be positive, and therefore the code is unreachable), we give it the abstract value $(0)$ (the bottom of this lattice). If we are otherwise sure that $x \geq 0$, we give it the abstract value $(+)$). If we know that $x \leq 0$, we give it the abstract value $(-)$. Lastly, if we are unsure about whether $x$ is positive or negative, we give it the abstract value $(\pm)$ (the top of this lattice). Every abstract value subsumes the values beneath it in the lattice; for example, $(+)$ represents a superset of the values represented by $(0)$. Therefore, it is always valid (and sometimes necessary) to weaken an assumption by moving up the lattice.

The abstraction function, $\alpha$, maps sets of concrete values onto abstract values; for example, $\alpha \{0, 3, 8\} = (+)$ and $\alpha \{-4, 4\} = (\pm)$. Conversely, the concretisation function, $\gamma$, maps abstract values onto sets of concrete values; for example, $\gamma (+) = \{x : x \in \mathbb{Q} \land x \geq 0\}$. Note that $\gamma (\alpha S)$ is always a superset of $S$, and usually much larger — this is the loss of information when we move into the abstract domain.

The abstract interpretation is the set of semantic operations for our language, rewritten in terms of the abstract domain. For example, in the concrete domain, the $\times$ operator would be expressed as arithmetic multiplication. In our abstract domain, we define $\times$ such that $(+) \times (+) = (+)$, $(+) \times (-) = (-)$, $(0) \times _{\pm} = (0)$, and so on. This allows us to “execute” a program in the abstract domain, at compile time (given that the abstract domain meets certain criteria). The interpretation is consistent with the concrete interpretation if the result of all abstract operations is equal to (or greater than) the $\alpha$ of the corresponding concrete values; this is required for correctness. We use the result of this execution to learn static information about the program, which can be applied to compile-time optimisation, error checking, and various other things.

Because the aliasing analysis is quite complicated, it is worth going over an intermediate example first. To this end, this chapter presents a complete abstract interpretation for MIR that determines statically, for each term value (instance of an algebraic data type), which constructors it might have been built with. This information could be used to give a compile-time error if a field is accessed for a constructor that the compiler is not 100% certain was used to construct the variable. It could also be used to optimise switch statements — if a variable is known to have been built with a certain constructor, a switch on the variable can be eliminated and we can jump straight to the correct case.
6.2 The abstract domain

Recall that, in the denotational semantics for MIR, each variable stores a value which is an element of the set $\text{Val}$:

$$\text{Val} = \mathbb{Q} \cup (\text{CtorName} \times \text{Val}^*) \cup [\text{Val}^*] \cup (\text{Val}^* \rightarrow \text{Val}_\bot)$$

That is, each value is either a number, a term, an array, or a function. By Definition 6.1, the $\text{Env}$ domain, which maps variables onto values, is our concrete domain. Note that just by choosing the concrete domain, we have made some decisions about what information is important — by using the value semantics, we are unable to distinguish between variables that share the same memory location, and those that merely have the same value. If we needed that information (as we will in our aliasing analysis), we would choose the memory semantics as our concrete domain.

The next decision is to choose the abstract domain that satisfies Definition 6.2. This choice has a twofold purpose: to pare the information in the concrete domain down to the characteristics we are interested in for this particular analysis, and to reduce the problem space down to a manageable size (usually, but not necessarily, a finite set of states). Ideally, the chosen abstract domain will be a finite-height lattice, which will mean the analysis naturally completes in finite time (given that the interpretation function is monotonic), as shown in Section 1.6.3. If the abstract domain is not finite, the compiler will need to apply widening [14] to artificially raise the value of the abstract state towards $\top$, in order to guarantee termination. We do not discuss widening in this chapter, as our abstract domain has finite height. As all approximations lose information, it is up to the designer of the analysis to choose a domain which preserves enough information to be meaningful, and discards irrelevant information.

**Definition 6.1.** The concrete domain is the set of possible states of execution in the program.

**Definition 6.2.** An abstract domain is a set of values that form a lattice. Each element of the abstract domain represents a set of elements of the concrete domain.

In Section 1.6.3, we defined any power set as a lattice with $\subseteq = \subseteq$, thereby allowing ourselves to use sets of any kind of value as an abstract domain. The power set of an infinite set (e.g., $\wp(\mathbb{Z})$) has infinite height and is therefore a poor choice as an abstract domain (requiring widening). The power set of a finite set (e.g., $\wp(\{a, b, c\})$) has a finite height and, as such, is a suitable choice for an abstract domain.

For our simple analysis, our abstract domain, $\text{Env}_a$, will be a mapping from variables to abstract values:

$$\text{Env}_a = \text{Var} \rightarrow \text{Val}_a$$
We can assume that even though Var is the infinite set of all possible variable names, we will consider only a fixed set of names in analysing any given function. Therefore, Env, when analysing a specific function, is a finite set.

We will start with an abstract value domain consisting of the constructor names:

$$\text{Val}_a = \text{CtorName}$$

We will represent each term abstractly by its constructor name, and forget about the contents of its fields. For example, the concrete value $x = \text{Cons}(4, \text{Nil})$ is approximated abstractly as $x = \text{Cons}$, and the concrete value $y = \text{Nil}$ is approximated precisely as $y = \text{Nil}$. However, a problem arises when we do not know which constructor was used to form a list $z$: we cannot assign either abstract value for sure. Indeed, the domain CtorName is not a lattice, as it has no $\bot$ or $\top$ elements. In general, values need to be approximated as sets of constructor symbols, so we redefine the abstract value domain:

$$\text{Val}_a = \wp(\text{CtorName})$$

We find that we can now represent the unknown case abstractly as $z = \{\text{Nil}, \text{Cons}\}$. We can also handle the over-constrained possibility: if we know that $x$ must be a Cons term, and also that $x$ must be a Nil term, then we are in unreachable code (a candidate for dead code elimination) — we need to be able to approximate this case, so we represent it abstractly as $x = \emptyset$. We also approximate $\bot$ (the concrete value representing the result of a non-terminating computation) with the abstract value $\emptyset$.

There is one final problem with this domain: it does not provide us with a way to represent non-term values (which are part of our concrete value domain). Since for this analysis, we are interested only in the values of terms, we do not need to record any information at all about non-term values,\footnote{This assumes that we do not care about the values of terms inside structures. If we need to be able to analyse the values of terms that have been, for example, placed into a tuple and extracted again, we need a more comprehensive abstract domain. We will preserve this sort of information in the main aliasing analysis.} such as numbers, arrays and functions, but we still need some way to represent them. Therefore, we introduce a special value $\text{builtin}$, which represents all values of built-in types in the concrete value domain. Thus, the full abstract value domain follows:

$$\text{Val}_a = \wp(\text{CtorName} \cup \{\text{builtin}\})$$

As a power set domain, the lattice is naturally defined with $\subseteq = \subseteq$, $\cup = \cup$ and $\cap = \cap$. The natural bottom ($\bot$) is $\emptyset$, while the natural top ($\top$) is CtorName $\cup \{\text{builtin}\}$.

Unlike the case above, this is a \textit{finite-height} lattice: it has a height of $|\text{CtorName}| + 2$. 
6.2. THE ABSTRACT DOMAIN

(CtorName has arbitrarily many values, but the size of the set is fixed by the program.) The Val_α domain has \(2^{|\text{CtorName}|} + 1\) elements, so our abstract interpretation will terminate in finite time without the need for widening. Furthermore, because of static type checking, the size of the lattice is even further constrained depending on the type of each variable. Variables of a specific term type \(T\) can only be in \(\text{CtorName}_T\). Variables of non-term types can only be in \(\emptyset, \{\text{builtin}\}\). Variables of unknown types (due to polymorphism) can only be in \(\emptyset, \top\); the compiler can have a special symbol for \(\top\) instead of needing to store the full (potentially huge) set \(\text{CtorName} \cup \{\text{builtin}\}\).

For example, a variable of type Tree23, an algebraic type with constructors Nil, Two and Three, holds abstract values from the set \(\wp(\text{CtorName}_{\text{Tree23}}) = \wp(\{\text{Nil, Two, Three}\})\), which has a height of 4, and can be represented graphically by the Hasse diagram in Figure 6.2. Note that there is no need for other constructors or the special \(\text{builtin}\) value.

![Hasse diagram for the domain \(\wp(\{\text{Nil, Two, Three}\})\)](image)

The elements of the lattice are ordered by knowledge or information content. A value lower on the lattice makes a stronger statement than a value higher on the lattice. The abstract value \(\{\text{Two}\}\) is a fairly strong statement, as we know the corresponding concrete value is some Two node. By contrast, the abstract value \(\{\text{Two, Three}\}\) is a weaker statement, as all we know is that it is non-Nil, while \(\{\text{Nil, Two, Three}\}\) (the top) is the weakest statement for values of this type: we do not know anything about the value except that it is a Tree23 (and we already knew that from the type checker). The bottom of a lattice is the strongest statement of all. In this case, it makes a claim so strong that there must be a contradiction: here, \(\emptyset\) means “this is not Nil, it is not a Two node and it is not a Three node.” Thus, the concrete value must be \(\bot\) (i.e., this is unreachable code).
6.3 Abstraction and concretisation

We formalise the relationship between the concrete and abstract domains by defining functions to convert between concrete and abstract states. Definition 6.2 provides that each element of the abstract domain represents a set of elements of the concrete domain; the elements it represents are determined by the $\alpha$ and $\gamma$ functions.

The abstraction function, $\alpha$, converts a set of concrete states to a single abstract state. $\alpha$ must be monotonic. In our abstract domain, $\alpha$ belongs to the following set:

$$\alpha : \wp(\text{Env}) \rightarrow \text{Env}_\alpha$$

The fact that it takes a set and not just a single concrete state is important: we may know that the program is in one of several concrete states, and wish to find an abstract state to cover all of those possibilities. For example, we may know that the state of a program is either $\{x \mapsto \text{Nil}\}$ or $\{x \mapsto \text{Two(\text{Nil}, 12, \text{Nil})}\}$. This means the program is in the abstract state $\{x \mapsto \{\text{Nil, Two}\}\}$.

The concretisation function, $\gamma$, converts an abstract state to a set of concrete states:

$$\gamma : \text{Env}_\alpha \rightarrow \wp(\text{Env})$$

$\gamma$ must also be monotonic. Despite appearances, $\gamma$ is not the inverse of $\alpha$ (as $\alpha$ is not injective). It produces the set of every possible concrete state that could be abstracted to its input (which is very often infinite). For example, $\gamma \{x \mapsto \{\text{Two}\}\}$ gives the infinite set of all states where $x$ is a Two.

To simplify the definition of $\alpha$, we supply a single-state abstraction function, $\beta$, which converts a single concrete state to a single abstract state. We then define $\alpha$ in terms of $\beta$:

$$\alpha C \equiv \bigsqcup_{c \in C} \beta c$$

This takes each of the supplied concrete states and maps it onto an abstract state using $\beta$, then takes the lub of all of the abstract states to produce a resulting abstract state that covers at least all of the given concrete states. Of course, this process loses information, but that is the whole point of abstraction. In the above example, $\beta \{x \mapsto \text{Nil}\} = \{x \mapsto \{\text{Nil}\}\}$ and $\beta \{x \mapsto \text{Two(\text{Nil}, 12, \text{Nil})}\} = \{x \mapsto \{\text{Two}\}\}$. Hence:

$$\alpha \{\{x \mapsto \text{Nil}\}, \{x \mapsto \text{Two(\text{Nil}, 12, \text{Nil})}\}\} = \{x \mapsto \{\text{Nil}\}\} \sqcup \{x \mapsto \{\text{Two}\}\} = \{x \mapsto \{\text{Nil, Two}\}\}$$
We can define $\beta$ for our abstract domain as follows:

\[
\begin{align*}
\beta &: \text{Env} \rightarrow \text{Env}_a \\
\beta_{\text{Val}} &: \text{Val}_\bot \rightarrow \text{Val}_a \\
\beta(c) &= \bigsqcup_{v \in \text{Var}} \{ v \mapsto \beta_{\text{Val}}(c, v) \} \\
\beta_{\text{Val}}(c, \ldots) &= \{ c \} \\
\beta_{\text{Val}} \bot &= \emptyset \\
\beta_{\text{Val}} \_ &= \text{builtin}
\end{align*}
\]

From this remarkably simple definition (the choice of abstract domain, partial order and $\beta$ function), all of the rest of the abstract interpretation can be derived naturally.

The $\gamma$ function can also be defined in terms of $\beta$:

\[
\gamma a \equiv \{ c : \beta c \sqsubseteq a \}
\]

The above general definition of $\gamma$ specifies that for all states $c$ in the concrete domain, if $\beta c \sqsubseteq a$ (that is, if the abstraction of the concrete state is contained within the given abstract state), then $c$ is part of the resulting set of concrete states.

Specifically, the $\gamma$ function for our abstract domain is as follows:

\[
\begin{align*}
\gamma &: \text{Env}_a \rightarrow \wp(\text{Env}) \\
\gamma_{\text{Val}} &: \text{Val}_a \rightarrow \wp(\text{Val}_\bot) \\
\gamma a &= \text{permute} \bigsqcup_{v \in \text{Var}} \{ v \mapsto \gamma_{\text{Val}}(a, v) \} \\
\gamma_{\text{Val}} cs &= \bigcup \{ \gamma'_{\text{Val}} c : c \in cs \} \cup \{ \bot \} \\
\gamma'_{\text{Val}} \_ \bot &= \text{Q} \cup [\text{Val'}] \cup \text{Func} \\
\gamma'_{\text{Val}} c &= \{ c \} \times \text{Val'}, \quad \text{if } c \in \text{CtorName}
\end{align*}
\]

The function $\text{permute} : \forall a, b. \ (a \rightarrow \wp(b)) \rightarrow \wp(a \rightarrow b)$ takes a mapping $f$ that maps values onto sets, and produces a set of mappings that map each input value onto one of the elements of the output set associated with that value in $f$, for all permutations. For example, $\text{permute} \{ 1 \mapsto \{10,11\}, 2 \mapsto \{20,21\} \} = \{ \{ 1 \mapsto 10, 2 \mapsto 20 \}, \{ 1 \mapsto 10, 2 \mapsto 21 \}, \{ 1 \mapsto 11, 2 \mapsto 20 \}, \{ 1 \mapsto 11, 2 \mapsto 22 \} \}$. This is used to construct all possible permutations of environments in the $\gamma$ function.

The abstraction functions can be chosen arbitrarily to suit the required analysis, but
there is one important rule: they must form a Galois connection, as defined formally in Definition 6.3, with $\alpha$ as the lower adjoint, and $\gamma$ as the upper adjoint. By satisfying this property, the abstract domain is said to make a conservative approximation of the concrete domain. Informally, this means that for any set of concrete states $C$, the abstract value corresponding to $C$ must re-concretise to the same set or a superset of $C$. If abstracting and re-concretising results in a superset of $C$, as it so often does, then we have lost precision, but still made a conservative approximation. If, on the other hand, abstracting and re-concretising results in a subset of, or incomparable value to $C$, the analysis is invalid.

**Definition 6.3.** An pair of monotonic functions over partial orders $(F: A \to B, G: B \to A)$ is said to form a Galois connection \[1\], with $F$ as lower adjoint and $G$ as upper adjoint, if:

$$\forall a \in A, b \in B. \ F a \sqsubseteq b \iff a \sqsubseteq G b$$

We prove that $\alpha$ and $\gamma$ form a Galois connection, with $\alpha$ as lower adjoint, and $\gamma$ as upper adjoint. Substituting our domains into Definition 6.3:

$$\forall C \in \wp(\text{Env}), a \in \text{Env}_a. \ \alpha C \sqsubseteq a \iff C \subseteq \gamma a$$

We must show that:

1. $\forall C \in \wp(\text{Env}), a \in \text{Env}_a. \ \alpha C \sqsubseteq a \implies C \subseteq \gamma a$, and

2. $\forall C \in \wp(\text{Env}), a \in \text{Env}_a. \ C \subseteq \gamma a \implies \alpha C \sqsubseteq a$

For #1, due to the monotonicity of $\gamma$, it is sufficient to show that:

$$\forall C \in \wp(\text{Env}). \ C \subseteq \gamma (\alpha C)$$

If we substitute the definitions of $\alpha$ and $\gamma$ in terms of $\beta$, we obtain:

$$\forall C \in \wp(\text{Env}). \ \forall c \in C. \ \beta c \sqsubseteq \bigcup \{\beta c' : c' \in C\}$$

This is always true, because $\beta c \in \{\beta c' : c' \in C\}$.

For #2, due to the monotonicity of $\alpha$, it is sufficient to show that:

$$\forall a \in \text{Env}_a. \ \alpha (\gamma a) \sqsubseteq a$$

If we substitute the definitions of $\alpha$ and $\gamma$ in terms of $\beta$, we obtain:

$$\forall a \in \text{Env}_a. \ \bigcup \{\beta c : \beta c \sqsubseteq a\} \sqsubseteq a$$
This is always true, and in fact shows that $\alpha (\gamma a) = a$ ($\alpha$ and $\gamma$ form a Galois insertion).

This proof is general; it does not assume our particular abstract domain. For any domain, if $\beta$ is monotonic, and $\alpha$ and $\gamma$ are defined as they are here in terms of $\beta$, they form a Galois connection.

### 6.4 Abstract semantics

To perform the abstract interpretation of a procedure, we must apply the statements and expressions in that procedure to the abstract state. In effect, we are “executing” the program at compile time, but keeping track of only the information we care about, in the abstract domain.

The abstract semantics must be consistent with the concrete semantics. An abstract operation $N_\alpha$ is consistent with its corresponding concrete operation $N$ if, for all abstract values $a$, $N_\alpha a$ gives the same (or weaker) result as finding $\gamma a$ (the set of all corresponding concrete values), applying $N$ to each value, then applying $\alpha$ to the set of results. This is formalized in Definition 6.4.

**Definition 6.4.** An abstract semantics is consistent with a concrete semantics if, for each concrete operation $N$, the corresponding abstract operation $N_\alpha$ satisfies:

$$\forall a. \alpha \{Nc : c \in \gamma a\} \sqsubseteq N_\alpha a$$

An abstract semantics that is consistent with its corresponding concrete semantics is sound.

In fact, in theory, we could mechanically define all of the abstract operations in this manner: $N_\alpha a = \alpha \{Nc : c \in \gamma a\}$; this is called the induced semantics [63]. In this generic form, it is difficult to implement or reason about, so we manually expand it into a definition for each syntactic element, which can be easily translated into actual compiler code. It is not strictly necessary for the abstract semantics to exactly match the induced semantics; in some cases, simpler, weaker semantics (called sub-optimal semantics) are sufficient or even desirable. The abstract operations must only satisfy Definition 6.4 (they must not give results stronger than, or incomparable to, the induced semantics; this would be unsound). They generally should be as precise as possible while satisfying this rule.

If $\alpha$ and $\gamma$ are defined in terms of $\beta$ as above, Definition 6.4 reduces to:

$$\forall a, c. \beta c \sqsubseteq a \implies \beta (Nc) \sqsubseteq N_\alpha a$$  \hspace{1cm} (6.1)
If it can be shown that the following two statements are true:

\[
\forall c. \beta(N c) \sqsubseteq N_{\alpha}(\beta c) \quad (6.2)
\]

\[
\forall x, y. x \sqsubseteq y \implies N_{\alpha} x \sqsubseteq N_{\alpha} y \quad (6.3)
\]

then Equation 6.1 holds, and the abstract semantics is consistent. Therefore, we have only to prove that, for each abstract operation \(N_{\alpha}:

1. for any given concrete state \(c, N_{\alpha}(\beta c)\) is the same or weaker than \(\beta(N c)\), and

2. \(N_{\alpha}\) is monotonic.

The monotonicity of abstract operations is already required to ensure termination, making this a general requirement for abstract interpretation.

We now define the abstract semantics for MIR within the CtorName domain. This semantics has been designed to give results as precise as possible while remaining consistent under Definition 6.4.

The domains for the abstract interpretation \(Env_{\alpha}\) and \(Val_{\alpha}\) are given above. The abstract denotation domain is as follows:

\[
Func_{\alpha} = \text{Val}^\ast_{\alpha} \rightarrow \text{Val}_{\alpha}
\]

\[
\text{Den}_{\alpha} = (\text{CGCName} \cup \text{FuncName} \cup \text{CTName}) \rightarrow (\text{Val}_{\alpha} \cup \text{Func}_{\alpha} \cup \text{Val}^\ast_{\alpha} \rightarrow \text{Func}_{\alpha})
\]

The abstract semantic functions are as follows:

\[
\mathcal{O}_{\alpha} : \text{Den}_{\alpha}
\]

\[
\mathcal{A}_{\alpha} : \text{Atom} \rightarrow \text{Env}_{\alpha} \rightarrow \text{Val}_{\alpha}
\]

\[
\mathcal{E}_{\alpha} : \text{Expr} \rightarrow \text{Den}_{\alpha} \rightarrow \text{Env}_{\alpha} \rightarrow \text{Val}_{\alpha}
\]

\[
\mathcal{S}_{\alpha} : \text{Stmt} \rightarrow \text{Den}_{\alpha} \rightarrow \text{Env}_{\alpha} \rightarrow \text{Env}_{\alpha}
\]

\[
\mathcal{F}_{\alpha} : \text{FuncDef} \rightarrow \text{Den}_{\alpha} \rightarrow \text{Den}_{\alpha}
\]

\[
\mathcal{P}_{\alpha} : \text{Program} \rightarrow \text{Den}_{\alpha}
\]
\begin{align*}
A_a[n]\sigma &= \text{builtin, if } n \in \text{Num} \\
A_a[v]\sigma &= \sigma v, \text{ if } v \in \text{Var} \\
E_a[a]\rho\sigma &= A_a[a]\sigma, \text{ if } a \in \text{Atom} \\
E_a[c]\rho\sigma &= \overline{\rho} c, \text{ if } c \in \text{CGCName} \\
E_a[s \cdot c / n]\rho\sigma &= \top \\
E_a[[a_1, \ldots, a_n]]\rho\sigma &= \text{builtin} \\
E_a[c(a_1, \ldots, a_n) \text{ hint}]\rho\sigma &= \{c\}, \text{ if } c \in \text{CtorName} \\
E_a[v(a_1, \ldots, a_n)]\rho\sigma &= \top, \text{ if } v \in \text{Var} \\
E_a[f(a_1, \ldots, a_n)]\rho\sigma &= (\overline{\rho} f) (A_a[a_1]\sigma, \ldots, A_a[a_n]\sigma), \\
&\text{ if } f \in \text{FuncName} \\
E_a[f(a_1, \ldots, a_n)]\rho\sigma &= \text{builtin} \\
S_a[e]\rho &= \text{id} \\
S_a[s_1 ; s_2]\rho &= (S_a[s_2]\rho) \circ (S_a[s_1]\rho) \\
S_a[v = e]\rho &= \lambda \sigma. \sigma[v \mapsto E_a[e]\rho\sigma] \\
S_a[\text{if } a: \text{then } s_1 \text{ else } s_2]\rho &= \lambda \sigma. S_a[s_1\sigma] \cup S_a[s_2\sigma] \rho\sigma \\
S_a[\text{switch } v: (c_1 : s_1, \ldots, c_n : s_n)]\rho &= \lambda \sigma. \bigcup_{i=0}^{n} \{S_a[s_i]\rho(\sigma[v \mapsto \{c_i\}]), \{c_i\} \subseteq \sigma v \}
\text{ otherwise} \\
S_a[\text{while } a \{u_1 = \psi(v_1, w_1), \ldots\}: s]\rho &= \text{lfp } \lambda h. \lambda \sigma. h (S_a[s]\rho\sigma[u_1 \mapsto v_1 \cup w_1, \ldots]) \\
F_a[f s e]\rho &= \{ f \mapsto E_a[e]\rho (S_a[s]\rho\emptyset) \} \\
F_a[f(a_1, \ldots, a_n)s e]\rho &= \{ f \mapsto \lambda(x_1, \ldots, x_n). \\
&\text{ let } \sigma = \left\{ \begin{array}{l} a_1 \mapsto x_1, \ldots, \\ a_n \mapsto x_n \end{array} \right\} \\
&\text{ in } E_a[e]\rho (S_a[s]\rho\sigma) \} \\
F_a[f\{v_1, \ldots, v_m\}(a_1, \ldots, a_n)s e]\rho &= \emptyset \\
P_a[p] &= \text{lfp } (\bigcup_{f \in p} (F_a f) \cup \lambda \rho. \Theta)\
\end{align*}
We define a handful of primitives in the abstract domain, found in the $\Theta$ mapping:

\[
\begin{align*}
\text{add}(x, y) &= \text{builtin} \\
\text{array}(n, v) &= \text{builtin} \\
\text{array_length}(a) &= \text{builtin} \\
\text{array_ref}(a, i) &= \top \\
\text{array_replace}(a, i, v) &= \text{builtin} \\
\text{error}(\text{msg}) &= \emptyset
\end{align*}
\]

We claim (but do not rigorously prove) that these semantics are consistent with the concrete semantics of Chapter 5. It is self evident that these operations satisfy Equations 6.2 and 6.3. For example, the $A_{\alpha}$ atom semantics:

1. $\forall a \in \text{Atom}. \forall \sigma \in \text{Env.} \ \beta_{\text{Val}}(A[a] \sigma) = A_{\alpha}[a](\beta \sigma).$ ($A_{\alpha}$ is the induced semantics for $A$ for this domain.)

2. $\forall a \in \text{Atom.} \forall \sigma_1, \sigma_2 \in \text{Env}_a. \sigma_1 \subseteq \sigma_2 \implies A_{\alpha}[a] \sigma_1 \subseteq A_{\alpha}[a] \sigma_2.$ ($A_{\alpha}$ is monotonic.)

### 6.5 Making use of the semantics

Now that we have formally defined the abstract semantics, we can make use of it. The key reason we have done all of this is to make it possible to abstractly execute a program without knowing its input, in guaranteed finite time. Together, these properties make it possible to analyse a program at compile time, and derive results that are true in general for all possible inputs to the program (due to the conservative approximation rule).

We can execute a program without knowing its input because for any expressions derived from the program input, we can just assign the result to the most general abstract value possible. We can execute a program in guaranteed finite time because the lattice has a finite height. The most iteration steps required is the height of the lattice.

Consider Example 6.1, a function that violates the rules laid out in Section 5.2.6, by accessing a field when the variable is not statically known to have been constructed with the specified constructor. This code assumes a function $\text{array_ref\_maybe}$, that returns $\text{Just}(x)$ if $x$ is the $i$th element of the given array, or $\text{Nothing}$ if the index is out of bounds. Our analysis could be used to detect this violation: after Line 5, the abstract value for $t$ is $\{\text{Nothing, Just}\}$. Thus, Line 6 is an error, because the set of possible constructors for $t$ is greater than $\{\text{Just}\}$. This shows how static analysis can be used to detect semantic errors.
Example 6.1 Illegal code that accesses a field

```haskell
1: type Maybe(a):
2:   Nothing
3:   Just(a)
4: def get_first(x :: Array(a)):
5:   t = array_ref_maybe(x, 0)
6:   return t.Just/0
   \> Illegal: t is not known to be a Just
```

Consider Example 6.2, a code snippet that switches over a value, only to immediately switch over the same value again. This sort of code appears naturally in MIR whenever the source code performs an explicit switch, then accesses fields by name — the code generator transforms the field access into a second switch, as shown in Example 5.5.

Example 6.2 Code that switches over the same value twice, before and after optimisation

```haskell
1: switch list:
2:   case Nil:
3:     \cdots
4:   case Cons:
5:     switch list:
6:       case Nil:
7:         \> t = “No such field head”
8:         r = error(t)
9:       case Cons:
10:      head = list.Cons/0
11:     \cdots
```

If we assume that the case analysis has, as of Line 1, assigned the variable list the abstract value \( \top \) (we have no idea what it could be), we can proceed to prove that the inner `switch` is not necessary. The abstract semantics for `switch` is the least upper bound of the semantics of all the cases, with the control variable constrained by the case’s constructor (as long as, for control variable \( v \) and case constructor \( c \), \( \{v\} \subseteq c \)). On Lines 2 and 4, this check succeeds, since \( \{\text{Nil}\} \subseteq \top \) and \( \{\text{Cons}\} \subseteq \top \), respectively. When recursing into the `case`, we know on Line 3 that \( \text{list} = \{\text{Nil}\} \), and on Line 5 that \( \text{list} = \{\text{Cons}\} \). Now we encounter the inner `switch` on Line 5, but this time, we notice that \( \{\text{Nil}\} \not\subseteq \{\text{Cons}\} \). This means that the case will never be entered, so rather than recursing, we assign the result of that case the abstract state \( \emptyset \). It recurses on the Cons case as usual.

The final result of this analysis is not particularly interesting, in this case. What is interesting is the potential for optimisations due to intermediate results. For example, an optimiser encountering the inner `switch` on Line 5 can check the abstract value of the control variable, and eliminate any cases that do not feature in that abstract value. If there is only a single constructor in the abstract value (as we have here), it can eliminate
the switch entirely and replace it with the body of the remaining case, as shown in the optimised Example 6.2b. Another possibility is that specialised versions of this function may be produced — for example, if the caller knew statically that list must be a Cons cell, it could call a specialised version which has both switch statements removed, and skips straight to the body of the Cons case.

Now consider the contrived example of a type like List, that has a Cons constructor, but instead of Nil, it features two possible termination symbols, A and B. The function in Example 6.3 finds the final node in the list.

Example 6.3 Function that finds the final node in an A/B list

```
1: def final_node(list):
2:     switch list:
3:         case Cons:
4:             tail = list.Cons/1
5:             r = final_node(tail)
6:         case A:
7:             r = list
8:         case B:
9:             r = list
10:     return r
```

This function is recursive, so the fixed-point iteration is important. Note that the semantics of the program, \( P \), is defined as the lfp (least fixed point) of the semantics of all functions. This is necessary because when a function calls another function (e.g., itself), the analysis could compute a different semantics of that function than was used in the recursive call. Since \( \bigcup_{f \in p}(F_{\alpha}f) \) is monotonic, its least fixed point is well-defined.

The least fixed point semantics is calculated via Kleene iteration, using Algorithm 1.1. The program semantics after each iteration is called \( P_0, P_1, \) and so on. We begin with \( P_0 \) mapping every function to \( \bot \), equivalent to a lambda that returns \( \emptyset \), with the presumption that no function ever terminates. We choose \( \bot \) because it is the strongest possible initial statement, which ensures that when we do find a fixed point, it is the strongest (i.e., least) one. \( P_0 \) is clearly unsound, because it will report that some functions do not terminate when in actual fact they do. Each subsequent pass uses the semantics of the previous pass — \( P_1 \) is computed using \( P_0 \), and so on. This continues until some \( P_n = P_{n-1} \), at which point the least fixed point has been found, and we say that \( P = P_n \).

In this example, we begin with the program semantics:

\[
P_0 = \{ \text{final\_node} \mapsto \lambda \text{list. } \emptyset \}\]

\[^2\text{This approach is very similar to Newton's method for calculating square roots} \[98\], \text{given in Example 3.2.}\]
This leads our analysis to conclude that if `list` is a Cons cell, the function will not terminate, but if `list` is an A or B cell, that cell will be returned. The stated behaviour in the case of Cons is clearly untrue, but this is not a problem because we are not finished yet. The program semantics after this first iteration can be represented as follows:

\[
P_1 = \{\text{final\_node} \mapsto \lambda \text{list}. \text{list} \cap \{A, B\}\}
\]

Because we have a different result than we started with, we iterate again. The variable `tail` is assigned from a field access expression, which always gives the abstract value `T` (to give a more precise answer would require a much richer abstraction that stores recursive information for each value). Therefore, as of Line 4, the abstract environment is \{`list` $\mapsto \{\text{Cons}\}$, `tail` $\mapsto T$\}. We recurse using the semantics we previously computed for `final\_node`, which gives \(r = \{A, B\}\). Therefore, we can conclude that if `list` is a Cons cell, the result will be either A or B, but if `list` is an A or B, that same value will be returned. The program semantics after this second iteration can be represented as follows:

\[
P_2 = \left\{\text{final\_node} \mapsto \lambda \text{list}. \begin{cases} \{A, B\}, & \text{if } \{\text{Cons}\} \subseteq \text{list} \\ \text{list} \cap \{A, B\}, & \text{otherwise} \end{cases}\right\}
\]

Because we again have a different result, we iterate again. We compute the recursive call to `final\_node(\top)`, which gives \(r = \{A, B\}\). This is the same result as last time, so we find that \(P_3 = P_2\). Therefore, \(P_3\) is the least fixed point of the program for the constructor symbol abstract domain, and we say that \(P = P_3\).

It should be noted that this result is more powerful than simply saying “the function might return A or B.” Because the analysis gives an abstract function for each source function, we can derive more specialised results when we have more information. For example, if we know that the argument to `final\_node` is A, the static analysis tells us that the result must also be A.

### 6.6 Condensing semantics and context-insensitive analysis

A key goal for our analysis is to be context-insensitive, meaning we can perform a bottom-up analysis, computing a summary of each function, and analyse function calls just by looking at the callee’s summary. An obvious advantage of such an analysis is that the result of analysing, say, a software library can be stored on disk, to be used by any client linked against the library without re-analysis. Another advantage is feedback for programmers. A context-insensitive analysis can be summarised in a report for the user, showing the general behaviour of each procedure. A context-sensitive analysis can only
produce reports about the behaviour of a procedure when called in a specific way.

Whether the analysis is context insensitive is a matter for the implementation (not
the semantics), but the choice of domain, as well as in defining the semantics (if weaker
than the induced semantics) will determine the complexity of implementing a context-
insensitive analysis. If and only if the abstract semantics is condensing, we can use a
simple trick known as condensation. Marriott and Sondergaard [63] define condensing for
logic programs. Definition 6.5 generalises this concept.

**Definition 6.5.** An abstract semantics is condensing if, given the abstract sequencing operator
\( \bowtie \), for each abstract operation \( \mathcal{N} \), for all statements \( Q \), for all programs \( \rho \), for all pairs of abstract
values \( (a, a') \):

\[
\mathcal{N} Q \rho (a \bowtie a') = a \bowtie \mathcal{N} Q \rho a'
\]

This definition relies on some “abstract sequencing operator”, \( \bowtie \), that represents an
operation that combines two statements that both apply on the same code path. In logic
programs, \( \bowtie \) is the abstract \( \land \) (conjunction) operator. In Mars, \( \bowtie \) is the abstract \( ; \) (statement
sequence) operator. To put this in the context of our analysis, \( \mathcal{N} \) might be the state-
ment semantics \( S_\alpha \), \( Q \) might be a statement \([v = f(x, y)]\), \( \rho \) is the \( \text{Den}_\alpha \) of the program,
while \( a \) and \( a' \) are abstract values in \( \text{Env}_\alpha \). \( \bowtie \) would be a simple map update operation:

\[
A \bowtie B = \lambda x. \begin{cases} B x, & \text{if } B x \text{ is defined} \\ A x, & \text{otherwise} \end{cases}
\]

If a semantics is condensing and \( \text{id} \mapsto \bowtie \) is the right identity element of \( \bowtie \) in the abstract
domain, the following equivalence holds:

\[
\mathcal{N} Q \rho a = \mathcal{N} Q \rho (a \bowtie \text{id} \mapsto) = a \bowtie \mathcal{N} Q \rho \text{id} \mapsto
\]

In our analysis, \( \text{id} \mapsto = \emptyset \). It is now possible to analyse each statement \( Q \) independently
of the abstract state. To perform a bottom-up analysis, analyse each statement of interest
(for example, a call to each function) \( Q \) as follows:

\[
\text{con}_{\rho, Q} \equiv \mathcal{N} Q \rho \text{id} \mapsto
\]

and record the result (\( \text{con}_{\rho, Q} \) is the condensation of \( Q \)). When required to compute \( \mathcal{N} Q \rho a \),
instead compute \( a \bowtie \text{con}_{\rho, Q} \), using the pre-computed condensation of \( Q \).

Unfortunately, our abstract semantics is not condensing, as several operations (the
atom semantics and switch statement) rely on the previous abstract state to compute new
abstract values. For example, the statement $[y = x]$ has the following abstract semantics:

$$S_n [y = x] \rho \sigma = \sigma[[y] \mapsto \sigma [x]]$$

Clearly, the condensing transformation would be unsound:

$$\sigma \bowtie S_n [y = x] \rho \emptyset = \sigma \bowtie \emptyset[[y] \mapsto \emptyset [x]]$$

$$= \sigma[[y] \mapsto \bot]$$

It is sometimes possible to obtain a condensing semantics by weakening a semantics that is not condensing. In our case, we simply need to change the atom and switch semantics, replacing $\sigma v$ with $\top$ in both cases. This means that the analysis will no longer be able to deal with copying values from one variable to another, nor will it ignore cases it knows are unreachable. However, it is now condensing (a simple proof sketch: the only remaining use of $\sigma$ is the assignment statement, which returns $\sigma[v \mapsto \ldots]$; observe that $(a \bowtie a')[v \mapsto \ldots] = a \bowtie a'[v \mapsto \ldots]$). The implementation is now free to pre-compute the result of any statement (including any function call).

With our weakened semantics, Example 6.3 comes to:

$$\{\text{final_node} \mapsto \{A, B\}\}$$

The analysis can still tell you that the result will be either A or B, but it lacks the nuance that tells us that $\{A\}$ maps to $\{A\}$ and $\{B\}$ maps to $\{B\}$.

If the precision loss of making the semantics condensing is unacceptable, we can still implement a context-insensitive analysis for a non-condensing semantics (at a cost of significant complexity). To do so, the data structure we use to represent abstract values must support algebraic values (terms for the unknown abstract values of function parameters). The algebraic data structure must be expressive enough to handle all of the abstract operations, such as conditions and $\cup$. For Example 6.3, we need to represent the following term inside the analyser (where list is the unknown abstract value of the parameter):

$$\left( \begin{array}{l} \text{if } \{\text{Cons}\} \subseteq \text{list} \\ \{A, B\}, \text{ otherwise} \end{array} \right) \cup \left( \begin{array}{l} \text{if } \{\text{A}\} \subseteq \text{list} \\ \{A\}, \text{ otherwise} \end{array} \right) \cup \left( \begin{array}{l} \text{if } \{\text{B}\} \subseteq \text{list} \\ \{B\}, \text{ otherwise} \end{array} \right)$$

This algebraic abstract value would represent the result of analysing the function, and stand as a summary for the function. When analysing calls to final_node, the analyser need only substitute list for the known abstract value, and does not need to re-analyse the function.
6.7 Efficient bottom-up analysis

We may colloquially call the context-sensitive analysis “top-down” due to its starting at main and analysing down the call graph of the program. Similarly, we may call the context-insensitive analysis “bottom-up” due to its ability to analyse the leaves of the call graph first before considering the callers of those leaves, and so on. However, the stated mechanism for analysing the program — analyse the entire program once and then repeat the analysis until a global fixed point is achieved — is highly inefficient.

We can make the following two simple observations that will be true for any context-insensitive analysis:

1. The analysis of a procedure depends only upon the analysis results of its direct or indirect callees. Therefore, it is possible to obtain the correct results for a given procedure $p$ without analysing any procedure which is not a successor of $p$ in the call graph.

2. Any non-recursive procedure which calls only procedures with a known correct analysis will produce the correct result in a single iteration.

Note that for the purposes of this analysis, we consider a self-recursive procedure to have a directed edge to itself. These observations lead to a simple strategy for analysing any program whose call graph is a directed acyclic graph: find a reverse topological sort of the graph, and analyse each procedure in that order. In this case, there is no need for a fixed point analysis at all — each procedure may be analysed correctly in a single iteration. This will begin by analysing a leaf node (a non-recursive procedure which calls no other procedure), and ensure that each subsequent procedure analysed is either a leaf node, or one in which all of the callees already have a known correct analysis.

This becomes more complicated when self or mutual recursion is involved; in this case, the program is not a directed acyclic graph and has no topological sort. Fixed-point iteration is required to resolve recursive dependencies, as the callees of the recursive procedure do not have known correct analyses. We can still retain most of the efficiency of the topological sort approach by identifying the strongly connected components of the call graph.

A strongly connected component (SCC) of a directed graph is a subgraph in which there is a path from every node to every other node. These subgraphs are disjoint (there is no node that appears in more than one SCC), and the edges between all of the SCCs in a graph form a directed acyclic graph. The SCCs of a call graph are the groups of mutually recursive procedures. Any procedure not involved in mutual recursion is in a singleton
SCC by itself. Several well-known algorithms exist for identifying the SCCs of a graph; the Mars compiler uses Tarjan’s algorithm [88], given in Algorithm 6.1.

Algorithm 6.1 Tarjan’s algorithm for computing strongly connected components

procedure strong_connect(v):
    \( \text{lowlink}_v = \text{number}_v = i = i + 1 \)
    put \( v \) on stack of points
    for \( w \) in the adjacency list of \( v \):
        if \( w \) is not yet numbered:
            \( \text{strong_connect}(w) \)
            \( \text{lowlink}_v = \min(\text{lowlink}_v, \text{lowlink}_w) \)
        else if \( \text{number}_w < \text{number}_v \):
            \( \text{lowlink}_v = \min(\text{lowlink}_v, \text{number}_w) \)
            if \( w \) is on stack of points:
                \( \text{lowlink}_v = \min(\text{lowlink}_v, \text{number}_w) \)
                if \( \text{lowlink}_v = \text{number}_v \):
                    \( \triangleright v \) is the root of a component
                    start new strongly connected component
                    while \( w \) on top of point stack satisfies \( \text{number}_w \geq \text{number}_v \):
                        delete \( w \) from point stack and put \( w \) in current component
                        \( i = 0 \)
                        empty stack of points
                \( \text{strong_connect}(w) \)

The algorithm is transcribed directly from Tarjan’s paper (with notational differences). The input to the algorithm is a directed graph, consisting of a set of vertices, each with an adjacency list (each vertex \( v \) contains a set of vertices \( w \) where there is an edge from \( v \) to \( w \)). The output of the algorithm is a sequence of SCCs (the algorithm does not explicitly produce output; it describes the creation and update of strongly connected components, and these should be considered the output).

Tarjan’s algorithm operates in \( O(v + e) \) time (for \( v \) vertices and \( e \) edges) [88], and produces a sequence of SCCs for a given directed graph with a useful property: for each SCC \( c \), the algorithm will yield all of the successors of \( c \) before \( c \) itself. Hence, not only does this algorithm compute all of the SCCs, but also a reverse topological sort of the directed acyclic graph formed by the edges between the SCCs. We take advantage of this and form a simple strategy, given in Algorithm 6.2, which reduces the amount of computation required to perform a bottom-up analysis of a program. A more sophisticated approach is outside the scope of this thesis.

For a program without any recursion, this will be the same as our previous simple approach of analysing each procedure once in reverse topological order. Self-recursive procedures will be analysed until a fixed point is found, but they will be analysed in-
Algorithm 6.2 Strategy for efficiently performing a bottom-up fixed point analysis

\[ \text{sccs} = \text{a reverse topological sort of the SCCs of the program’s call graph (Tarjan’s algorithm)} \]

\text{for} each SCC \( c \) in \( \text{sccs} \):

\text{if} \( c \) has a single procedure \( p \) and \( p \) is non-recursive:

analyse the procedure \( p \) and update the program state

\text{else:}

\text{repeat:}

\text{for} each procedure \( p \) in \( c \):

analyse the procedure \( p \) and update the program state

\text{until} the program state is unchanged from the previous iteration

---

Figure 6.3: Call graph with non-singleton SCCs marked in grey

individually, so a procedure requiring a large number of iterations will not cause the re-analysis of any other procedure. Only groups of mutually recursive procedures will be analysed together until a fixed point is reached.

For example, consider the program whose call graph is depicted in Figure 6.3. Tarjan’s algorithm yields the following sequence of SCCs, in reverse-topological order:

\[ \{h\}, \{g\}, \{e,f\}, \{d\}, \{a,b,c\} \]

Hence, the analysis takes the following route:

1. Analyse \( h \) one time. (As \( h \) is non-recursive, the analysis is guaranteed to produce the correct results in one attempt.)

2. Analyse \( g \) repeatedly until a fixed point is found.

3. Analyse \( e \) and \( f \), in an arbitrary order. Repeat the analysis of both procedures until a fixed point is found.

4. Analyse \( d \) one time.

5. Analyse \( a, b \) and \( c \) repeatedly until a fixed point is found.
6.8 Conclusion

In this chapter, we introduced the concept of abstract interpretation, defining key terms and fully working through a simple example for our language, Mars. Rather than beginning by tackling the aliasing analysis (which is the subject of the remainder of this thesis), our example is a first-order analysis that tracks compile-time knowledge about which constructors may have been used to build any given variable.

In Chapter 5, we defined two concrete domains for Mars: the value domain and the memory domain. We begin an abstract interpretation by choosing a concrete domain that provides the relevant properties (for example, for our constructor analysis, we need only know the value of each variable, but for our aliasing analysis, we need to know which variables share memory locations).

We then define an abstract domain, which is a simplified representation of the program state that we are interested in. For example, in our constructor analysis, the abstract domain records which possible sets of constructors were used to build each variable, but nothing else about a variable’s value. Each state in the abstract domain represents one or more concrete states, as defined by the \( \gamma \) mapping function.

We finally define the abstract semantics, which are the analogue to the concrete semantics, but operate in the abstract domain. To be sound, every operation in the abstract semantics, for any abstract input \( a \), must give the same (or weaker) result as the corresponding concrete operation applied to the concrete states represented by \( a \). By recording the abstract state at any given program point, we can learn specific facts for the purpose of optimisation, compile-time errors, and so on.

The remainder of the chapter is devoted to performing a context-insensitive analysis precisely and efficiently. We wish to analyse each function independently of its calling context (otherwise, we will have to re-analyse each function every time it is called with different abstract inputs). If the semantics is condensing, we can quite easily produce a summary of each function by assuming the initial abstract state is the identity element of the abstract sequencing operator. If not, we must produce an algebraic summary of the function, and substitute known values at each call site. We also described an algorithm for deciding on the order in which to traverse the functions in a program, to minimise the number of iterations required for the global analysis to reach a fixed point.

In the next chapter, we shift the focus to aliasing analyses and compile-time garbage collection. We examine the history of theoretical and real-world examples of compilers that have in some way pushed the envelope for detecting when allocated memory is no longer live, and finding opportunities to reuse existing dead memory.
Chapter 7

Existing sharing and reuse analyses

7.1 Introduction

In Chapter 3, we presented a case for programming languages with interface integrity (presented formally in Definition 3.5), which essentially means languages in which execution units have no side-effects. This theory was used to build a pure imperative language, Mars, outlined in Chapter 4. A major caveat has so far been applied to this design: the imperative programming style, which Mars encourages, makes heavy use of incremental modifications to data structures; in particular, arrays. However, the programmer, in a language with interface integrity, cannot explicitly request modifications to data structures. Our language design, it would appear, hinges on the ability of the compiler to optimise a program with interface integrity such that it makes efficient destructive updates where possible without changing the program’s semantics. We shall devote the remainder of this thesis to addressing this problem.

Purely declarative programming languages, such as Haskell [49], Clean [78] and Mercury [38], are distinguished by the rule that all values, once created, are immutable. Any supposed “modification” of a value is performed by creating a partly modified copy of the original, thus preserving the original unmodified value. The rationale for this is that any other pointers to the value’s memory space will not be affected by the update, so no accidental aliasing errors can be made. For example, in other programming languages, it is possible to pass an array as input to a function without realising that that function modifies the array. This category of errors does not arise in a pure language.

However, having to copy all or part of a data structure every time it is updated is a major source of inefficiency when compared with impure languages, which let programmers explicitly modify data. If a single pointer is pointing at a place in memory and the program requests the data be copied (possibly with only minor modifications), and the
original pointer will never be used again, the compiler could transform this copy operation into a much cheaper update of the original data. However, this would be disastrous if multiple pointers were pointing at that data, or if the original pointer was used again, because the optimised version would be changing live values that the original program did not intend to be modified.

Therefore, determining at compile time when a value is definitely not aliased (or not shared) is of key interest to the purely declarative languages community. This analysis is known as aliasing analysis or sharing analysis, and the use of the results to perform destructive update or free memory is called compile-time garbage collection.

This chapter describes the relevant literature that presents new analyses, provides experimental feedback, or describes these concepts in more detail. First, memory reuse approaches based on type systems are discussed. Then, a number of related sharing analyses and compile-time garbage collection systems are discussed.

7.2 Runtime reference counting

Among the simplest approaches to structure reuse is to simply track all pointer aliasing at runtime, by keeping a reference count on each object — an integer counting the number of pointers to that object. This technique is typically used as an automatic memory management system, to reclaim objects that are no longer reachable in languages such as Python. However, unlike runtime garbage collection, reference counting can also be used to perform automatic destructive update: whenever a variable is to be copied and modified, if the variable is about to go out of scope and its reference count is exactly 1, update the existing object instead of copying it. This technique is used in Sisal [24] and its descendant, SAC [30].

The downside of reference counting is that it consumes memory and time at runtime — whenever a pointer is created or destroyed, its target must be both read and written. These language implementations provide compile-time optimisations to reduce the number of reference increments and decrements [24].

7.3 Type-based approaches

The goal of this thesis is to allow destructive updates to arrays and other structures in a language with interface integrity. While many such languages already feature destructive update operations, they have various limitations and often lack the benefits of programming with interface integrity in the first place. In this section, we explore various
existing approaches to destructive update in pure programming languages. A common theme among these approaches is some use of the type system to force the programmer’s destructive updates to maintain interface integrity.

### 7.3.1 The store

A common approach towards destructive update is to offer an opaque “store” type with corresponding opaque “reference” types. The programmer may create new structures or arrays in the store (obtaining references to these objects), and may subsequently read or write to fields of those objects via references. The language must provide some guarantee that a) the store objects will not be duplicated, and b) no reference may be used with any store other than the one that created it (which can be accomplished with parametric types). If these two guarantees can be enforced, then the object updates may be implemented as direct destructive memory updates, yet the store operations may be considered to have interface integrity. This is because if there is only ever one reference to a store, mutating the object can be conceptualised as destroying the old store and creating a new slightly different one.

In Haskell, monads can be used to prevent store duplication. The ST (state threading) monad is Haskell’s basic store type. In Mercury, store duplication is prevented using uniqueness typing, which is introduced in the next section.

In Section 2.4, the example of building a hash table from an STArray inside Haskell’s ST monad was discussed in detail (see Example 2.9). While this example shows that it is possible to achieve the desired performance using a store, it forces the programmer into an imperative semantics where aliases may inadvertently be destructively updated. The store is ideal when aliased update semantics is actually desirable, but if we only need destructive update for performance, the store does not meet our goal of being able to perform updates without compromising the purely declarative semantics.

### 7.3.2 Linear logic and uniqueness typing

A mechanism for determining whether a variable is eligible for destructive update is linear types, introduced by Wadler [93], who applied the concept of linear logic [26] to a Hindley-Milner [40, 71] type system. Under this type system, each variable can have a nonlinear type or a linear type. A linearly typed value can have only a single usage: it can neither be duplicated nor discarded, unless a “discard” primitive is supplied. For example, a linear array may be destroyed by passing it to the primitive dealloc; linear variables cannot simply be “forgotten about.” Wadler’s original proposal made nonlinear
and linear types completely distinct — it was not possible to convert a normal type to a linear type and vice versa. A later paper by Wadler [94] added a feature called dereliction, which allows a variable of nonlinear type to be coerced to a linear type.

Linear types, for example, can be used to represent the “world” value, in the state-of-the-world threading concept discussed in Section 3.3.6. Whenever an I/O action is performed, the old world is conceptually destroyed and a new world is conceptually created; thus there is always exactly one world object at any given time. Similarly, they can be used to model the store discussed above, guaranteeing that it is never duplicated. Dereliction is not a problem for these scenarios, because it is not possible to create a nonlinear I/O or store type at all. However, we now turn our attention to modelling individual data objects such as arrays, to guarantee that they can always be destructively updated. Assuming that our language provides a mechanism for creating both nonlinear and linear arrays, we can see that the dereliction rule can be used to coerce a nonlinear array to a linear one, breaking our assumption that a linear value can be safely destructively updated, as Wadler admits [94]:

*Dereliction means we cannot guarantee a priori that a variable of linear type has exactly one pointer to it. But if we know this by other means, then linearity guarantees that the pointer will not be duplicated or discarded.*

Wadler hints at other analyses for determining whether a variable is shared, which we will come to later. Nevertheless, a very similar system called uniqueness typing [7, 20] solves the problem of dereliction. In this scheme, each variable can have a non-unique type or a *unique type*. Unlike linear-type variables, unique-type variables can be duplicated (but then they lose their uniqueness property). Unique-type variables can also be discarded without an explicit “discard” operation. Importantly, though, it is not possible to coerce a non-unique variable into a unique one. Thus, uniqueness typing more precisely captures the attributes we are interested in for destructive update: we do not care if it becomes duplicated (as long as no subsequent destructive update occurs), we do not care if it is forgotten about (the runtime garbage collector will free it), but we do need to make sure that there is exactly one reference to the variable’s memory cell at the time of the update.

It would seem that allowing the programmer to discard a unique value would make uniqueness typing unsuitable for state-of-the-world I/O, but this is not the case. In order to ensure that programs do not discard the world value, it is merely sufficient for there to be a) no way to fabricate a world value during the program’s execution, other than by modifying an existing unique world, and b) a requirement that the program entry-point (i.e., the main function) return a world value. In that case, any function which discards
the world will not be usable in a valid program, as that program would have no way to return a unique world at the end of main.\footnote{This system is not without its problems. For example, while Clean \cite{78} ensures that the World is not destroyed, it makes no such guarantee about File objects. Programmers must remember to fclose a file or any writes may or may not be written to disk \cite{3}. Under a linear typing discipline, forgetting to close the file would be an error.}

Clean \cite{78} is a lazy purely functional programming language, very similar to Haskell. It has featured uniqueness types since at least 1993 \cite{7} and remains the choice language for research in this area. Mercury \cite{38} features a similar system based on uniqueness typing.\footnote{Although Mercury performs uniqueness checking in the mode system, not the type system.} Both languages use unique values for state-of-the-world I/O; Clean also uses uniqueness for destructive update of arrays, which is of key concern to this thesis.

In Clean, the type $\ast a$ (read as “unique $a$”) denotes a value of type $a$ that references an object with a reference count of 1 — an unshared value. Arrays in Clean have special syntax: the type $\{a\}$ is an array of $a$ values; the expression $\{a \& [i] = v\}$ is the array replacement operator, producing a new array like $a$ but with element $i$ mapped to the value of $v$. Unlike Mars, the array replacement operator requires that $a$ be unique, and therefore always performs a destructive update. As an example, the type $\ast \{\text{Int}\}$ denotes a unique array of integers. This allows a function like setFirst in Example 7.1 to operate in constant time.

\begin{example}{Example 7.1}{The setFirst function in Clean}

\begin{verbatim}
setFirst :: $\ast \{a\} \ a \rightarrow \ast \{a\}$
setFirst \(xs\ \ y\) = \(\{xs \& [0] = y\}\)
\end{verbatim}

\end{example}

The setFirst function cannot be called unless the caller can guarantee that the array argument is unique — during the execution of setFirst, the variable $xs$ is the only live reference to the array’s memory cell. In return, this function destructively modifies the array, and promises that the resulting array is unique. Roughly speaking, a variable in a given function is unique if a) the variable’s type is a unique type, and b) the variable is referred to at most once in the right-hand side of the function. If the function has multiple branches (such as guards or a case statement), then it is permitted that the variable be referred to (at most once) in each branch. Hence, the function setLast in Example 7.2 is illegal, because it refers to $xs$ multiple times in the function’s right-hand side — it is a compiler error to attempt to update the array.

\begin{example}{Example 7.2}{The setLast function, which will not compile in Clean}

\begin{verbatim}
setLast :: $\ast \{a\} \ a \rightarrow \ast \{a\}$
setLast \(xs\ \ y\) = \(\{xs \& [(\text{size} \ xs) - 1] = y\}\)
\end{verbatim}

\end{example}
Note that the laziness of Clean does add complexity here: in general it would be incorrect to assume that \( \text{size } xs \) has been evaluated before calling a function, but even in this case (where the index for array replacement is strict), it is not considered unique. To work around this issue, Clean provides an additional rule: if a reference to a variable \( x \) appears in a strict let-before expression (a strict assignment statement, denoted with a \( \#! \)), then Clean performs a simple sharing analysis and if the assigned variable does not alias \( x \), then it does not prevent subsequent references to \( x \) from being considered unique. Therefore, Example 7.3 is a legal Clean implementation of setLast — it uses a let-before to strictly assign the result of \( (\text{size } xs) - 1 \) to the variable \( i \), which does not alias any part of \( xs \), and therefore, \( xs \) is still considered unique at the moment of the array update. A similar rule covers usage of variables on the left-hand side of guard expressions.

Example 7.3 Corrected version of setLast in Clean

```clean
setLast :: ∗{a} a → ∗{a}
setLast xs y
  #! i = (size xs) - 1
  = {xs & [i] = y}
```

However, this only covers special cases involving mentioning the same variable multiple times. Clean has no mechanism for dealing with a case where a variable becomes shared, then becomes unshared again, then needs to be destructively updated. Consider Example 7.4, a contrived implementation of \( \text{id} \) which passes the variable \( x \) to both arguments of \( \text{const} \). (The type of \( \text{const} \) indicates that the first argument may either be unique or non-unique, and the result will match.) The variable is considered non-unique in both references, and so it is passed to \( \text{const} \) as a non-unique type. Hence, the resulting type is non-unique, despite the fact that once \( \text{id} \) returns, \( x \) is just as unique as it was when it was passed in. This causes \( \text{id} \) to be given the type \( a \to a \) — note the lack of uniqueness. If a unique value was passed to this implementation of \( \text{id} \), the result would be considered non-unique from that point on.

Example 7.4 The \( \text{id} \) function defined in terms of \( \text{const} \), in Clean

```clean
id :: a → a
id x = \text{const } x x
\text{const} :: u:a v:b → u:a
\text{const } x y = x
```

Note that Clean does not require explicit uniqueness annotations. As well as being able to automatically infer types, the compiler can also infer uniqueness annotations.\(^3\)

\(^3\)However, it does not appear to be possible to explicitly specify a type but have the compiler infer its uniqueness. The special attribute “\( . \)” can infer the appropriate uniqueness variables such as \( u \) and \( v \), but cannot infer whether a type must be unique or must be non-unique.
7.3. TYPE-BASED APPROACHES

With inferred annotations, Clean’s uniqueness system operates as a simple conservative sharing analysis that statically ensures values are unshared when certain operations are applied.

De Vries et al. [20, 21] describe a simplification to Clean’s uniqueness system. The proposed system unifies the uniqueness annotations with the type system, and removes the complexity of subtyping in Clean’s existing uniqueness system, without compromising expressiveness. This simplification has two practical consequences. First, it allows the uniqueness system to be implemented easily in existing compilers with a Hindley-Milner type system and kind system. Second, it can be more easily understood by programmers, which aids in understanding compiler errors. However, it does not change the underlying problem of Clean’s uniqueness system outlined above.

Aldrich et al. [5] present an extension to Java called AliasJava that adds uniqueness annotations to the language. The language is extended with subtyping, allowing the user to annotate the types of variables, fields, arguments and return values with annotations such as \texttt{unique}, to indicate that there may not be any other reference to that object. AliasJava code compiles to standard Java Virtual Machine bytecode via an extension to the ArchJava compiler.

AliasJava programs can therefore contain explicit guarantees regarding aliasing. As with Clean, it can also automatically infer annotations. Where Clean uses uniqueness typing to allow semantically pure destructive update, AliasJava allows Java’s normal aliased destructive update; the uniqueness annotations are instead used to describe aliasing and prevent accidental aliased update errors.

In C++, an object can be passed by reference using the \& type qualifier, but such references often need to be deep-copied in order to avoid aliased destructive update, and also to guarantee unique ownership. C++ recently added a new type qualifier, \&\& (an rvalue reference [45]), which is essentially a uniqueness annotation. Like Clean’s *, an rvalue reference parameter means the caller will no longer use the argument’s value, so it is safe to move it into a new object without copying, invalidating the old object. Note that aside from temporary expressions, C++ programmers are required to manually specify when a value is unique (there is no uniqueness inference algorithm).

Because they use type annotations to track aliasing, both the linear types and the uniqueness typing approaches are fundamentally unable to consider variables unique if they become shared and then become unshared. For this reason, we do not consider the type-based approach further. However, we do note that the type-based approach, with the expressive power of the Hindley-Milner type system, does intrinsically provide sup-

\footnote{For example, Haskell.}
port for higher-order functions, which many of the more powerful analyses are lacking.

We are interested in a solution that does not require the programmer to explicitly choose when to destructively modify a data structure — to fully allow aliasing between objects, but preserve the value semantics of pure programming, and opportunistically perform destructive update when it is semantically safe to do so. Neither the store nor uniqueness typing approaches provide these features, and so we turn to a more sophisticated analysis known as compile-time garbage collection.

7.4 Compile-time garbage collection

Garbage collection dates back to 1958 [68], when it was introduced by McCarthy for use in Lisp. The concept is familiar to users of modern programming languages: a runtime system automatically discovers memory locations that have been allocated but are no longer reachable by any variables in the program, and frees those locations, making them available for subsequent allocations. Compile-time garbage collection (CTGC) is the automatic discovery of unreachable memory locations, applied statically to a program at compile-time. There are two distinct applications of CTGC analysis: the compiler may insert “free” instructions into a program, where it is known that a particular memory cell is never referenced after a particular program point, or the compiler may find opportunities to reuse memory directly, by finding dead memory cells to reuse instead of allocating new ones. Our research focuses on the latter task.

The term “compile-time garbage collection” is due to Darlington and Burstall [18], a very early (1976) analysis for transforming a memory allocation into a destructive update:

\[\text{The optimisation attempted is to avoid store usage by re-using any list cells that will have been discarded. This process can be thought of as a compile time garbage collection.}\]

Their analysis identifies the static memory cells associated with each program variable. It appears to work only for the top level of a program, not the body of a procedure, as it assumes input variables are non-aliased. If it finds that a particular memory cell is no longer referenced by any variable, it attempts to modify an allocation instruction to instead assign to that memory cell. They demonstrate the analysis of a list reverse program, given in Example 7.5, showing their ability to identify that at the beginning of Line 3, the memory cell named by \(x\) is never used again (despite its head and tail, \(\text{hd}(x)\))

\[\text{Although it was not called “garbage collection” in [67], it was known by that name internally; see the anecdote at the end of [68].}\]
and \(\text{tl}(x)\), respectively, being subsequently used), and therefore are able to replace the call to Cons with a reuse of the existing cell named by \(x\).

**Example 7.5** List reverse program, before and after optimisation (from [18])

1: \(\text{result} := \text{nil}\)
2: while not null\((x)\):
3: \(\text{result} := \text{cons}(\text{hd}(x), \text{result})\)
4: \(x := \text{tl}(x)\)

1: \(\text{result} := \text{nil}\)
2: while not null\((x)\):
3: \(\text{newvar}_1 := \text{tl}(x)\)
4: \(\text{tl}(x) := \text{result}\)
5: \(\text{result} := x\)
6: \(x := \text{newvar}_1\)

The analysis explicitly tracks which cells are pointed to by a given variable. However, it only has cells for the parts of a data structure that are immediately visible in the source code (in this example, there is a cell for \(\text{result}\), \(x\) and \(\text{tl}(x)\)). Any deeper parts of data structures are not explicitly modelled, and the treatment of the while loop is glossed over — the analysis seems to assume that the abstract state at the end of the loop body is the same as at the beginning (as it does not perform any kind of fixed-point iteration).

Hudak [41] provides a far more detailed and formal model for compile-time garbage collection. This work provides the standard denotational semantics for a simple programming language, as well as a second denotational semantics that explicitly models reference counts of objects. Finally, an abstract interpretation of the reference count semantics is presented, which allows the approximate reference counts to be computed at compile time. This three-step approach is common in compile-time garbage collection, and we also take a similar approach in this thesis. Hudak’s abstract reference count domain is defined as:

\[
\text{Src} = \{0, 1, \ldots, \text{maxrc}, \infty\}
\]

This means that the analysis precisely tracks the number of references to each object up to some arbitrary maximum \((\text{maxrc})\), after which it loses count — more precise than other analyses that simply consider whether an object is shared or not. However, this model only counts references from variables, not fields or array elements.

Hederman [37] implemented a compile-time garbage collection scheme based on Hudak’s model [41] for the functional language Russell. Like Hudak, this used an abstraction of reference counting to track aliases of variables, and gives up after an object is referred to by more than some arbitrary maximum number of variables, or by any other object in the heap. A contribution of this work was a context-insensitive interprocedural analysis — by analysing the procedures bottom-up, a library may be analysed without regard for how it will be used. The result of analysing each function comprises two pieces of information:
• result: whether the result is newly allocated, a pointer to static or stack data, a copy of an input parameter, some combination of the above, or completely unknown (top), and

• for each parameter, stored (whether the parameter is stored in memory and hence cannot be deallocated) and returned (whether the parameter is returned and hence aliased to the return value).

A caller can use this information to decide how to update the reference counts for its own variables.

Jones and Le Métayer [47] present an analysis which uses aliasing information to determine points in a program at which the compiler can insert free statements, to save time collecting garbage at run-time. Such an analysis is necessarily conservative, as it is not possible to compute all states of a program at compile time.

Like Hudak, Jones and Le Métayer’s analysis is an abstract interpretation of reference counting. While this analysis does not distinguish objects with a reference count of 1 or more (objects are considered either “shared” or “unshared”), it improves over Hudak in a significant respect: it individually tracks the sharing of the fields of a structure, and so is able to handle recursive data structures such as lists.

Their analysis operates on a language which, like the original Lisp, has only two types of value: a “cons cell” (a heap-allocated pair, usually representing a linked list with the head in the first field and tail in the second field) and an atomic value, such as a number or the special value nil. They introduce a domain \( P \) (patterns), which is defined recursively as:

\[
P = \{0, 1\} \cup (P \times P)
\]

A pattern is either 0, 1, or a pair of patterns, such as \((0, 1)\) or \(((1, 0), ((1, 1), 1))\). Patterns are used for several different interpretations, but for the sharing analysis, they are used as abstract values to represent sharing information for cons cells. In this interpretation, a variable \( x \) having abstract value 0 indicates that no part of \( x \) is shared; a 1 indicates that all parts of \( x \) may be shared; a pair \((a, b)\) indicates that the top cell of \( x \) is not shared, but that the first and second fields of the cell may be shared according to patterns \( a \) and \( b \), respectively. For example, the pattern \((1, 1)\) indicates that the variable is not shared, but its head and tail may be. The pattern \((0, 0)\) never appears in the sharing interpretation, as it would have the same meaning as 0.

In order to make the \( P \) domain finite, so that analysis may be performed at compile-time, they introduce domains \( P_n \) for all \( n \in \mathbb{N} \), where each domain \( P_n \) is like \( P \) but with a maximum height of \( n \). This technique is called \( k\)-limiting by Jones and Muchnick [46],
and is a common theme among aliasing analyses. Formally, they are defined as:

\[
P_0 = \{0, 1\}
\]

\[
P_{n+1} = P_n \cup \{P_n \times P_n\}
\]

The analysis writer may choose any sufficiently high value for \(n\); higher values allow for a more precise but possibly slower analysis, but Jones and Le Métayer state that “In practice \(P_1\) is most often enough.” Under \(P_1\), the abstract value domain in the sharing interpretation would consist of the values in the set \(\{0, 1, (0, 1), (1, 0), (1, 1)\}\), which for linked lists allows separate tracking of the first cons cell in the list, the head of the list and its components, and the spine of the list together with all elements of the list besides the first. This is fairly limiting in recursive cases, because it is this third category which blurs the sharing information about nearly all parts of the list into a single Boolean value.

In the paper, the example of a naïve reverse is given, which works by calling append. Both reverse and append are analysed and it is concluded that if reverse is passed a list which is totally unshared (with the pattern 0), not only will the result also be unshared, but the compiler will be able to transform all memory allocations into reuses of existing memory cells. We note that there is very little tolerance for partially shared data here — if we had a list in which the entire spine was unshared, but some elements of which may be shared, we would have the input pattern \((1, 1)\), and therefore be unable to detect any reuse opportunities after the first list cell.

A further limitation of this analysis is that it makes no attempt to track which variables share with which other variables, only that a variable may be shared. Therefore, while it can identify variables that have never been aliased and reuse them, it is unable to determine when a shared variable becomes unshared. Once a variable is shared, it is no longer a candidate for compile-time garbage collection. This is approximately the same limitation as Clean’s uniqueness analysis, discussed above. We shall see why, using Example 7.6, the same example we used with Clean. Consider id applied to a completely unshared value (an abstract value 0).

**Example 7.6** The id function defined in terms of const

\[
id(x) = \text{const}(x, x)
\]

\[
\text{const}(x, y) = x
\]

We can determine the sharing properties of id by applying Jones and Le Métayer’s sharing interpretation, \(S\). There are two separate analyses of each function, which are joined to produce the final result. \(S_t\) denotes the “transmission,” giving the sharing that
arises due to existing sharing in the inputs:

\[
S_t \text{ const } (x, y) = x \\
S_t \text{ id } (x) = x
\]

This means that the result of both const and id has the same sharing as the \( x \) argument. \( S_c \) denotes the "creation," giving the sharing that is created as a result of the function's behaviour, independent of the existing sharing in the function parameters:

\[
S_c \text{ const } = 0 \\
S_c \text{ id } = 1
\]

This means that while const does not create any new sharing, id does. Because it passes the variable \( x \) as both arguments to const, both parameters are considered shared, and therefore the resulting sharing is 1, despite the fact that at the end of id, \( x \) is no more shared than it was at the start. The result for each function is a join of the transmission and creation, showing that no matter what you put into id, the result is always shared:

\[
S \text{ const } (x, y) = x \sqcup 0 = x \\
S \text{ id } (x) = x \sqcup 1 = 1
\]

In order to detect that a previously aliased variable has become unique and hence is eligible for destruction, an analysis must go beyond a simple reference count abstraction and explicitly record the aliasing between each pair of variables. For example, if on one branch, \( y \) may alias \( x \), and on another, \( z \) may alias \( x \), it is insufficient to record reference counts for each variable:

\[
\{ x \mapsto 1, y \mapsto 1, z \mapsto 1 \}
\]

Rather, an analysis must record an alias pair for each pair of variables that may be aliased:

\[
\{ \{ x, y \}, \{ x, z \} \}
\]

Bruynooghe [11] describes such an analysis for a Prolog-like language. The global analysis tracks aliasing between the local variables of a procedure as a set of aliasing pairs, and propagates aliasing across procedure calls. The analysis is context sensitive: each procedure is analysed separately for each call pattern, and each call pattern may produce

\[^6\text{In [11], aliases between } x \text{ and } y \text{ are described with the notation } x \text{ AL } y, \text{ whereas we use the notation } \{ x, y \} \text{ for consistency with the rest of this thesis.}\]
a different version of the procedure with different update behaviour. A liveness analysis
determines when a variable will no longer be used and removes it from the alias sets,
so that previously aliased variables can be considered unique. In the above example,
if the liveness analysis determined that \textit{x} would no longer be used, \textit{y} and \textit{z} would be
considered unique.

A key contribution of this work is a more precise solution to the recursive types prob-
lem. Any recursive type could cause the analysis not to terminate, as it computes an
ever-expanding set of aliases. While other analysis, such as Jones and Le Métayer [47],
use \textit{k}-limiting to cut off the tree at some arbitrary level, Bruynooghe introduces a type-
based approach that better preserves the interesting properties of the program. Recall
that Jones and Le Métayer’s analysis folded up sharing information about all of the ele-
ments and spine of the list together, and hence was unable to distinguish between a list
whose spine was shared and a list whose elements were shared.

Instead of addressing components of a variable by field name (e.g., “the head of the
tail of the tail of \textit{x}”), Bruynooghe introduces a notation \text{Comp}(\textit{T}, \textit{x})
which refers to all
components of variable \textit{x} with type \textit{T}. For example, if \textit{x} had type \text{List}(\textit{a}),
\text{Comp}(\text{List}(\textit{a}), \textit{x})
would refer to the spine of \textit{x}, while \text{Comp}(\textit{a}, \textit{x}) would refer to the elements of \textit{x}. By re-
ferring to all names in \text{Comp} notation, the analysis is guaranteed to terminate, as there is
a finite number of types and variables. This is a more useful way to derive a finite set of
names, because it allows us to make statements such as “all of the elements of the list are
aliased, but the spine is unique.” Note, however, that this approach combines unrelated
cousin nodes of the same type: for example, if \textit{x} has type \text{Pair}(\text{Int}, \text{Int}), the two fields of
the pair will be considered the same object for aliasing purposes.

Hughes [43] describes a CTGC analysis for a simple higher-order functional language.
Of particular note is the support for higher-order function values. The language is quite
limited, supporting only atomic values, lists and functions, with the CTGC being de-
signed specifically for lists. Unusually, this is not an aliasing or reference counting anal-
ysis at all. The optimisation is built upon two analyses: generation and inheritance. An
expression \textit{e} is said to \textit{generate} a memory cell if evaluating \textit{e} causes the cell to be allo-
cated, and the cell is reachable from the result of \textit{e}. An expression \textit{e} is said to \textit{inherit} a
memory cell from a subexpression \textit{g} if the cell is reachable from the result of both \textit{g} and \textit{e}.
The memory deallocation strategy rests on a key insight: if \textit{g} generates a cell, and \textit{e} does
not inherit that cell from \textit{g}, the cell may be freed after \textit{e} completes.

The analysis is advanced enough to be able to compute the generation and inheritance
sets precisely in higher-order function calls. The abstract semantics for a function is a
function mapping generation or inheritance information (respectively) from the function
CHAPTER 7. EXISTING SHARING AND REUSE ANALYSES

input to its output.\(^7\)

The system is not intended for making destructive updates to data structures, only freeing memory to save time in the runtime garbage collector. Furthermore, in the form described, the analysis does not collect memory until the function body is completely evaluated. Consider the analysis of the following contrived function:

\[
f(a, b) = \text{cons} \left( \text{length} \left( \text{cons} \ a \ \text{nil} \right) \right) \ b
\]

The system would correctly deduce that the inner \text{cons} expression generates a list, and the \text{length} expression (and hence the function body) does not inherit it, and so upon exiting the function, it would reclaim the inner \text{cons}’s cell without the aid of the runtime collector. However, it would not realise this in time to allow the outer \text{cons} to reuse the inner \text{cons}’s cell. Even if we adapted the analysis to reclaim memory immediately and reuse cells, it is fundamentally only able to reclaim memory that was generated by subexpressions of the function — it is actually closer to an escape analysis \cite{77} than CTGC. It never reclaims cells from actual parameters, which is often what we want. Consider the \text{reverse} example given above: if \text{reverse} is given a list that is not aliased or used again by the caller, it should be able to reuse the list cells by changing their tail pointers without allocating any memory. To do that, aliasing or at least reference-count information is required.

Mohnen \cite{73} describes a CTGC system similar to Hughes’ \cite{43}, but which operates on arbitrary algebraic data types rather than only lists (and only handles first-order programs). Like Hughes, this work only reclaims memory at the end of the evaluation of a function, and does not attempt to perform destructive update.

Shankar \cite{82} presents a destructive update optimisation for the functional subset of PVS, a simple strongly typed pure higher-order functional language with eager evaluation. Like our work, this focuses on converting copying array update operations \(a[(i) := v]\) into destructive array updates \(a[(i) \leftarrow v]\). For each top-level function \(f\), a specialised destructive version, \(f^D\), is produced. Instead of explicitly tracking aliasing between variables, the analysis result for each function is a list of formal parameters that must be dead in the caller, and not aliased to a given set of other formal parameters. For example, consider a function \(f(A, B)\):

\[
f(A, B) = A[(0) := 10] + B[(1) := 20]
\]

The analysis generates a destructive version \(f^D\) with the following definition and liveness

\footnote{This is fairly straightforward to specify due to the fact that the abstract environment is just a mapping from variable names to abstract values, and not a more complex data structure, such as aliasing pairs.}
condition:

\[ f^D(A, B) = A[0 \leftarrow 10] + B[1 \leftarrow 20] \quad LA(f^D) = \langle A \mapsto \{B\}, B \mapsto \emptyset \rangle \]

which would mean that, in order to use \( f^D \), \( A \) and \( B \) must be unique, and \( A \) must not alias \( B \). A call such as \( f(X, X) \) would not be transformed into a call to \( f^D \), because it would violate the liveness condition. This scheme has a richness not captured by most other analyses, including our own. It would be possible to generate a less aggressive destructive version that only updates \( B \):

\[ f^D(A, B) = A[0 := 10] + B[1 \leftarrow 20] \quad LA(f^D) = \langle B \mapsto \emptyset \rangle \]

This liveness condition means, as long as \( A \) and \( B \) have no other aliases, they are allowed to alias one another in the caller (because \( B \) is destructively updated after \( A \) goes out of scope). The call \( f^D(X, X) \) would be permitted as long as \( X \) was unique. However, Shankar’s analysis never generates such a version, as it always produces the most aggressive destructive version possible.

The analysis is also limited in that it does not track the aliasing between variables, only variable liveness. Ignoring lambda expressions (and having no other way to bind variables), this is sufficient, in combination with liveness conditions on function calls. While the analysis is aware of higher-order functions, it does not generate destructive updates within lambda (or let) expressions, so this simplistic analysis suffices. However, it does not suffice for Mars or, in general, any code with arbitrary variable bindings.

### 7.4.1 Mercury’s compile-time garbage collection system

Mazur [65] presents the most comprehensive compile-time garbage collection approach that we have found, and the most similar to our approach. Therefore, we investigate this in more detail than the above algorithms.

This compile-time garbage collection strategy was implemented for the Mercury programming language, a high-level purely declarative higher-order functional/logic language with eager evaluation. This work features an aliasing analysis derived from Mulkers [74],\(^8\) which computes detailed information about sharing of variables and their structural components at each program point. As with Bruynooghe, this analysis is aware of which pairs of variables are aliased, not just their reference counts.

Mazur’s approach is to use the aliasing information (and other information from static

\(^8\)Mulkers’ thesis was unavailable, so our understanding of this source is based solely on Mazur’s account of Mulkers’ work.
analyses) to statically transform code that constructs an object into code that reuses an existing dead object. An advantage of this approach is that it does not require the programmer to explicitly request reuse, or link the old and new structures in any way — the optimisation should automatically find an appropriate dead data structure to reuse. To this end, the work focuses on heuristics for determining which structure to reuse when one is required.

In this section, we give a high-level overview of Mazur’s approach. In later chapters, we describe our own approach in detail, highlighting the differences and similarities with Mazur’s.

The aliasing analysis (or “structure sharing analysis”) (Chapter 6 of [65]) is context insensitive, performed once per procedure. A data structure (what we call a name) is a variable $V$ with a selector $S$, written: $V^S$. For example, consider a linked list type, defined in Mercury as:

\[
\begin{align*}
\text{type} & \quad \text{list}(A) \rightarrow \text{nil} \\
& \quad ; \quad \text{cons}(A, \text{list}(A)).
\end{align*}
\]

A variable $X$ of type list(int) may be expressed as a data structure $X^\epsilon$ (the empty selector, $\epsilon$, indicates that the top-level cell of $X$ has been chosen). The head of the list may be selected with $X^{(\text{cons},0)}$ — the selector describes the constructor name and field index.\(^9\) The tail of the list may be selected with $X^{(\text{cons},1)}$, and the second element of the list may be selected with $X^{(\text{cons},1)\cdot(\text{cons},0)}$.

Recall that Bruynooghe [11] solved the imprecision of $k$-limiting by naming cells for aliasing purposes with $\text{Comp}(T, x)$, denoting all components of $x$ with type $T$. This prevents the analysis from generating infinite data structures at compile time, by merging together all components of the same type. However, it still had the problem of unrelated components being unnecessarily merged together. Mazur uses finite type graphs to achieve the same end in a way that better preserves aliasing information about cousin fields. This analysis merges components with the same type as an ancestor (which would result in an infinite type tree), but does not merge components with the same type as a cousin. This means that sharing information is kept precisely and to any depth, as long as the type does not recurse. Recursive types still (necessarily) suffer a loss of precision, but at least this keeps separate names for components of separate types, or cousins with the same type.

\(^9\)Mercury has special syntax for lists: [] for nil and [[]] for cons. While Mazur uses this notation, we use words for clarity and consistency with our syntax used in later chapters.

\(^{10}\)Mazur gives field indices starting from 1; we start from 0, again for consistency with other parts of this thesis. For example, where we write $X^{(\text{cons},0)}$, Mazur writes $X^{[1]}$.1
As such, in the abstract interpretation, selectors and data structures are always referred to in type-folded form (denoted with an overbar). For the list type, there is no abstract selector for list element 1 (the tail), because it has the same type as the list itself. Thus there are only two abstract list selectors: $\overline{\varepsilon}$ (the spine of the list) and $(\overline{\text{cons}},0)$ (the elements of the list). Thus, for a list variable $X$, we refer to the spine with $X\overline{\varepsilon}$, and the elements with $X\overline{(\text{cons},0)}$.

The abstract environment is a sharing set (what we call a sharing graph): a set of pairs of data structure names. In the abstract interpretation, these sets represent may alias information – if there is a pair between two data structures, then those corresponding memory cells might be shared; if not, then they must not be shared. Hence, the sharing set:

$$\{(A\overline{\varepsilon} - B\overline{\varepsilon}), (B\overline{\varepsilon} - C\overline{\varepsilon})\}$$

denotes that some part of the spine of $A$ may be shared with some of the spine of $B$, and that some of the spine of $B$ may be shared with some of the spine of $C$, but no part of $A$ is shared with any part of $C$. This may be possible, for example, if $B$ shares different elements with $A$ and $C$ (so there is no overlap between $A$ and $C$), or if $B$ might be shared with either $A$ or $C$ but not both.

Consider the Mercury version of `append`, a predicate for concatenating two lists, shown in Example 7.7. This code takes two parameters, $X$ and $Y$ and returns a result $Z$. It pattern-matches on $X$, and if $X$ is nil, it just assigns $Y$ to $Z$. If $X$ is a cons cell, it recursively calls `append` on the tail and $Y$, then constructs a new cons cell with $X$’s head and the result of the recursive call.

**Example 7.7** The `append` predicate in Mercury (from [65])

1: $\text{pred append(list(T) :: in, list(T) :: in, list(T) :: out) is det.}$
2: $\text{append(X, Y, Z) :-}$
3: $\quad (\quad )$
4: $\quad X = \text{nil},$
5: $\quad Z = Y$
6: $\quad ;$
7: $\quad X = \text{cons}(Xe, Xs),$
8: $\quad \text{append}(Xs, Y, Zs),$
9: $\quad Z = \text{cons}(Xe, Zs)$
10: $\quad ).$

The structure sharing analysis computes that, for any call to `append(A, B, C)`, the following sharing set will result:

$$\{((C\overline{(\text{cons},0)}) - A\overline{(\text{cons},0)}), (C\overline{\varepsilon} - B\overline{\varepsilon})\}$$
That is, some of the elements of \( C \) may share with some of the elements of \( A \), and some of the spine of \( C \) may share with some of the spine of \( B \). This is a correct, general structure sharing statement for \texttt{append}.

This information can be used to enable destructive update in \texttt{append} and in other procedures that call it (Chapter 9 of [65]). The goal of the reuse analysis is to identify constructor calls, and find memory cells that may be garbage collected at that program point. In \texttt{append}, there is only one constructor call, on Line 9 (note that the very similar Line 7 is a deconstruction or pattern match on an existing cell). Hence, we attempt to find a free memory cell at Line 9.

If we assume that the input arguments \( X \) and \( Y \) are unshared, the sharing set before Line 7 is \( \emptyset \) and the variable \( X \) is not used again after Line 7. Therefore, the deconstruction \( X \equiv \texttt{cons}(Xe, Xs) \) is considered to make the data structure \( X \) available for reuse. At Line 9, the construction requires a heap cell of the same size as the one freed on Line 7, so these two program points are said to be a matching deconstruction-construction pair. Thus, the procedure can be transformed such that Line 9 reuses the memory cell named by the variable \( X \).

This is all predicated on the assumption that \( X \) is not shared when the procedure is called. Since \texttt{append} could always be called with \( X \) shared to another object, this version is not generally safe to call. Hence, this does not replace the existing non-destructive version of \texttt{append}; rather, it creates a reuse version of the procedure, which is called when the caller knows that its input is unshared. The system described by Mazur only generates two versions of the code: one original un-optimised version, and one version optimised with as many reuse opportunities as possible. If a caller knows that it is safe to call the reuse version, it does so, but otherwise, it falls back to the completely un-optimised version. Mazur describes a work-in-progress system (Chapter 13 of [65]) for describing multiple differently optimised versions of a procedure, so that many more versions can be created without the exponential explosion of all possible versions.

This is the most advanced compile-time garbage collection system we have seen described, and very similar to our first-order CTGC system described in Chapter 8. A major drawback of this and other sophisticated CTGC systems\textsuperscript{11} is the lack of support for higher-order procedures or procedures as first-class values. The nature of analysing a procedure is that it requires knowledge about all of the procedures that it calls. If a procedure calls a variable, then it cannot make any assumptions about the callee without a much more sophisticated analysis that explicitly tracks the sharing information of closures. Mazur makes a convincing argument that an analysis aware of higher-order

\textsuperscript{11}Notwithstanding type-based systems like Clean, which have a much simpler view of sharing information but benefit from supporting higher-order functions as easily as the rest of the type system.
procedures may not be necessary due to the Mercury compiler’s optimisation that specialises higher-order functions, creating a first-order version for each known usage of the function. This is a reasonable argument, but it only works for straightforward cases: in any situation where the compiler needs to create a closure at runtime, that approach will not work. We discuss this possibility and give examples of failing cases in Chapter 9.

7.5 Dynamic memory reuse flags

Lee et al. [59] introduce a compile-time garbage collection scheme for ML, that inserts explicit free operations into the program. It deliberately inserts the free immediately before allocation instructions, so that the compiler will convert the free and allocation into a memory reuse. The work is only applied to first-order programs with binary tree data structures.

Of key interest to this thesis is that the decision to call free may be informed by a Boolean argument value at runtime. Consider Example 7.8a, an ML program that makes a copy of the left side of a binary tree. Lee et al.’s system transforms this code into Example 7.8b, which will free each node of the tree immediately before reconstructing it.

Example 7.8 Partially copying a tree in ML, before and after optimisation (from [59])

<table>
<thead>
<tr>
<th>fun copyleft ( t ) =</th>
<th>fun copyleft ( \beta, \beta_{ns} ) ( t ) =</th>
</tr>
</thead>
</table>
| \begin{align*}
\text{case } t \text{ of} \hfill \text{case } t \text{ of} \\
\text{Leaf} \quad &\Rightarrow \text{Leaf} \hfill \text{Leaf} \quad &\Rightarrow \text{Leaf} \\
| \text{Node} (t_1, t_2) \Rightarrow \text{let } p = \text{copyleft } t_1 \\
\text{in } \text{Node} (p, t_2)
\end{align*}
| \begin{align*}
\text{let } p = \text{copyleft } (\beta \land \beta_{ns}, \beta_{ns}) \ t_1 \\
\text{in } (\text{free } t \text{ when } \beta; \text{Node} (p, t_2))
\end{align*} |

Note that this free is only legal if the actual parameter \( t \) is not aliased or subsequently used by the caller — otherwise, this would be catastrophic, as the caller would use memory after it is freed. A common solution to the problem is to generate two versions: the normal un-optimised version and a specialised reuse version [11, 65]. Instead of specialising, this work generates a single version with two additional dynamic flag parameters: \( \beta \) and \( \beta_{ns} \).

The flag \( \beta \) is true only if the actual parameter \( t \) is unshared, while \( \beta_{ns} \) is true only if all of the sub-components of the tree are unshared. The caller of copyleft must ensure these conditions are met. The top-level term \( t \) will only be freed if \( \beta \) is true, while on subsequent recursive calls, \( t \) will only be freed if both \( \beta \) and \( \beta_{ns} \) are true.

\(^{12}\)Higher-order programs are partially supported via closure conversion; supporting higher-order this way is the same argument given by Mazur [65], which we discuss in Section 9.2.
This approach is a mix of compile-time and runtime garbage collection — the dynamic flags are used to make runtime decisions, but the values for the flags are based on static analysis, so only a small amount of work is done at runtime.

### 7.6 Aliasing analysis for impure and unsafe languages

Languages without interface integrity — in particular, those without advanced pointer manipulation — present a more complex picture for aliasing analysis. If we take C as one of the less safe languages, we see that any aliasing analysis must handle or approximate the following: pointer fabrication (making pointers from arbitrary integers), taking the address of stack variables, pointers to the middle of an array, functions modifying arbitrary pointers, and mutation of global pointer variables.

The situation is in fact more dire than that: almost any C statement can have almost any effect imaginable, since an array access with a variable integer for the index can address any memory location in the system. Even if an analysis were to assume that a C program is well-behaved, it must still cope with broader issues than we deal with in safe languages. In particular, the fact that pointers can refer to stack variables and be passed to arbitrary functions makes it much more difficult to reason about the value of a variable.

Impure languages also represent an entirely different use case: we typically do not detect aliasing in these languages for the purpose of introducing automatic destructive update, as programmers are free to manually request destructive update. Rather, aliasing analyses have other uses, including:

- **Pointer replacement.** If we know that \( p \) points to a variable \( y \), we can transform the statement \( x = *p \) into \( x = y \) [23].

- **Register promotion.** If some code repeatedly reads and writes a pointer, and we know that the pointer is not aliased to other in-scope variables, we can use a register to store the intermediate values of the pointer [72].

- **Feedback (debugging).** We can report information to the programmer about which variables may be aliased.

Therefore, aliasing analysis still has important applications for impure and unsafe languages.

We have so far seen analyses that determine that a variable \( a \) may be aliased to some other variable, and more sophisticated analyses that determine that two variables \( a \) and \( b \)
may be aliased to one another. Note that there is some implicit indirection here: when we give an aliasing pair \( \{a, b\} \), we are not suggesting that the variables \( a \) and \( b \) themselves occupy the same memory (which would imply that assigning to \( a \) would also update \( b \)); rather that the variables \( a \) and \( b \) are references to the same memory cell. This simplistic view can be taken in any number of languages, including Mars, Java and Python, where variables can hold aliased references, but cannot be aliased to other variables.

If we consider a language like C that allows pointers to stack variables, the domain is more complex: we need a syntax for referring to the memory locations of the variables themselves, as well as the cells they point to. If \( a \) and \( b \) are pointers to the same memory location, we must describe the situation with the alias pair \( \{\ast a, \ast b\} \). The bare (un-starred) notation refers to the variable’s own memory location on the stack. For instance, if a pointer \( p \) points at the memory location for \( a \), we have the pair \( \{\ast p, a\} \) [54].

Landi and Ryder [54] present an early solution to the interprocedural pointer may-alias problem in C. The authors found earlier analyses of Fortran to be inappropriate in C, for the reasons mentioned above, and devised a domain where variables could point to arbitrary memory locations, including those of other variables. Their domain uses sets of aliases, where each alias is an unordered pair of object names, defined by the following grammar:

\[
\text{object\_name} \quad \rightarrow \quad \ast\text{object\_name} \\
\quad \mid \quad \text{object\_name.\_field\_of\_structure} \\
\quad \mid \quad \text{variable}
\]

As with Jones and Le Métayer’s earlier aliasing analysis, Landi and Ryder use \( k \)-limiting to truncate arbitrarily large data structures at some finite depth \( k \), ensuring that all object names are finite. Landi and Ryder’s implementation reported far fewer superfluous aliases than prior work by Weihl [96].

An alternative domain is a points-to analysis [23]. In this domain, rather than identifying pairs of locations that alias, a canonical name is given to each abstract memory location, and pointer variables are identified that point to those locations. Instead of the alias pair \( \{\ast p, a\} \), we would have the points-to pair \( (p, a) \), which says “\( p \) points to the location of the variable \( a \).” At first glance, this representation seems to simply be alias pairs with the left side having an implicit \( \ast \), but it turns out to be more concise and have some different characteristics. If we consider the C code in Example 7.9, we would calculate the alias pairs \( \{\{\ast p, x\}, \{\ast q, x\}, \{\ast p, \ast q\}\} \); that is, \( p \) and \( q \) both point to \( x \), and also point to the same thing as one another.
CHAPTER 7. EXISTING SHARING AND REUSE ANALYSES

Example 7.9 Aliased pointers to a stack variable in C

1: int x; int* p; int* q;
2: p = &x;
3: q = p;

The points-to information for the same example would be \{(p, x), (q, x)\} — note that p and q are not directly related, only indirectly through their shared pointing at x. Not only is this more concise, but it also can prevent superfluous aliasing, as in Example 7.10, due to Emami et al. [23].

Example 7.10 Modifying an aliased pointer in C (from [23])

1: int** x; int* y; int z, w;
2: x = &y;
3: y = &z;
4: y = &w;

The alias pairs after Line 3 in this example are:

\{\{*x, y\}, \{**x, *y\}, \{*y, z\}, \{**x, z\}\}

After Line 4, which mutates y, the alias pairs are:

\{\{*x, y\}, \{**x, *y\}, \{*y, w\}, \{**x, z\}, \{**x, w\}\}

Note that the idea that **x aliases z is true after Line 3, but false after Line 4, yet it was not removed from the alias set. Conversely, the points-to information never explicitly stated that **x aliased z. After Line 3, the points-to set is \{(x, y), (y, z)\} and after Line 4, it is \{(x, y), (y, w)\}. The points-to domain is therefore more precise and concise for tracking pointers to stack locations. For languages that do not allow pointers to the stack, there is no advantage to the points-to domain — note that the above example of over-aliasing was only possible because an aliased location was modified.

It should also be noted that points-to is inappropriate for dynamically allocated pointers (which in our case is all pointers), because it does not track the relationships between pointers. A typical points-to analysis simply gives one name, “heap,” to all dynamically allocated locations, making it impossible to distinguish pointers to different heap cells. Emami et al. argue that stack and heap analyses should be separate. Therefore, we consider the points-to analysis inappropriate for analysing Mars code and other languages that do not allow pointers to the stack, or by-reference argument passing.

The contribution of Emami et al. is handling function pointers [23]. The analysis is
context sensitive, so when a call to a function pointer is made, the analysis has information about what functions it may point to. Upon encountering a call $f(x_1, \ldots, x_n)$, where $f$ is a function pointer, the analysis can use the points-to information to obtain a set of concrete functions that may be pointed-to by $f$. It then analyses a call to each of these functions in place of $f$, and then joins all of the results together. This is a simpler situation than our higher-order analysis faces, because the analysis is context sensitive (having access to information about $f$), and also because function pointers in C do not carry closure data.

We have also seen aliasing analysis applied to memory-safe but impure high-level languages like Java. Guyer et al. [31] describe a compile-time garbage collection system for Java, which operates in conjunction with a traditional mark-and-sweep collector. The analysis outputs code that runs on the Jikes RVM, a research-oriented implementation of the Java Virtual Machine (JVM). The authors modified the RVM to support an explicit `free` operation, which is not supported on the regular JVM. Their analysis makes use of aliasing and liveness information to statically determine when memory becomes unreachable, and insert explicit calls to `free`. The traditional garbage collector is still used to free memory that could not be marked for removal at compile time.

Guyer et al. claim that the explicit frees produce a significant speed increase in short-lived allocations in a traditional mark-and-sweep collector. However, the authors admit that their analysis could not enhance the performance of a state-of-the-art generational garbage collector. This CTGC approach may be contrasted with work by Khedker et al. [53], discussed later in this chapter, which inserts `null` assignments (as a hint for the runtime garbage collector) instead of explicitly freeing memory.

### 7.7 Dynamic aliasing analysis

Mock et al. [72] describe a dynamic version of the points-to analysis for C programs. The program is instrumented such that at runtime, every time a pointer is dereferenced (for loading or storing), the address of the pointer is noted, and mapped onto one of a finite set of object names. For pointers to global variables or the stack, the address is mapped onto the name of the variable. For pointers to the heap, the address is mapped onto the program point of the `malloc` site that allocated the cell at that address, so all pointers to cells allocated by the same static program point will be considered aliased.\(^\text{13}\)

The output from the dynamic approach has the same form as the static approach, but with quite different properties. While a static points-to approach will detect all possible

\(^{13}\)Recall that static points-to analyses make similar approximations of heap addresses, with some analyses considering all heap addresses to be the same name [23].
names that a pointer may point to, it may approximate conservatively and report some targets that may never occur in practice. By contrast, the dynamic approach will only ever report points-to targets that truly occurred in practice, but it may not report all possible targets (it is potentially unsound). It should be noted that even though this approach reports only true pointer targets, it will still only give approximate aliasing: even if two pointers point at the same target, in some code path, it does not mean that those pointers have the same address at the same time. Furthermore, heap-allocated object names may represent any number of distinct addresses at runtime.

The main advantage of this analysis is when used in conjunction with static analysis — the results complement one another. If both analyses give the same results for a given pointer, then the static analysis must be optimal, and the dynamic analysis must be complete. A suggestion by Mock et al. is that a cheap static analysis could be run and compared with the dynamic analysis, which could then prompt an expensive and more precise static analysis only for the cases where it proved to be sub-optimal. The dynamic analysis could also be useful for debugging, as a user wishing to track down an aliasing bug is likely to be satisfied with being told what aliasing is actually occurring, and not the other theoretical aliasing.

The dynamic approach is not one we have explored in this thesis. One possible avenue for future work would be applying a similar dynamic analysis to complement our aliasing analysis, to empirically test the precision of the static analysis.

7.8 Static pointer nullification

Khedker et al. [53] present an analysis for Java programs that inserts null assignments into programs where it is safe to do so. As such, it is not a true compile-time garbage collection scheme, but a compile-time transformation that improves the behaviour of the runtime garbage collector. The transformation is performed partially by hand, with Java programs being manually transformed into an appropriate data structure with manual inlining and tail call optimisation. (This could theoretically be automated.) Programs are automatically analysed, and then null assignments are manually inserted into the original Java program. The goal of this transformation is to remove unnecessary references to heap cells, allowing the runtime garbage collector to reclaim the memory earlier. This is shown to free a significant amount of memory sooner in several programs, as well as reduce the “high water mark” for memory usage in one benchmark. It also shows slight to large improvements in execution time, due to the use of a copying garbage collector.

This is quite different to other CTGC approaches, most of which insert free or de-
structive update instructions, requiring that the reference is *definitely not aliased*. This approach inserts **null** assignments, which only requires that the reference is *not live* (will not be used again) — it is perfectly safe to set a reference to **null**, even if the object it refers to is aliased. As well as a liveness analysis, this paper features the inverse of the normal aliasing analysis, a *must-alias* analysis, which determines whether a reference is *definitely aliased* (in which case it is deemed not profitable to assign a **null** value).

The key improvement of this paper is in the liveness analysis, which determines not only when variables will no longer be used, but also when specific fields of a variable will no longer be used despite the variable itself still being live. This allows the runtime to collect part of a data structure while other parts of it are still in use. This liveness analysis is far more powerful than the simple variable-based one used in this thesis, and therefore these techniques could be considered in unison with ours.

### 7.9 Conclusion

Our language Mars is designed as an imperative programming language with declarative semantics, which means Mars cannot allow aliased objects to be destructively updated. Still, for reasons of efficiency, the compiler should destructively update objects when doing so would not alter the semantics of the program. Specifically, this is true when the object being modified has no aliases, and is not referred to again. This seems more important in an imperative language, like Mars, that encourages the use of imperative data structures like arrays and hash tables, which programmers will expect to be able to update in constant time.

This chapter is an account of the history of this memory reuse technique. We first looked at type-based approaches, which are common in existing languages, and explicitly make the programmer request destructive update. In Haskell, Mercury, and other languages, a *store* is conceptually a dictionary mapping references to values. By ensuring the store is not duplicated, the language can allow the *references* to be duplicated and still perform destructive update. This preserves the pure semantics, but it does not protect the programmer from accidentally updating an aliased reference. Alternatively, Clean has a *uniqueness* system, which explicitly encodes whether or not an object is unique in the type system. This means that Clean arrays can be updated destructively, and it is a compile-time error to do so if the array is not unique. This is closer to the approach we are interested in, but it is not automatic, and is also not very precise (we gave examples that trick the system into believing a value is aliased, when in fact it is not).

*Compile-time garbage collection* (CTGC) is an automated analysis and optimisation which
determines at compile time when memory is eligible for collection. This can be used to insert \texttt{free} operations into the source code, saving the runtime garbage collector the effort, or in more advanced cases, to replace allocation operations with a reuse of existing dead memory cells, saving both a collection \textit{and} an allocation. This latter usage is known as \textit{automatic destructive update}, and it is the approach we take with Mars. Earlier works saw compile-time garbage collection as an abstraction of reference count, tracking the number of aliases associated with each variable. More advanced analyses track pairs of aliases between variables, allowing aliased variables to be considered unique again, once all of their aliases are dead. Authors have also developed sophisticated methods for tracking aliasing between components of variables while keeping compile-time data structures finite.

CTGC is more complicated and less precise when applied to languages with side-effects, or languages which allow pointers to stack locations. In the latter case, it requires knowledge of not only which variables point to the same memory cells, but also which variables point to which other variables. Some analyses find it helpful to use a \textit{points-to} graph for stack pointers, which has various advantages and disadvantages over alias pairs.

Lastly, we surveyed several alternative approaches to CTGC. A dynamic analysis of pointers can be used to determine true aliasing information, as opposed to the conservative information gathered by static analysis approaches. This can be used in conjunction with static analysis results to determine when the analysis is optimal, along with several other useful statistics. Instead of inserting \texttt{free} operations when a cell is \textit{definitely} dead, an analysis may set the pointer to \texttt{null} when the pointer itself is dead, and the cell it points to \textit{might} be dead. This means the runtime garbage collector will still be required to collect the dead memory, but it can permit dead memory to be collected earlier than it otherwise would have been.

In the next chapter, we describe our own uniqueness inference system for Mars, largely equivalent to that of Mercury, but with a few additional features. Over the coming chapters, we will develop this system into a full context-insensitive higher-order analysis that is as precise for higher-order calls as it is for first-order.
Chapter 8

First-order uniqueness inference

8.1 Introduction

In order to build an optimisation that automatically replaces creation instructions with destructive update instructions, the compiler must build an approximate view of which objects are aliased at any given program point. Doing this at compile time requires a static analysis that simulates the aliasing behaviour of each instruction in the program. The bulk of the work is performed in a context-insensitive abstract interpretation of the program’s execution units, with most of the actual decisions about when to perform destructive update being performed in a context-sensitive static analysis or at runtime.

The context-insensitive uniqueness inference analysis is presented in two stages to simplify the explanation. This chapter introduces the analysis without much regard for the higher-order features of the language (closure construction and application). The analysis is still sound in the presence of closures, but makes no attempt to preserve aliasing information in their presence. This approach simplifies the analysis considerably, with the caveat that a lot of information is lost when applying a closure. We will recover this information in Chapter 9.

The results of the first-order analysis are comparable to that of Mazur [65]. We use a different notation and terminology to that of Mazur, as we have found our notation more suitable for extension to higher-order analysis. The novelty in our work is the precise handling of higher-order cases, which will be covered in detail in Chapter 9.

We describe the memory aliasing abstractly using a sharing graph, an undirected cyclic graph, the nodes of which are variable names or expressions identifying components of variables (e.g., $x.0$ identifies the first field of the structure variable $x$). An edge between two nodes indicates that those names may refer to the same memory location (this is a
In order to maintain this abstract state over procedure calls, the compiler must determine the aliasing behaviour of each procedure that is called. Our analysis is **condensing** (as described in Section 6.6), which makes it naturally context-insensitive. Each procedure in the program is condensed into a procedure summary, or **uniqueinfo**, which describes the effect of calling a procedure, without requiring re-analysis of that procedure.

The remainder of this chapter is organised as follows. In Section 8.2, we walk through the workings of the analysis at a high level, with examples and diagrams. This should give the casual reader a good intuition about how the analysis works. In Section 8.3, we formally define the abstraction, and give the abstract memory denotational semantics, for a precise mathematical description of the first-order analysis. In Section 8.4, we compare our approach with that of Mercury. Finally, Section 8.5 concludes.

### 8.2 Motivating examples

Before formally describing the first-order analysis, we first present some motivating examples in this section.

#### 8.2.1 Knowing when to optimise for destructive update

We begin by attempting to optimise the simple program fragment shown in Example 8.1a. We shall assume that \( a \) is an array of integers which is known not to be aliased with any other variable. Semantically, this code non-destructively updates the first element of \( a \), storing the resulting array in the new variable \( a_1 \). It then non-destructively updates the second element of the original \( a \), storing the result in the variable \( a_2 \). For example, if \( a = [1, 2, 3] \), after executing these instructions, we find that \( a = [1, 2, 3] \), \( a_1 = [10, 2, 3] \), and \( a_2 = [1, 20, 3] \).

<table>
<thead>
<tr>
<th>Example 8.1</th>
<th>Code that updates the same array twice, before and after optimisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( a_1 = \text{array}_\text{replace}(a, 0, 10) )</td>
<td>1: ( a_1 = \text{array}_\text{replace}(a, 0, 10) )</td>
</tr>
<tr>
<td>2: ( a_2 = \text{array}_\text{replace}(a, 1, 20 } \Omega{a}</td>
<td>2: ( a_2 = \text{array}_\text{replace}_d(a, 1, 20) )</td>
</tr>
</tbody>
</table>

The naïve implementation of \( \text{array}\_\text{replace} \) simply copies the array and then modifies one element. Clearly, we need to actually make a copy on Line 1, because there are to be two distinct arrays afterwards. However, the \( \Omega \) annotation on Line 2 indicates that \( a \) is a **dead variable** (it will not be mentioned again), so we can do better than simply copying the array — we can destructively modify its second element, then copy the array’s pointer into \( a_2 \). A simple local analysis can compute the \( \Omega \) annotation. The resulting code
will have indistinguishable behaviour, since \( a \) will never be accessed again. To achieve this, we simply change the call to \texttt{array.replace} to a call to \texttt{array.replace} (the “d” is for “destructive”), which is a special version of \texttt{array.replace} that destructively mutates its first argument, and returns it, without making a copy. The optimised code is shown in Example 8.1b.

### 8.2.2 Aliasing between variables

We now extend the above example in Example 8.2a, by introducing a second variable, \( b \), that \texttt{array.replace} is applied to. The code is equivalent to Example 8.1, except it first makes an alias of \( a \) called \( b \), and rather than replacing \( a \) twice, it replaces \( a \), then \( b \). For example, if \( a = [1, 2, 3] \), after executing these instructions, we find that \( a = [1, 2, 3] \), \( b = [1, 2, 3] \), \( a_1 = [10, 2, 3] \) and \( b_1 = [1, 20, 3] \). In this version, we are required to track the aliasing between variables.

**Example 8.2** Code that updates two copies of an array, before and after optimisation

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Initial State</th>
<th>Optimised Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( b = a )</td>
<td>1: ( b = a )</td>
<td>1: ( b = a )</td>
</tr>
<tr>
<td>2</td>
<td>( a_1 = \texttt{array.replace}(a, 0, 10) ) ( \Omega {a} )</td>
<td>2: ( a_1 = \texttt{array.replace}(a, 0, 10) )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( b_1 = \texttt{array.replace}(b, 1, 20) ) ( \Omega {b} )</td>
<td>3: ( b_1 = \texttt{array.replace}(b, 1, 20) )</td>
<td></td>
</tr>
</tbody>
</table>

Our analysis is an abstract interpretation which computes an abstract state in between each instruction (at each *program point*). For the first-order analysis, the abstract state is a sharing graph, with one node for each variable and variable component. In a sharing graph, an edge between two nodes means there *may* be an alias between memory cells of those names; if there is no edge between two nodes, then there *is no* alias between any cells of those two names. Mathematically, an edge is represented as an unordered pair: e.g., \( \{a, b\} \). Figure 8.1 shows the step-by-step abstract interpretation of this code.

The initial state shows the single node, \( a \), representing the memory occupied by the array object itself. After Line 1, an additional node for the array \( b \) has been introduced. The sharing graph shows that \( a \) and \( b \) may be aliased (in this case, we know that they are). After Line 2, we introduce \( a_1 \), which we know is not aliased to \( a \), because \texttt{array.replace} creates a copy of the input array. We also take into account the fact that \( a \) is dead after Line 2, and delete it from the graph. (Throughout this thesis, dotted outlines show nodes and edges that have been removed from the graph; they are for illustrative purposes only and may be considered to be completely removed.) Finally, after Line 3, we introduce \( b_1 \), which is not aliased to \( b \), and as before, we delete \( b \) from the sharing graph.

So in our final sharing graph, we see that \( a_1 \) and \( b_1 \) are not aliased. Note that the *final* sharing graph is not all that we are interested in: we will make use of each of the sharing
graphs generated along the way.

![Diagram](image.png)

(a) Initial  (b) Line 1  (c) Line 2  (d) Line 3

Figure 8.1: Internal sharing graph for Example 8.2a after each line of analysis

Now we may attempt to optimise the calls to `array_replace` for destructive update. On Line 2, we consult the sharing graph from the previous line (Figure 8.1b), and find that $a$ may be aliased to $b$. Even though $a$ will never be used again, if we mutate it, we may also be mutating $b$; therefore, we cannot optimise Line 2. On Line 3, we again consult the sharing graph (Figure 8.1c), and find that $b$ is not aliased. Furthermore, we know that $b$ will be dead after this line. Therefore, it is safe to optimise this call to `array_replace` into a call to `array_replace_d`, as shown in Example 8.2b. This is again the best we can do: we need to create at least one copy, because we have two distinct arrays afterwards.

### 8.2.3 Functions and specialisation

Consider the function `replace_twice` in Example 8.3a, a continuation of the above example expanded into a full function. This function performs the same operation as above, but then returns the two arrays $a_1$ and $b_1$ in a pair. The new function `replace_twice_test` creates two arrays, and calls `replace_twice` on each of them. There is an important difference between $x$ and $y$: $x$ is never mentioned again after calling `replace_twice`, whereas $y$ is returned at the end.

We first analyse `replace_twice`. The analysis is the same as that of Figure 8.1, but the optimisation is different — since $a$ is a parameter to the function, we will not know whether $a$ will be used again until the function is called. Therefore, we cannot optimise the call to `array_replace`.

---

1The final sharing graph for `replace_twice` is omitted, since it is complex and unimportant for this example.
8.2. MOTIVATING EXAMPLES

Example 8.3 A function that updates two copies of an array, before and after optimisation

```
1: def replace_twice(a):
2:     b = a
3:     a1 = array_replace(a, 0, 10) \Omega\{a\}
4:     b1 = array_replace(b, 1, 20) \Omega\{b\}
5:     return Pair(a1, b1)
6: def replace_twice_test():
7:     x = [1, 2, 3]
8:     y = [4, 5, 6]
9:     x1 = replace_twice(x) \Omega\{x\}
10:    y1 = replace_twice(y)
11:    return Triple(x1, y, y1)
```

We can now optimise `replace_twice_test` for destructive update, as we did previously with calls to `array_replace`. On Line 9, we know that `x` is not aliased, and will be dead after this line. Therefore, this call to `replace_twice` should be optimised to destructively update `x`. On Line 10, while `y` is not aliased, it is still live after this line, so this call to `replace_twice` must not be optimised. Unfortunately, `replace_twice` does not destructively update the array, since we do not know whether `a` would be referenced after the call. This is true in general, but not true in the specific case of Line 9. Therefore, we create a specialised version of `replace_twice` called `replace_twice_d`, which, unlike the primary version, is allowed to assume that `a` is not aliased, and will not be used after the call, as shown in Example 8.3b.\(^2\)

We can then replace Line 9 (but not Line 10) with a call to `replace_twice_d`.

The techniques for specialising and selecting which version of a function to call are discussed in Chapter 10. For the remainder of this chapter, and the next, we focus on computing the sharing between variables.

8.2.4 Joins and bottom-up analysis

Example 8.4 shows a function with two code paths, where the path taken is determined by the input. As demonstrated in Figure 8.2, the sharing graph after Line 3 is \{\{a, r\}\}; after Line 5 it is \{\{b, r\}\}; before Line 6 (just after the `if` statement), these code paths are joined by taking the union of the sharing graphs, resulting in \{\{a, r\}, \{b, r\}\}. The structure does not over-generalise by assuming that `a` and `b` must alias; there is no code path on which they do.

This shows the advantage of a data structure that explicitly records all aliasing pairs, versus one that more closely models the concrete semantics by binding names to abstract

\(^2\)The specialisation technique was not implemented in the compiler, but we did implement a dynamic technique for selecting destructive update primitives. Details are given in Chapter 10.
Example 8.4 Function with two possibilities, based on input

```python
def if_then_else(c, a, b):
    if c:
        r = a
    else:
        r = b
    return r
```

Figure 8.2: Internal sharing graph for if_then_else after selected lines of analysis

“memory cells.” The latter would not allow us to express “a may alias c and b may alias c, but a does not alias b.”

As well as variables, the analysis must also track the aliases of the function’s result value. We introduce a special variable, $, which represents the function result (it may be helpful to think of the statement “return r” as “$ = r”). Because the return statement simply binds a variable, on Line 6, we alias $ to r. This causes everything aliased to r (a and b) to become aliased to $.

After r is projected away, the final sharing graph is represented mathematically as the following set of unordered pairs: \{{$}, \{a, b\}\}. From here, we can determine the abstraction of the if_then_else function, which we call a uniqueinfo:

\[
\text{if\_then\_else}_a \quad :: \quad \text{(Name, Name, Name)} \rightarrow \text{Name} \rightarrow \text{Sharing} \\
\text{if\_then\_else}_a (c, a, b) \quad \$ \quad = \quad \{\{a\}, \{b\}\}
\]

A uniqueinfo is an abstract function that describes what aliasing a function will create when it is called. We could equivalently express it as a lambda: the uniqueinfo of if_then_else is \(\lambda (c, a, b) \cdot \{\{a\}, \{b\}\}\). Uniqueinfos always take two arguments and produce a sharing graph: the first argument is a tuple of names, with one name for each concrete argument; the second argument is a single name which represents the name of the result (by convention, we call this parameter $, but it could be called anything). The
resulting sharing graph is all of the edges created between the result and the arguments.

The uniqueinfo can be applied by passing the names of the arguments and result in the caller. For example, if another function were to call if_then_else with the statement:

\[ w = \text{if}_\text{then}_\text{else}(x, y, z) \]

then our analysis would evaluate the uniqueinfo:

\[
\text{if}_\text{then}_\text{else}_a([x], [y], [z]) [w] = \{[w], [z], \} \}
\]

which would result in the insertion of an edge between \(w\) and \(y\), and an edge between \(w\) and \(z\), in the caller’s sharing graph.

### 8.2.5 Using structure information

Consider the function pair\_const in Example 8.5, which places two values (of arbitrary type) in a pair, copies the pair, and retrieves one of the values. This function is equivalent to one that simply returns its first argument. For example, if \(x = 1\) and \(y = 2\), this will set \(p = \text{Pair}(1, 2)\) and \(q = \text{Pair}(1, 2)\), then return \(q.Pair/0\), which is 1. It is critical that any caller of this function know that the result may be (in this case, is) aliased to \(x\); furthermore, it is desirable for the caller to know that the result is not aliased to \(y\).

**Example 8.5 Returning the first argument via a pair**

1: `def` pair\_const\((x, y)\):
2: 
3: `p` = `Pair(x, y)`
4: 
5: `q` = `p`  
6: `return` `q.Pair/0`

Figure 8.3 shows the sharing graph at each stage of the abstract interpretation of pair\_const. Here we see that the sharing graph is capable of storing more than just aliases between variables: it tracks individual component cells of each variable. For example, on Line 2, \(p\) itself is not aliased to anything, but we know that the first field of \(p\) \((p.0)\) is aliased to \(x\), while the second field of \(p\) \((p.1)\) is aliased to \(y\). The name \(p\) refers only to the top-level cell of the pair — it is not aliased to its own children, which makes it possible for \(x\) to be aliased to \(p.0\) without being aliased to \(p\) itself.

Note that on Line 3, when \(p\) is copied to \(q\), the analysis ensures that a) all parts of \(p\) are aliased to the corresponding parts of \(q\), and b) all aliases of each part of \(p\) are copied as aliases of the corresponding part of \(q\). In effect, \(q\) holds the same position in the graph as \(p\) held on the previous step — \(q.0\) aliases \(x\), and \(q.1\) aliases \(y\).
On the last step, we see $ alias $q.0$. By extension, we alias $ to everything aliased to $q.0$. Fortunately, this includes $x$, but not $y$, so the result of the analysis is fully precise. The final sharing graph is $\{\{\$, x\}\}$, and so we derive the function’s uniqueinfo:

$$\text{pair\_const}_a (x, y) \$ = \{\{\$, x\}\}$$

By contrast, consider the function $\text{array\_const}$ in Example 8.6. Like $\text{pair\_const}$, this simply returns the first argument, but does so by placing the values into an array rather than a pair. For example, if $x = 1$ and $y = 2$, this will set $a = [1, 2]$ and $b = [1, 2]$, then return $b[0]$, which is 1.

**Example 8.6** Returning the first argument via an array

1: def array\_const(x, y):
2:  a = [x, y]
3:  b = a
4:  return array\_ref(b, 0)

Figure 8.4 shows the sharing graph for the abstract interpretation of $\text{array\_const}$. This is nearly the same as $\text{pair\_const}$, except that the analysis does not track the individual array elements, instead combining them together into a single node called “$a[]$”. The consequence of this is that when element 0 is pulled out of the array, the analysis does not know whether it aliases $x$, $y$, both or neither, so it has to assume that both $x$ and $y$ are aliased to the result. The uniqueinfo for $\text{array\_const}$ is:

$$\text{array\_const}_a (x, y) \$ = \{\{\$, x\}, \{\$, y\}\}$$

This is an inherent limitation of the analysis: we cannot generally track the individual
array elements because we do not know how many there will be at compile time (unlike algebraic data values, where the number of elements is fixed statically). Therefore, we combine them into a single node. Fortunately, program code typically treats all array indices the same way (for example, looping over an array)\(^3\), so we usually retain a lot of precision even when combining elements in this way.

### 8.2.6 Aliased arguments and component aliasing

Recall Rule 5.2, which states that no variable may appear more than once in a function argument list. This rule forbids MIR code such as the following:

\[
\text{return } \text{Pair}(a, a)
\]

Were we to analyse this code, we would obtain an unsound result:

\[
\{ \{$.0, a\}, \{$.1, a\} \}
\]

This misses the crucial edge between $.0 and $.1, which shows that the components of the pair are aliased to one another (and will be remembered even after \(a\) goes out of scope).

The unique arguments rule requires that the compiler must transform code such as the above into an equivalent form such as in Example 8.7.

The analysis of component\_alias is shown in Figure 8.5. The pre-existing edge \(\{a, t\}\)

\(^3\)Program code that does treat individual array elements specially would typically be better off being written using tuples — hence \text{pair\_const} is more sensible than \text{array\_const}. 
Example 8.7 Returning a pair whose components are aliased

```python
1: def component_alias():
2:     a = [1, 2, 3]
3:     t = a
4:     return Pair(a, t)
```

(a) Line 3  
(b) Line 4

![Diagram of internal sharing graph for component_alias after selected lines of analysis](image)

Figure 8.5: Internal sharing graph for component_alias after selected lines of analysis

is copied by the standard propagation rule to \{$.0, $.1\}, which persists after both \(a\) and \(t\) go out of scope. The uniqueinfo for component_alias is:

\[
\text{component_alias}_\alpha() \$ = \{$.0, $.1\}
\]

8.2.7 Types with alternative constructors

In the previous section, we saw how the analysis deals with structure types such as Pair. Here, we shall deal with algebraic types with more than one alternative. Consider the Either type, which stores either a value of type \(a\) or of type \(b\):

```python
type Either(a, b):
    Left(a)
    Right(b)
```

In MIR, as described in Section 5.2.6, fields are explicitly addressed with their constructor name (the fields of the Left and Right constructors are referred to as Left/0 and Right/0, respectively). In the abstract semantics, instead of explicitly mentioning the constructor name, we flatten algebraic types, giving all fields a unique index. In this case, we would give the field Left/0 the index 0 and the field Right/0 the index 1.\(^4\)

\(^4\)This is a purely pragmatic choice; our abstract field selectors have the same information content as the explicitly named concrete field selectors. The decision to use flattened field indices was made to reduce the clutter in sharing graph diagrams and uniqueinfo descriptions, which tend to mention field names regularly.
Having the abstract interpretation simultaneously track every field of every constructor allows us to precisely analyse a program such as Example 8.8.

**Example 8.8** Function that builds an Either with two possibilities

```python
def build_either(c, x, y):
    if c:
        r = Left(x)
    else:
        r = Right(y)
    return r
```

The analysis of build_either (shown in Figure 8.6) shows that at the end of the `if-then` statement, the graphs are joined, such that the graph reads “`x` may alias r.Left/0 and `y` may alias r.Right/0.” The uniqueinfo of build_either is:

\[
\text{build\_either}_\alpha(c, x, y) \; \{\{.0, x\}, \{.1, y\}\}
\]

In the analysis, Pair and Either are nearly the same, and for most of our analysis, we will treat these two types equivalently. It makes sense that the abstract type Either should be equivalent to Pair: while a concrete Either only has a left or right value (but not both), it may be necessary, as in Example 8.8, to assume that an abstract Either contains both a left and right value, and so this type is represented as a pair of abstract values.

### 8.2.8 Recursive analysis

As described in Chapter 6, recursive code requires special consideration, because when analysing a recursive function, we do not know the aliasing behaviour of all of the functions it calls. We must apply fixed-point iteration to repeatedly analyse a recursive strongly connected component of functions until the uniqueinfo stabilises.
In addition, we must ensure that the set of names is finite through a process called type folding. Imagine that \( x \) names a linked list, with element 0 being the head, and element 1 being the tail of the list. If we let the name \( x.0 \) name the head and \( x.1 \) name the tail, then \( x.1.0 \) would name the head of the tail and \( x.1.1 \) would name the tail of the tail, and so on. At compile time, we have no idea how long the list may eventually get, so we would have to have a name for all potential list elements — \( x.1.1.1.\ldots \), and so on — resulting in an infinite number of names. A recursive function dealing with lists is likely to continue growing the names it addresses on each iteration, meaning it will never reach a fixed point and the analysis will not terminate! This is a problem for any recursive type. We must ensure that the set of available names, for any given type, is finite.

To solve this problem, we type fold any name with the same type as one of its ancestors into that ancestor, so all of the memory cells in a path along the type tree that have the same type will share a name. For example, if \( x \) has type \( \text{List}(a) \), then \( x.0 \) has type \( a \) and \( x.1 \) has type \( \text{List}(a) \). As \( x.1 \) has the same type as \( x \), it is folded up, and therefore, the name \( x \) names both the list, and the tail of the list. Applied recursively, we see that \( x.1.0 \) (the second item in the list) is now named by \( x.0 \), and \( x.1.1 \) is now named by \( x \). Indeed, there are now just two names for the entire list: \( x \), which names each cons cell in the list (the “spine”) of the list, and \( x.0 \), which names all of the items in the list. This is analogous to arrays, in which \( a \) might name the array container, while \( a[\ ] \) names all of the items in the array. The names \( x.1 \), \( x.1.1 \), and so on, are illegal names, which should never appear.

**Example 8.9** The append function

```python
1: def append(x, y):
2:     switch x:
3:         case Nil:
4:             r = y
5:         case Cons:
6:             h = x.Cons/0
7:             t = x.Cons/1
8:             t1 = append(t, y)
9:             r = Cons(h, t1)
10:     return r
```

With this in mind, we analyse `append`, given in Example 8.9. On the Nil branch, we trivially obtain the sharing graph \( \{\{r, y\}, \{r.0, y.0\}\} \) (not shown visually). Figure 8.7 shows the analysis progressing through the Cons branch.

On Line 7, we encounter aliasing between \( x.1 \) (\( x \)'s tail) and \( t \). However, as discussed above, there is no such name \( x.1 \) due to type folding. Therefore, we fold up any references to \( x.1 \) into \( x \) itself, and end up with a sharing graph edge between \( x \) and \( t \). This is a
conservative approximation of the truth: it says “the spine of \( x \) may alias all or part of the spine of \( t \),” which is true, despite the fact that the top cell of \( x \) is never aliased to \( t \).

On Line 8 is the recursive call to \texttt{append}. Because we know nothing about \texttt{append}, we assume its \texttt{uniqueinfo} is \perp (no aliasing). Thus, the sharing graph shows that the result, \( t_1 \), is unique. This is clearly false, but it will be rectified on subsequent iterations. After Line 9, we know that some of the elements of \( r \) may be aliased to some of the elements of \( x \), but are missing the relationship between \( r \) and \( y \). Finally, we join the two branches, and obtain the following \texttt{uniqueinfo}:

\[
\texttt{append}_a (x, y) \ = \ \{ \{$.0, x.0\}, \{$, y\}, \{$.0, y.0\} \}
\]

which happens to be correct, but we have no way of knowing this, and furthermore, some of the intermediate sharing graphs were unsound. Therefore, we repeat the analysis, using the new \texttt{uniqueinfo} for \texttt{append} on the recursive call.

Figure 8.8 shows the second iteration analysis of the Cons branch. This time, we correctly acknowledge the edge between \( t_1 \) and \( y \) (and also pick up an unnecessary edge
between \( t_{1.0} \) and \( h \). On Line 9, we correctly infer the edge between \( r \) and \( y \). Our final sharing graph is identical to the previous iteration, which means that we have reached a fixed point and obtained the correct result. Therefore, the uniqueinfo for \( \text{append} \) is:

\[
\text{append}_x(x, y) = \{ \{$.0, x.0\}, \{$.0, y.0\}\}
\]

This tells us that some of the spine of the result may be aliased to some of the spine of \( y \) (we over-approximate and assume that the entire result is aliased to \( y \), which may be true if \( x \) is empty), and that some of the elements of the result may be aliased to some of the elements of \( x \). Note that the result does not alias \( x \) itself, because \( \text{append} \) creates a copy of \( x \)'s spine. This is the most precise result possible, given our choice of abstract domain.

We have avoided discussing whether \( \text{append} \) would be able to reuse the input memory cells here. That problem will be addressed in Chapter 10.

### 8.2.9 Static data

All programming languages have *static data* — objects that have the lifetime of the entire program. Static data is often known as *global data* because it is commonly stored in global variables. In Mars and MIR, static data is stored in computable global constants (CGCs). Our aliasing analysis needs a special case to handle static data: a global variable should never be destructively updated, even if there is only one local variable pointing at it.

**Example 8.10** Aliasing with a computable global constant

```python
1: def my_array:
2:     return [1, 2, 3]
3: def my_array_add(x):
4:     a = my_array
5:     return array_add(a, x)
```

Consider Example 8.10. Any call to \( \text{my_array} \) will result in an array value that can never be destructively updated — doing so would ruin the result of any subsequent call to \( \text{my_array} \). But if we consider the analysis of \( \text{my_array_add} \), after Line 4, the variable \( a \) is not aliased to any other variable, and hence eligible for an unsound destructive update on Line 5.

The solution is to introduce a special name, “@,” pronounced “static data.” This name is automatically aliased to any variable assigned the result of a CGC, and this has a twofold effect: the variable will be considered forever aliased, and it will “infect” any other nodes it is copied into. A variable aliased to @ can be thought of as having a special mark that prevents it from being destructively updated. Figure 8.9 shows this process.
After Line 4, $a$ is aliased to $\oplus$, and so it is not destructively updated. The call to \texttt{array\_add} creates a fresh non-static array, which is eligible for destructive update.

### 8.2.10 Closure data

In this first-order analysis, we are not particularly concerned with precisely maintaining aliasing information for code that treats functions as first-class values. However, because we are analysing a higher-order language, we do at least need to handle such situations soundly — we must not underestimate the aliasing.

**Example 8.11** Aliasing via the result of a closure template

```python
1: \textbf{def} const\_1\{x :: a\}(y :: b):
2: \hphantom{1:} return x
3: \textbf{def} modify\_a(z):
4: \hphantom{3:} a = [1, 2, 3]
5: \hphantom{3:} f = const\_1\{a\}
6: \hphantom{3:} b = f(0)
7: \hphantom{3:} c = array\_add(a, z)
8: \hphantom{3:} \textbf{return} Pair(c, b)
```

In Example 8.11, the function \texttt{modify\_a} creates an object $a$, then creates a function $f$, which will return $a$ when called. Since the result of $f$ is assigned to $b$, the variables $a$ and $b$ are aliased, and neither may be destructively updated. We must ensure that our rules can identify the aliasing between $a$ and $b$. A simple approach would be to alias $f$ to $a$ on Line 5, then alias $b$ to $f$ on Line 6 (which would automatically propagate an edge between $a$ and $b$). However, $a$ and $f$ are type-incompatible, so this is not allowed under the typing rules.
8.3 Abstract memory denotational semantics

We now begin to formally describe the abstract interpretation. First, we describe our abstract domain and the abstraction and concretisation functions for converting between our concrete memory domain of Section 5.7 and the abstract domain. We then provide a formal denotational semantics for the abstraction of the concrete memory semantics.
8.3. ABSTRACT MEMORY DENOTATIONAL SEMANTICS

8.3.1 Abstract cell names

A memory cell is a single memory location in the concrete memory semantics. In the abstract semantics, a name may refer to (or name) one or more memory cells. For example, in the concrete semantics, there may be an array $a$ with three elements, $a[0]$, $a[1]$ and $a[2]$. We say that the abstract name $a[]$ names the concrete cell $a[0]$, as well as $a[1]$ and $a[2]$. A simpler example is shown in Figure 8.11a, with a pair $p$. In this example, there are three names, one per concrete cell: $p$, $p.0$ and $p.1$, each naming their respective cells.\(^5\)

For any given abstract interpretation, there is a finite set of names (though this set may be arbitrarily large, depending on the variables and types in the program). Each name may refer to an infinite set of potential memory cells that may be created at runtime (for example, one name can refer to all elements of an array or linked list, which may have an arbitrarily large size at runtime). Furthermore, a memory cell may be named by several names (for example, the names $x$ and $y$ may refer to the same memory cell at runtime; this would mean that $x$ and $y$ are aliased). This may occur in combination. For example, if an array $a$ contains an element $a[2]$ that is aliased to $i$, then $a[]$ names several memory cells, one of which is also named by $i$ (see Figure 8.11b). In this situation, we say that $a[]$ and $i$ are aliased. Note that in both situations, the aliasing is not definite: $x$ and $y$ may only be aliased on some code paths, and not others; $i$ is only aliased to some elements of $a$, and never others. At the abstract level, “aliased” implicitly means may alias.

---

\(^5\)The boxes with numbers can be considered to be of a type with constructor Box(Num), artificially boxing the numeric values for the purpose of demonstration.
Name is either a variable or a component of a variable. The name \( x.n \) refers to the \( n \)th field of the cells named \( x \). The name \( x[] \) ("\( x \)'s array-contents") refers to all of the array elements of the cells named \( x \). The name \( x() \) ("\( x \)'s closure-data") refers to all of the memory cells, and their children, that were passed to the closure template when the closure \( x \) was created. The name \( @ \) refers to the immutable static (global) data store.

In addition, we must ensure that the set of names is finite through a process called type folding, due to Bruynooghe [11]. This process was outlined informally in Section 8.2.8. Recall that, in order to ensure a finite set of names, we fold any name with the same type as one of its ancestors into that ancestor. Formally, each name of type \( \tau \) implicitly refers to all descendant memory cells with type \( \tau \). Figure 8.11c shows an example of a concrete list, and the type-folded names that refer to each element of the list. As \( x \) has type \( \text{List}(\ldots) \), all child nodes with type \( \text{List}(\ldots) \) are named by \( x \) and do not have their own individual names. This means that \( x.0 \) refers not only to the head of the list, but also the head of all cells named by \( x \) — the set of all list elements.

This means that the set of names is finite for any given execution unit, as the set of variables and the set of types is finite (the size of the set \( \text{Name} \) is the sum of the transitive number of fields in the type of every variable in the execution unit).

The function \( \text{names} \) takes a concrete state (as defined in Section 5.7.1) and name, and produces the set of memory cells the name refers to. Names that refer to unboxed values (such as numbers) map to the empty set. The function \( \text{closure} \) takes a value and produces the memory location of the value and all of its descendants, as per the above rule. The function \( \text{typeclosure} \) takes a value and finds the subset of the closure of the value whose elements have the same type as the value. The function \( \text{type} \) gets the type of a value.\(^6\)

\[\begin{align*}
\text{names} & : \text{State} \to \text{Name} \to \wp(\text{Ref}) \\
\text{closure} & : \text{Heap} \to \text{Val} \to \wp(\text{Ref}) \\
\text{typeclosure} & : \text{Heap} \to \text{Val} \to \wp(\text{Ref}) \\
\text{type} & : \text{Val} \to \text{Type}
\end{align*}\]

\(^6\)In this chapter, we are not concerned with the form of the type, just that each value has a specific type which determines the type of each of the value's fields.
8.3. ABSTRACT MEMORY DENOTATIONAL SEMANTICS

\[
\text{names} \ (\eta, \sigma) \ [v] = \ \text{typeclosure} \ \eta \ (\sigma v)
\]
\[
\text{names} \ (\eta, \sigma) \ [x.\tilde{a}] = \bigcup \{\text{typeclosure} \ \eta \ r_i : r \in \text{names} \ (\eta, \sigma) \ [x]\}
\]
\[
\text{names} \ (\eta, \sigma) \ [x[]] = \bigcup \{\text{typeclosure} \ \eta \ r_i : r \in \text{names} \ (\eta, \sigma) \ [x] \land 0 \leq i < |r|\}
\]
\[
\text{names} \ (\eta, \sigma) \ [x()] = \bigcup \{\text{closure} \ c_i : r \in \text{names} \ (\eta, \sigma) \ [x] \land (c, _) = \eta \ r \land 0 \leq i < |c|\}
\]
\[
\text{names} \ (\eta, \sigma) \ [@] = \{\text{cgcref} \ n : n \in \text{CGCName}\}
\]
\[
\text{closure} \ \eta \ v = \begin{cases} 
\emptyset, & \text{if} \ v \in \{\bot\} \cup Q \\
\{r\} \cup r', & \text{otherwise}
\end{cases}
\]
where
\[
r = \begin{cases} 
\text{let} \ (data, ct) = v \ \text{in} \ data, & \text{if} \ v \in \text{Closure} \\
\end{cases}
\]
\[
r' = \bigcup \{\text{closure} \ \eta \ (\eta \ r_i) : 0 \leq i < |\eta \ r|\}
\]
\[
\text{typeclosure} \ \eta \ v = \{x : x \in \text{closure} \ \eta \ v \land \text{type} \ x = \text{type} \ v\}
\]

8.3.2 The abstract domain

The abstract domain is quite different from the one given in Chapter 6, in that it does not have a one-to-one mapping between concrete and abstract domain types. For example, there is no abstract equivalent of the Ref or Val concrete domain, because we do not explicitly assign an abstract value to each variable. Our domain is relational: we are not interested in the properties of individual variables, but in the relationship between variables, so we instead consider the abstract state of all variables together.

In the concrete memory semantics (Section 5.7), the domain State is the set of all possible computation states at any given program point. We introduce the abstract domain Sharing, which formally represents the sharing graph used in the examples above. For the first-order analysis, Sharing is the entire abstract state, also known as State_\alpha.

Sharing represents an undirected graph of Names. An edge between two Names \(x\) and \(y\) indicates that one or more memory cells referred to by \(x\) may be the same as one or more of the memory cells referred to by \(y\) (they may alias). If \(x\) and \(y\) do not have an edge, then the set of memory cells named by \(x\) is disjoint from the set of cells named by \(y\) (they do not alias).

\[
\text{State}_\alpha = \text{Sharing}
\]
\[
\text{Sharing} = \wp(\text{Name} \otimes \text{Name})
\]

Recall that \(X \otimes Y\) is the unordered Cartesian product of \(X\) and \(Y\).
The domain Sharing forms a simple power-set lattice, with the following properties:

\[
\begin{align*}
X \subseteq Y & \equiv X \subseteq Y \\
X \supseteq Y & \equiv X \supseteq Y \\
\sqcup X & \equiv \cup X \\
\sqcap X & \equiv \cap X \\
\bot & \equiv \emptyset \\
\top & \equiv \{ \{x, y\} : x, y \in \text{Name} \land x \neq y \}
\end{align*}
\]

For a given execution unit, the set Name is constrained to only include names of variables appearing in the unit, with finite field qualifiers based on the variables’ types. With only the constrained set of names allowed, the set Sharing is also finite, with \( \top \) being the set of edges between all legal names in the execution unit. Figure 8.12 shows a complete Sharing lattice for an execution unit consisting of the names \( a, b \) and \( c \).

![Hasse diagram for the domain Sharing with Name = \{a, b, c\}]

The maximum number of edges in the graph is equivalent to the handshake problem [36]. Let \( n = |\text{Name}| \), the number of names in a given execution unit. The maximum number of edges in the graph is \( |\top| = \binom{n}{2} = \frac{n(n-1)}{2} \). The total number of abstract states (the total size of State\( _a \)) is \( 2^{|\top|} \). The height of the lattice is \( |\top| + 1 \) — the maximum number of iterations required to reach a fixed point (given that the result of each iteration will be greater than the previous). Therefore, the worst case time for the analysis of a given execution unit is \( O(n^2) \) iterations.

We have only considered a pair-sharing domain, where we explicitly record pairs of variables that may alias. Some of the worst exponential cases could be mitigated by instead using a set-sharing domain:

\[
\text{SetSharing} = \wp\left(\wp(\text{Name})\right)
\]
8.3. ABSTRACT MEMORY DENOTATIONAL SEMANTICS

With this domain, we can more efficiently encode dense sharing graphs (e.g., \( \{\{x, y, z\}\} \) instead of \( \{\{x, y\}, \{x, z\}, \{y, z\}\}\)). We have not explored this approach further, as it invites significant complexity to the implementation. Note that in logic program analysis, such a domain is commonplace, as it is more precise than pair-sharing for encoding information such as groundness and whether three or more names may share a common memory cell, but this extra information is redundant when all we care about is knowing whether two given variables are aliased [6].

The other domains are as follows:

\[
\begin{align*}
\text{Func}_\alpha &= \text{Name}_\bot^* \rightarrow \text{Name} \rightarrow \text{Sharing} \\
\text{CT}_\alpha &= \text{Name}_\bot^* \rightarrow \text{Name}_\bot^* \rightarrow \text{Name} \rightarrow \text{Sharing} \\
\text{CGC}_\alpha &= \text{Name} \rightarrow \text{Sharing} \\
\text{Den}_\alpha &= (\text{CGCName} \cup \text{FuncName} \cup \text{CTName}) \rightarrow (\text{CGC}_\alpha \cup \text{Func}_\alpha \cup \text{CT}_\alpha)
\end{align*}
\]

A function is represented abstractly as a \( \text{Func}_\alpha = \text{Name}_\bot^* \rightarrow \text{Name} \rightarrow \text{State}_\alpha \). This is what we call a “uniqueinfo;” it contains all of the information about how that function aliases its arguments and result. This meta-function takes the (possibly empty) name of each actual parameter, and the name of the location to store the result in, and returns a new sharing graph showing just the new aliasing introduced between the input and result (and parts thereof), as well as between parts of the result. A requirement of all functions is that the resulting \( \text{State}_\alpha \) features the result name (e.g., $) in every edge (it cannot introduce aliasing between two nodes which are not the result or part thereof, since Mars functions cannot modify their arguments).

Similarly, the uniqueinfo of a closure template is a \( \text{CT}_\alpha \), which takes the name of each free variable, the name of each actual parameter, and the name of the result location, and models the closure template’s behaviour in the same way as a function. Finally, the uniqueinfo of a computable global constant is a \( \text{CGC}_\alpha \), which takes the name of the result and produces the sharing graph, modelling the aliasing between the components of the result.\(^7\)

8.3.3 Abstraction and concretisation

We define functions that translate between abstract and concrete states. The state abstraction function, \( \alpha : \emptyset(\text{State}) \rightarrow \text{State}_\alpha \), converts a set of concrete states into an abstract one. The state concretisation function, \( \gamma : \text{State}_\alpha \rightarrow \emptyset(\text{State}) \), converts an abstract state into a set of concrete ones.

\(^7\)This is not strictly necessary, since CGCs always share with static data, but is done for completeness.
First, we define \( \beta : \text{State} \to \text{State}_a \), which converts a single concrete state into an abstract state.

\[
\beta : \text{State} \to \text{State}_a
\]

\[
\beta s = \{ \{x, y\} : (\text{names } s x \cap \text{names } s y) \neq \emptyset \}
\]

The abstraction of a concrete state \( s \) includes an edge between any two names \( x \) and \( y \) if there is at least one memory cell in \( s \) named by both \( x \) and \( y \).

The \( \alpha \) and \( \gamma \) functions are expressed in terms of \( \beta \) in the usual way:

\[
\alpha : \wp(\text{State}) \to \text{State}_a
\]

\[
\gamma : \text{State}_a \to \wp(\text{State})
\]

\[
\alpha S = \bigcup_{s \in S} \beta s
\]

\[
\gamma g = \{ s : \beta s \sqsubseteq g \}
\]

The abstraction of a set of concrete states \( S \) includes an edge between any pair of names \( \{x, y\} \) if there is at least one state in \( S \) with at least one memory cell named by both \( x \) and \( y \). The concretisation of an abstract sharing graph \( g \) includes any conceivable concrete state with no cell named by both \( x \) and \( y \) for any pair of names \( \{x, y\} \) that do not have an edge in \( g \).

### 8.3.4 The alternating closure operator

The alternating closure operator (\( \bowtie \)) is used to combine two sharing graphs that appear in sequence (it is the abstract sequencing operator). Whereas the lub (\( \sqcup \)) operator combines sharing graphs that are true in alternative code paths, \( \bowtie \) combines sharing graphs that are true in sequence on the same code path.

The operation is due to Mulkers [74], but we use the definition from Mazur [65]:

\[
\bowtie : \wp(\text{State}_a \otimes \text{State}_a) \to \wp(\text{State}_a) \to \wp(\text{State}_a)
\]

\[
X \bowtie Y = \left\{ \begin{array}{l}
\exists a_0, \ldots, a_n \in S, \\
\{\{a_0, a_1\}, \{a_1, a_2\}, \ldots, \{a_{n-1}, a_n\}\} \subseteq X \cup Y \wedge \\
\text{type compatible}(\text{type } a_0, \text{type } a_n) \wedge \\
n \geq 1 \wedge \forall i \in \{1, \ldots, n-1\}, \\
\{a_{i-1}, a_i\} \in X \implies \{a_i, a_{i+1}\} \in Y \wedge \\
\{a_{i-1}, a_i\} \in Y \implies \{a_i, a_{i+1}\} \in X
\end{array} \right\}
\]
This operation is called “altclos” in [65], and appears here with notational changes, as well as with the addition of the “type compatible” clause. The \( \bowtie \) operator can be described as follows: given two undirected graphs \( X \) and \( Y \), create a pair between any two nodes \( a_0 \) and \( a_n \) if it is possible to form a path between them using alternating edges from \( X \) and \( Y \) (for example, with \( \{a_0, a_1\} \) in \( X \), \( \{a_1, a_2\} \) in \( Y \), \( \{a_2, a_3\} \) in \( X \), and so on). This is a superset of \( X \cup Y \). It is commutative and associative.

This is used as the abstract sequencing operator for combining sharing graphs between two statements. If \( X \) is the sharing resulting from statement \( s_1 \) and \( Y \) is the sharing resulting from statement \( s_2 \), then \( A \bowtie Z B \) is the sharing resulting from the statement sequence \( s_1 ; s_2 \). The alternating nature of the algorithm is due to the reasoning that if \( a \) and \( b \) share the same object (in \( X \)), and \( b \) and \( c \) share the same object (in \( Y \)), then \( a \) and \( c \) may also share that same object, so a new edge \( \{a, c\} \) is required. Conversely, if edges \( \{a, b\} \) and \( \{b, c\} \) are both in \( X \), there is no need for an edge \( \{a, c\} \), since whatever process constructed \( X \) has already determined that they definitely do not alias.

**Example 8.12** Propagating an alias across three variables

1: \texttt{def id}(x):
2: \hspace{1em} \texttt{y = x}
3: \hspace{1em} \texttt{z = y}
4: \hspace{1em} \texttt{return z}

The most basic example of this operator’s necessity is propagating a single alias across three variables. Consider Example 8.12. The analysis of Line 2 computes the sharing graph \( \{\{x, y\}\} \). The analysis of Line 3 computes the additional alias \( \{\{y, z\}\} \). A simple union of these graphs would result in \( \{\{x, y\}, \{y, z\}\} \), which is missing the fact that \( x \) is aliased to \( z \). To properly compute the abstract state after Line 3, we must find \( \{\{x, y\}\} \bowtie \{\{y, z\}\} \). This finds a path \( \{x, y\} - \{y, z\} \), giving an additional edge \( \{x, z\} \):

\[
\{\{x, y\}\} \bowtie \{\{y, z\}\} = \{\{x, y\}, \{x, z\}, \{y, z\}\}
\]

The \( \bowtie \) operator is much more precise than simply having aliasing be fully transitive, because it preserves the notion that a variable may alias one (but not both) of two other variables. Firstly, consider the case where \( a \) is aliased to \( b \), and a statement causes \( b \) to alias either \( c \) or \( d \) but not both. We find paths \( \{a, b\} - \{b, c\} \) (giving \( \{a, c\} \)), and \( \{a, b\} - \{b, d\} \) (giving \( \{a, d\} \)), but importantly, do not find any alternating path from \( c \) to \( d \), preserving the knowledge that \( c \) and \( d \) are not aliased:

\[
\{\{a, b\}\} \bowtie \{\{b, c\}, \{b, d\}\} = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}\}
\]
Secondly, consider the case where $a$ is aliased to $b$, and a statement causes $c$ to alias either $b$ or $d$ but not both. We find a path $\{a, b\} - \{b, c\}$ (giving $\{a, c\}$), but there is no path from $a$ to $d$ or $b$ to $d$:

$$\{a, b\} \bowtie \{b, c, c, d\} = \{a, b, a, c, b, c, c, d\}$$

The predicate $\text{type}_{\text{compatible}}(S, T)$ succeeds if types $S$ and $T$ are either the same type, or one of them is typeless (i.e., the type of closure-data names and the special name @). The “$\text{type}_{\text{compatible}}$” clause in $\bowtie$ removes edges between any two well-typed names if their types are incompatible; it is always safe to remove such edges because they must not be aliased in the concrete semantics. Such edges can arise by propagating edges between two well-typed names with a typeless name; for example:

$$\{\{f(), x\}\} \bowtie \{\{f(), y\}\} = \{\{f(), x\}, \{f(), x\}, \{f(), y\}, \{x, y\}\}$$

The edge $\{x, y\}$ is omitted if $x$ and $y$ have different types.

Note that the result of $A \bowtie B$ is always a superset of both $A$ and $B$, so $\bowtie$ can be thought of as a “union with additional aliasing propagation.” The associativity of $\bowtie$ is important because it means that the meaning of a sequence of statements does not change if it is abstracted into a block or a separate procedure.

### 8.3.5 Helper functions

This section introduces a number of functions used to define the abstract semantics.

The function $\text{firstfield} : \text{CtorName} \rightarrow \mathcal{Z}$ determines the index of the first field of a given constructor. For the first constructor of a given type, this is always 0. For example, in the Either type, $\text{firstfield Left} = 0$ while $\text{firstfield Right} = 1$. This is used to map concrete $\text{constructor/field selectors}$ onto abstract flattened $\text{field selectors}$, introduced in Section 8.2.7.

$\text{completion } \eta$ produces a new sharing graph which is $\eta$ with all aliases explicitly having their corresponding descendants aliased. If a variable with a known type is aliased to a typeless node (e.g., $\{v(), x\}$), all descendants of $x$ will be aliased to $v()$.

$$\text{completion } \eta : \text{Sharing} \rightarrow \text{Sharing}$$

$$\text{completion } \eta = \left\{ \{f, x, f, y\} : \{x, y\} \in \eta \land f \in \left( \text{descendants (type x)} \cup \right) \left( \text{descendants (type y)} \right) \right\}$$
8.3. ABSTRACT MEMORY DENOTATIONAL SEMANTICS

\[
\begin{align*}
\text{descendants} &: \text{Type} \rightarrow \wp(\text{Name} \rightarrow \text{Name}) \\
\text{descendants Num} &= \{\lambda x. x\} \\
\text{descendants Array}(a) &= \{\lambda x. x\} \cup \{\lambda x. f(x[i]) : f \in \text{descendants } a\} \\
\text{descendants } (a_1, \ldots, a_n) \rightarrow b &= \{\lambda x. x, \lambda x. x()\} \\
\text{descendants } (a_0, \ldots, a_{n-1}) &= \{\lambda x. x\} \cup \{\lambda x. f(x.i) : i \in \{0, \ldots, n-1\} \land f \in \text{descendants } a_i\} \\
\text{descendants } t &= \{\lambda x. x\}, \text{ where } t \text{ is a type variable}
\end{align*}
\]

For example, \(\text{descendants } \text{Pair} = \{\lambda x. x, \lambda x. x.0, \lambda x. x.1\}\). This creates a set of functions which, if applied to a name, produce all descendants of that name. Hence, \(\{f[s] : f \in \text{descendants } \text{Pair}\} = \{[s], [s.0], [s.1]\}\).

We can now take a more formal look at Example 8.5 (pair_const). After Line 2, we have the sharing graph \(\{\{p.0, x\}, \{p.1, y\}\}\) (Figure 8.3b). The statement \([q = p]\) provisionally generates the sharing graph \(\{\{p, q\}\}\), but we apply completion to generate all of the aliases between corresponding children:

\[
\text{completion } \{\{p, q\}\} = \{\{f, p, f, q\} : f \in \text{descendants } \text{Pair}\} = \{\{p, q\}, \{p.0, q.0\}, \{p.1, q.1\}\}
\]

This is the new sharing generated by Line 3. We now import the new sharing into the existing sharing graph, using \(\bowtie\) to propagate the sharing between \(x/y\) and the children of \(p\) to the children of \(q\):

\[
\begin{align*}
\{\{p.0, x\}, \{p.1, y\}\} &\bowtie \{\{p, q\}, \{p.0, q.0\}, \{p.1, q.1\}\} \\
&= \{\{p, q\}, \{p.0, q.0\}, \{p.0, x\}, \{p.1, q.1\}, \{p.1, y\}, \{q.0, x\}, \{q.1, y\}\}
\end{align*}
\]

Lastly, because \(p\) is dead after Line 3, we remove all edges involving \(p\) or its descendants. Therefore, the sharing graph after Line 3 is \(\{\{q.0, x\}, \{q.1, y\}\}\) (Figure 8.3c).

Because the atom semantics \(A_a\) may return \(\perp\) for atoms with unboxed values, we tend to end up with sharing pairs like \(\{v, \perp\}\). We use a function called \(s\) to clean up any Sharing that might contain a \(\perp\) in its pairs, producing a true sharing graph:

\[
\begin{align*}
\text{s} &: \wp(\text{Name}_\perp \otimes \text{Name}_\perp) \rightarrow \text{Sharing} \\
\text{s }\eta &= \{p : p \in \eta \land \perp \notin p\}
\end{align*}
\]

\(\text{assigned} : \text{Stmt} \rightarrow \wp(\text{Name})\) gets the set of names of all variables on the left-hand side of any assignment statements within a given statement.

\(\text{remove} : \wp(\text{Name}) \rightarrow \text{Sharing} \rightarrow \text{Sharing}\) removes any edges from a Sharing which feature members or descendants of the given name set on either end.
CHAPTER 8. FIRST-ORDER UNIQUENESS INFERENCE

\[ \text{keep} : \varphi(\text{Name}) \to \text{Sharing} \to \text{Sharing} \text{ removes any edges from a Sharing which do not feature members or descendants of the given name set on both ends.} \]

8.3.6 Abstract semantics

This denotational semantics explicitly tracks memory usage over the abstract domain. Operations approximate the real memory usage in provably finite time.

Recall from Definition 6.4 and Section 6.4 that an abstract interpretation must be consistent with the concrete semantics, by satisfying the following rule for each abstract operation \( N_\alpha \) relative to its corresponding concrete operation \( N \):

\[ \forall c. \beta (N c) \subseteq N_\alpha (\beta c) \]

This assumes that \( N_\alpha \) is monotonic (i.e., \( x \subseteq y \implies N_\alpha x \subseteq N_\alpha y \)). This semantics has been designed to give results as precise as possible while remaining consistent.

\[ \Theta_\alpha : \text{Den}_\alpha \]

\[ A_\alpha : \text{Atom} \to \text{Name}_\perp \]

\[ E_\alpha : \text{Var} \to \text{Expr} \to \text{Den}_\alpha \to \text{Sharing} \]

\[ S_\alpha : \text{Stmt} \to \text{Den}_\alpha \to \text{State}_\alpha \]

\[ F_\alpha : \text{FuncDef} \to \text{Den}_\alpha \to \text{Den}_\alpha \]

\[ P_\alpha : \text{Program} \to \text{Den}_\alpha \]

\[ \Theta_\alpha = \{ f \mapsto \text{abstract semantics of } f : f \text{ is the name of a primitive function} \} \]
8.3. ABSTRACT MEMORY DENOTATIONAL SEMANTICS

\[ \mathcal{A}_\alpha[n] = \bot, \quad \text{if } n \in \text{Num} \]
\[ \mathcal{A}_\alpha[v] = [v], \quad \text{if } v \in \text{Var} \]

\[ \mathcal{E}_\alpha[a \in \text{Atom}] \rho = s \{ [[[v]], \mathcal{A}_\alpha[a]] \} \]
\[ \mathcal{E}_\alpha[c \in \text{CGCName}] \rho = (\rho[c]) v \cup s \{ [[[v]], \oplus] \} \]
\[ \mathcal{E}_\alpha[v \cdot c / n] \rho = \{ [[[v]], [s].(n + \text{firstfield } c)] \} \]
\[ \mathcal{E}_\alpha[[a_1, \ldots, a_n]] \rho = s \{ [[[v]], \mathcal{A}_\alpha[a_1]], \ldots, [[[v]], \mathcal{A}_\alpha[a_n]] \} \]
\[ \mathcal{E}_\alpha[c(a_1, \ldots, a_n) \ hint] \rho = s \{ [[[v]], (\text{firstfield } c), \mathcal{A}_\alpha[a_1]], \ldots, [[[v]], (\text{firstfield } c + n - 1), \mathcal{A}_\alpha[a_n]] \} \]
\[ \mathcal{E}_\alpha[v \in \text{Var}(a_1, \ldots, a_n)] \rho = \{ [[[v]], \oplus], [[[v]], \mathcal{A}_\alpha[f]()}, \ldots, [[[v]], \mathcal{A}_\alpha[a_n]] \} \]
\[ \mathcal{E}_\alpha[v \in \text{FuncName}(a_1, \ldots, a_n)] \rho = (\rho f) (\mathcal{A}_\alpha[a_1], \ldots, \mathcal{A}_\alpha[a_n]) [v] \]
\[ \mathcal{E}_\alpha[v \in \text{FuncName}(a_1, \ldots, a_n)] \rho = \{ [[[v]], \mathcal{A}_\alpha[a_1]], \ldots, [[[v]], \mathcal{A}_\alpha[a_n]] \} \]

\[ S_\alpha[c] \rho = \emptyset \]
\[ S_\alpha[s_1 \cdot s_2] \rho = S_\alpha[s_1] \rho \cdot S_\alpha[s_2] \rho \]
\[ S_\alpha[v = e] \rho = \text{completion } (\mathcal{E}_\alpha[e] \rho) \]
\[ S_\alpha[\text{if } a : \text{then } s_1 \ldots \text{else } s_2] \rho = S_\alpha[s_1] \rho \cup S_\alpha[s_2] \rho \]
\[ S_\alpha[\text{switch } v : (c_1 : s_1, \ldots, c_n : s_n)] \rho = S_\alpha[s_1] \rho \cup \cdots \cup S_\alpha[s_n] \rho \]
\[ S_\alpha[\text{while } a \ [u_1 = \phi(v_1, w_1), \ldots] : s] \rho = \text{lfp } \lambda \eta. \]
\[ \text{let } \eta_1 = \eta \cup \{ u_1, v_1, \ldots \} \gg S_\alpha[s] \rho \]
\[ \eta_2 = \text{remove } \{ u_1, \ldots \} \eta_1 \]
\[ \eta_3 = \eta_2 \gg \{ u_1, v_1, \ldots \} \]
\[ \text{in remove } \text{assigned } [s] \eta_3 \]

\[ F_\alpha[f \ s e] \rho = \{ f \mapsto \lambda r. \}
\[ \text{let } \eta = (S_\alpha[s] \rho) \gg (\text{completion } (\mathcal{E}_\alpha[e] \rho)) \]
\[ \text{in } \{ r \} \eta \}
\[ F_\alpha[f(a_1, \ldots, a_n) \ s e] \rho = \{ f \mapsto \lambda (x_1, \ldots, x_n) r. \}
\[ \text{let } \eta = s \{ a_1, x_1, \ldots, \} \gg \]
\[ (S_\alpha[s] \rho) \gg (\text{completion } (\mathcal{E}_\alpha[e] \rho)) \]
\[ \text{in keep } \{ r, x_1, \ldots, x_n \} \eta \}
\[ F_\alpha[f(v_1, \ldots, v_n) (a_1, \ldots, a_n) \ s e] \rho = \{ f \mapsto \lambda (w_1, \ldots, w_n) (x_1, \ldots, x_n) r. \}
\[ \text{let } \eta = s \{ v_1, w_1, \ldots, \} \gg \]
\[ (S_\alpha[s] \rho) \gg (\text{completion } (\mathcal{E}_\alpha[e] \rho)) \]
\[ \text{in keep } \{ r, x_1, \ldots, x_n, w_1, \ldots, w_n \} \eta \}

\[ P_\alpha[p] = \text{lfp } \bigcup_{f \in p} (F_\alpha f) \sqcup \lambda \rho. \Theta_\alpha \]
A brief explanation of each of these rules follows.

The atom semantics ($A_\alpha a$) denotes the name associated with a given atom $a$. Each atom is associated with zero or one names: number literal atoms have no name, so are denoted as $\bot$. Local variable atoms $v$ are denoted as the corresponding name $v$. Wherever the result of $A_\alpha$ is used, $s$ is applied to remove any edges involving $\bot$.

The expression semantics ($E_\alpha v e$) denotes the sharing pairs that may be introduced by evaluating expression $e$ and assigning the result to local variable $v$. This does not include the aliases between corresponding child nodes; that is taken care of in a later step. An atom expression $a$ creates a single edge between $v$ and the name of the atom $a$. A CGC reference expression $c$ creates an edge between $v$ and @ (the static data), and also any additional aliases produced by the CGC $c$. A field-reference expression $s.c/n$ creates a single edge between $v$ and the child node for field $c/n$ of $s$’s node. An array expression $[a_1, \ldots, a_n]$ creates an edge between $v[]$ and each of the element atom names $a_1$ through to $a_n$. A constructor call expression $c(a_1, \ldots, a_n)$ creates an edge between each field node of $v$ and the corresponding argument atom name. Calls to functions are handled differently depending on whether the function is a local variable or global function name. Because we know nothing about the aliasing they might cause, local function call expressions $f(a_1, \ldots, a_n)$ are given a “worst-case” treatment, with $v$ being aliased to @ (the static data), $f()$ (f’s closure data), and every argument $a_1$ through to $a_n$ — the maximum possible aliasing that could be caused by any pure function. Global function call expressions are handled precisely, looking up the abstract semantics (uniqueinfo) of the function via $\rho$, and applying the arguments and return value — this creates new edges as specified by the analysis of that function. Lastly, a closure template reification expression $f\{a_1, \ldots, a_n\}$ creates an edge between $v()$ and each of the argument atom names $a_1$ through to $a_n$.

The statement semantics ($S_\alpha s$) denotes the sharing pairs that may be introduced as a result of executing statement $s$. An empty statement $\epsilon$ gives the empty set. A statement sequence $s_1; s_2$ gives the alternating closure ($\Rightarrow$) of the sharing of the two statements $s_1$ and $s_2$, which propagates sharing between variables as required (described in detail in Section 8.3.4). An assignment statement $v = e$ computes the new sharing edges for the expression $e$ assigned to $v$ using the expression semantics, then takes the completion of these edges (which adds edges between corresponding child nodes of any edges in the graph, as defined in Section 8.3.5). The if and switch statements simply join together their nested statements. The while statement performs a fixed-point computation to capture the effects of repeated execution; the precise details are discussed in Section 8.3.7.

The procedure semantics ($F_\alpha$) denotes the abstraction of a single CGC, function or

---

8This part, $(\rho c) v$, is actually redundant, because the only aliases a CGC can create are between $v$ or part thereof and the static data, which is subsumed by the edge $\{[v], @\}$. It is left in for completeness.
8.3. ABSTRACT MEMORY DENOTATIONAL SEMANTICS

closure template (as a Den with a single item). The abstract semantics of a procedure is its uniqueinfo. This works the same way as described in Chapter 6, except that after computing the final sharing graph of the function, keep is applied, which projects away any edges of the sharing graph that do not feature the result or parameters.

The program semantics \((P_\alpha \rho)\) denotes the abstraction of all procedures in a program. It computes the least fixed point of the join of the semantics of each procedure, as described in Chapter 6.

### 8.3.7 The while loop semantics

The semantics of the \textbf{while} statement computes the sharing of its body until a fixed point is reached. It is significantly complicated by the need to avoid interference from sharing with a variable of the same name on a previous iteration. The semantics is repeated here:

\[
\begin{align*}
S_\alpha [\text{while } a \{ u_1 = \phi(v_1, w_1), \ldots \}: \{ s \}] \rho &= \text{lfp } \lambda \eta. \\
\text{let } \eta_1 &= \eta \cup \{ \{ u_1, v_1 \}, \ldots \} \triangleright (S_\alpha [s] \rho) \\
\eta_2 &= \text{remove } \{ u_1, \ldots \} \eta_1 \\
\eta_3 &= \eta_2 \triangleright \{ \{ u_1, w_1 \}, \ldots \} \\
\text{in } \text{remove } (\text{assigned } [s]) \eta_3
\end{align*}
\]

Recall that \textbf{while} loops have a special phi notation for specifying the variables that are updated in the body of the loop, in order to avoid assigning a variable that is already assigned. To do so would violate the reassignment rule (Rule 5.1), and cause major imprecision in the sharing analysis, because the old value of the variable would be unnecessarily aliased to the new value of the variable.

Let us assume we have a \textbf{while} statement with a single loop variable, \([ u = \phi(v, w)]\), meaning that on the first iteration, \(v\) will be copied into \(u\), the body is expected to assign \(w\), and after each iteration, \(w\) will be copied into \(u\). We must ensure that as we compute the sharing semantics in a fixed-point iteration, we never alias one iteration’s \(w\) with the previous iteration’s \(u\). Each iteration, we take the following steps:

1. Begin with \(\eta\), the sharing graph from the previous iteration, or \(\emptyset\) if this is the first.
2. Union \(\eta\) with \(\{ \{ u, v \} \}\), the initial variable assignment implicit in the phi notation.
3. Apply (with \(\triangleright\)) the sharing graph of the body. At this point, \(u\) represents the loop variable from the previous iteration, while \(w\) represents it from the current iteration.
   Note that \(w\) is not necessarily aliased to \(u\).
4. Remove \(u\) from the sharing graph. This is crucial to prevent its aliases from spilling
over to \( w \) in the next step.

5. Apply (with \( \Rightarrow \)) the edge \( \{u, w\} \). This will copy any aliases of \( w \) to the “new” \( u \). Now \( u \) represents the loop variable from the current iteration.

6. Remove any variable that was assigned in the body of the loop, including \( w \), from the sharing graph. This is crucial to prevent their aliases from interfering with the next iteration.

Note that Steps 5 and 6 effectively rename \( w \) to \( u \), but not before cleaning out any old aliases of \( u \) in Step 4. Therefore, we have cleanly reset the sharing graph for the next iteration; it is free to alias \( w \) as required without there being any aliases left over from the previous iteration.

It is very important to start with a clean \( w \) on each iteration, because \( \Rightarrow \) is not idempotent. Consider a loop body with sharing \( \{\{w, a\}, \{w, b\}\} \). If we did not rename \( w \) to \( u \), then on the second iteration, we would be forced to compute:

\[
\{\{w, a\}, \{w, b\}\} \Rightarrow \{\{w, a\}, \{w, b\}\} = \{\{w, a\}, \{w, b\}, \{a, b\}\}
\]

With our more complicated approach, we avoid the unnecessary edge \( \{a, b\} \).

Note that the abstract semantics of a \texttt{while} loop is equivalent to the abstract semantics of a corresponding recursive function: both use a fixed-point computation to compute the recursive semantics of a statement (although transforming a \texttt{while} loop into recursion requires that all values updated in the loop body be returned in a tuple).

8.3.8 Primitives

For reference, the abstract semantics of some of the built-in functions follows:

\[
\begin{align*}
\text{add}_a \ (x, y) & \quad r \quad = \quad \{\} \\
\text{eq:Num}_a \ (x, y) & \quad r \quad = \quad \{\} \\
\text{array}_a \ (\text{len}, v) & \quad r \quad = \quad \{\{r[], v\}\} \\
\text{array_length}_a \ a & \quad r \quad = \quad \{\} \\
\text{array_ref}_a \ (a, i) & \quad r \quad = \quad \{r, a[i]\}\} \\
\text{array_replace}_a \ (a, i, v) & \quad r \quad = \quad \{\{r[], a[i]\}, \{r[], v\}\} \\
\text{array_add}_a \ (a, v) & \quad r \quad = \quad \{\{r[], a\]}, \{r[], v\}\}
\end{align*}
\]
8.3.9 Condensation

This semantics is trivially condensing, per Definition 6.5. None of the abstract operations accept a Sharing as input, so clearly do not depend on the previous abstract state — in fact, the statement semantics $S$ is already in a condensing-transformed form (as described in Section 6.6). The alternating closure operator ($\otimes$) provides the abstract conjunction required in a condensing analysis. Hence, a context-insensitive analysis is easy: simply compute the resulting sharing graph of each function (in the order determined by Algorithm 6.2), and store it as the function’s condensation, or summary. When analysing a function call, the callee’s condensation is available, and may be incorporated into the existing sharing graph using $\otimes$.

8.4 Comparison to Mercury’s CTGC system

In Section 7.4.1, we gave an overview of the Mercury compile-time garbage collection system [65] by Mazur. Mercury’s CTGC is largely the same as the system described in this chapter. Because our system was developed independently for a different language, we use rather different terminology and notation; this section reconciles those differences.

Mazur uses the term data structure to refer to what we call a name. Mazur uses selectors that are parameterised by the constructor name; hence for the data type Either (with constructors Left and Right with one field each), Mazur would refer to the two fields of a variable $X$ as $X^{\text{Left},1}$ and $X^{\text{Right},1}$, whereas we would flatten them into a single sequence of fields, $x.0$ and $x.1$, respectively. This was chosen for a more succinct notation, as well as a more natural visual representation when we draw sharing graphs.

Like Mazur, our abstract interpretation is performed over a sharing set environment (we call it a sharing graph to emphasise that we view it as a collection of edges between vertices). Most of Mazur’s operations have analogues in our analysis.

Mazur’s termshift operation takes a sharing set and updates it to make all of the implicit sharing between corresponding descendants of a sharing pair explicit; this is our completion operation. Finally, Mazur’s altclos (alternating closure) operation is our $\otimes$.

One difference of note is that our “first-order analysis” (so-called because it does not treat calls to unknown functions with much precision) is still applied to a higher-order language, whereas Mazur operates on a first-order subset of Mercury. This forces us to support the creation of closure values soundly, by tracking which variables are aliased to each closure’s closure data. We also directly support imperative looping constructs, which do not appear in the Mercury language.
8.5 Conclusion

In this chapter, a simplified abstract interpretation for Mars was presented that determines whether any two variables, or components thereof, may alias at any program point. This simplified analysis is highly precise for first-order cases, and still sound in higher-order cases, albeit with a severe loss of precision. The analysis is performed via a context-insensitive algorithm; a summary of each function is generated via the condensing transformation of Section 6.6.

This analysis draws together concepts from other aliasing analyses, in particular the type folding of Bruynooghe [11] and alternating closure of Mulkers [74]. As such, our analysis is approximately as precise as Mazur’s analysis for Mercury [65], but is tailored for an imperative language (e.g., we have a rule for while loops), and also contains a treatment for aliasing with static (global) data and closures (function objects).

Our analysis explicitly tracks aliasing between pairs of variables, so that if a variable becomes dead (provably unused for the remainder of the function), variables that were aliased to it may be considered unique again. This allows destructive update of variables that were once aliased. This also allows us to track when a variable $a$ may alias one of two variables $b$ or $c$, but not both, so that if $a$ becomes dead, we do not consider $b$ and $c$ to be aliased.

We also track the components of variables independently. For example, we can know the aliasing of each field of a non-recursive record type as precisely as separate variables. Only upon encountering recursive types or arbitrary-length arrays do we begin to lose precision and combine memory cells together.

In the next chapter, we extend this simplified analysis with the capability to precisely track aliasing in higher-order programs, including functions that call closures passed as arguments, as well as functions that return closures.
Chapter 9

Higher-order uniqueness inference

9.1 Introduction

In the previous chapter, an abstract interpretation was presented for determining whether any two variables, or their components, may be aliased. The analysis allows for each function’s procedure summary (uniqueinfo) to be computed in a context-independent fashion, to precisely track aliasing across function boundaries. While the analysis is sound in the presence of closures, they present a major source of imprecision — any call to an unknown function forces the analysis to assume that the result is aliased to all parts of all arguments and the global state. In this chapter, we revise the analysis to treat higher-order functions with the same degree of precision as first-order functions. This work forms the main contribution of this thesis.

It is important to note that a first-order analysis such as the one presented in Chapter 8 is sufficient in certain cases, such as passing a single function to another function, when the compiler performs inlining or specialisation of higher-order functions. However, in more complex cases, such as aggregate data structures containing function values, a proper higher-order abstract interpretation is required to maintain precision.

The analysis is introduced in three phases in this chapter. First, we expand the abstract semantics from the previous chapter to deal with upward higher-order functions, which return functions as their result. We now record not only the sharing created as a result of a function, but also the uniqueinfo of the value that it creates. Second, we further expand the semantics to deal with downward higher-order functions, which accept functions as inputs. To maintain context independence, we introduce algebraic uniqueness parameters for function arguments, allowing their abstract values to be tracked as they move through the program. Finally, we expand the uniqueness algebra to the sharing graph itself, allowing the analysis of applicative downward higher-order functions (algebraically tracking
the sharing that results from calling an unknown function).

The remainder of this chapter is organised as follows. We discuss and refute claims that higher-order analysis is unnecessary in Section 9.2, justifying our investment in this advanced algebraic approach. In Section 9.3, we give a brief overview of the changes to the first-order abstract semantics to make it precise for higher-order programs. In Sections 9.4, 9.5 and 9.6, we work through the new rules in detail, with examples and diagrams. These sections discuss the approach for upwards, non-applicative downwards and applicative downwards higher-order functions, respectively. In Section 9.7, we give a precise mathematical description of the higher-order analysis, by formally defining the abstraction, and giving the abstract memory denotational semantics. In Section 9.8, we discuss the algebraic data structures that are used in the bottom-up analysis to obtain the complete higher-order summary of a function independently of its call site. In Section 9.9, we briefly discuss the higher-order analysis of non-Mars features, such as type classes, existential types, and lazy evaluation. Finally, Section 9.10 concludes.

9.2 Why analyse higher-order?

With higher-order analysis as one of our major contributions, it is worth addressing claims that it is not necessary to perform sharing or reuse analysis of higher-order functions. As Mars is a language with first-class closures, we certainly need to address this issue. However, one may claim that a compiler could first transform a program into a first-order program by specialising any uses of higher-order functions, and then analyse the resulting first-order program. In this section, we examine the drawbacks of such an approach.

An example of such a claim appears in Mazur [65], who argues (in this case, with respect to the Mercury programming language):

[A ray tracer benchmark] uses the classic map, foldl, filter procedures defined for manipulating list structures. As such, higher-order calls are not handled at all by our CTGC system. Yet it appears that all the higher-order elements in the ray tracer are specialised into plain first-order predicates. The code that the CTGC system therefore handles is a simple first-order logic program, yielding the interesting results discussed earlier. This seems to suggest that in the presence of a good performing higher-order specialising compiler, there is no real need to extend the CTGC system to cope with higher-order language features itself. Moreover, such an extension is far from trivial.

The argument relies on a compiler pass illustrated in Example 9.1. This is not an optimisation currently performed by the Mars compiler, but it is suggested that such a pass
9.2. WHY ANALYSE HIGHER-ORDER?

would absolve us of the need to perform higher-order CTGC.\(^1\) The code in Example 9.1a is the Mars library implementation of \texttt{foldl} and a typical higher-order function that neatly converts a linked list to an array by folding the \texttt{array_add} function over the list. The \texttt{array_add} function is a built-in that non-destructively appends a single element to the end of an array, returning the resulting array; it is one of the built-ins that will automatically be made destructive by our CTGC system if the opportunity arises.

\begin{lstlisting}[language=Python]
Example 9.1 Usage of \texttt{foldl}, before and after higher-order specialisation

```python
def foldl(f, acc, list):
    switch list:
        case Nil:
            return acc
        case Cons(x, xs):
            t = f(acc, x)
            return foldl(f, t, xs)

def list_to_array(list):
    return foldl(array_add, [], list)
```

```python
def foldl_array_add(acc, list):
    switch list:
        case Nil:
            return acc
        case Cons(x, xs):
            t = array_add(acc, x)
            return foldl_array_add(t, xs)

def list_to_array(list):
    return foldl_array_add([], list)
```

Neither our first-order CTGC system described in the previous chapter, nor Mazur’s CTGC system for Mercury, can destructively update the array if given the code on the left (resulting in \(O(n^2)\) execution time). However, a compiler could automatically transform it into Example 9.1b, in which a special version of \texttt{foldl} (here called \texttt{foldl_array_add}) has been generated that does not take a parameter \(f\), but is instead hard-coded to apply \texttt{array_add} to each list element. The \texttt{foldl_array_add} function is an ordinary first-order function and both of our analyses would happily compile a destructive update version of the code on the right, resulting in \(O(n)\) execution time.\(^2\)

This is a sensible argument, but it does not entirely justify the lack of a CTGC system that natively handles higher-order procedures, for a variety of reasons.

First and foremost, it only works in straightforward cases, such as a direct call to \texttt{foldl} with a known function name, or a simple partial application (for example, partially applying \texttt{add(n,...)} in Mars notation, and passing the result to \texttt{map}). However, this approach is not able to specialise the higher-order function in any of the following more complex situations:

- the function variable is assigned a different closure based on some condition, and then passed to a higher-order function,

\(^1\)An equivalent optimisation can be enabled in the Melbourne Mercury Compiler via the \texttt{-optimise-higher-order} flag.

\(^2\)While, as far as we can tell, Mercury’s CTGC system does not optimise array update operations at all, it can optimise similar examples.
CHAPTER 9. HIGHER-ORDER UNIQUENESS INFERENCE

- a function returns a closure, and the result is passed to a higher-order function, or
- a function value is stored in a data structure (for example, a list of function values) and the data structure is passed as an argument.

Clearly, not all uses of function values will result in specialisation, and any case that does not may result in a failure to detect reuse opportunities. Since such a specialisation is based on heuristics, this makes the whole CTGC algorithm unpredictable in the face of higher-order programming.

Secondly, relying on such a specialisation makes it harder to reason about the CTGC system. Ideally, we would like to be able to produce a general statement about the aliasing behaviour of a procedure. Just as we did in Chapter 8 for append, we would like to be able to generate a report for map that says, roughly speaking, “the spine of the resulting list is unshared, but the elements of the resulting list are shared according to the sharing properties of the function passed to map,” and furthermore, “if the function passed to map, f, has a destructive version and the elements of the list are unshared, map will call the destructive version of f.” If we rely on specialising higher-order function calls before analysis, then the best we can do is give specific reports about individual specialisations of map, and not its general aliasing or destructive update behaviour. The same would be true for all higher-order functions.

Thirdly, performing a true higher-order analysis decouples the compile-time garbage collection system from the code generator’s strategy for implementing first-class functions. For example, if the heuristic decides it is not worth specialising a given higher-order function call, an approach relying on the specialisation would fail to optimise this case. A small change to the program could potentially change its space and time complexity. This decoupling also means it is permissible to turn down or disable the specialisation if desired, avoiding the associated code bloat without compromising the optimisation.

Finally, our higher-order analysis maintains the context insensitivity of our first-order analysis: we can analyse a higher-order library function, such as map, and store the result. If the analysis were to rely on higher-order specialising, it would require separate analysis of each specialised version of each higher-order function, increasing compilation times.

9.2.1 Defunctionalisation

Another approach to mechanically converting a higher-order program into a first-order one is defunctionalisation [79]. This approach is a global program transformation that relies upon identifying all functions in the program that are used as values (in MIR terminology, all closure templates). An algebraic data type, FUNVAL, is introduced that
9.2. WHY ANALYSE HIGHER-ORDER?

contains a constructor for each function in the program. All function values are replaced by FUNVAL values, which are first-order. A call to a function value is replaced by a call to apply, which switches over all possible FUNVAL constructors and calls the appropriate function.

Example 9.2 Use of foldl, before and after global defunctionalisation

```python
def foldl(f, acc, list):
    switch list:
    case Nil:
        return acc
    case Cons(x, xs):
        t = f(acc, x)
        return foldl(f, t, xs)
def sum(list):
    return foldl(Add, 0, list)
def product(list):
    return foldl(Mul, 1, list)
```

```python
def apply2(f, x, y):
    switch f:
        case Add:
            return add(x, y)
        case Mul:
            return mul(x, y)
def foldl(f, acc, list):
    switch list:
        case Nil:
            return acc
        case Cons(x, xs):
            t = apply2(f, acc, x)
            return foldl(f, t, xs)
def sum(list):
    return foldl(Add, 0, list)
def product(list):
    return foldl(Mul, 1, list)
```

Example 9.2 shows a global defunctionalisation process applied to a short higher-order program. Note that in the defunctionalised version (Example 9.2b), the calls to foldl pass the simple enumeration values Add and Mul, rather than the code-carrying closures add and mul. Inside foldl, rather than directly calling f, apply2 is called, which calls the correct function based on the value of f.

This is a first-order program, but its prospects for meaningful static analysis are quite diminished: the analysis for apply will be the least upper bound of all of the functions in FUNVAL, and this result will filter up through its callers (e.g., foldl). This means that, for example, the analysis for sum would include the results of mul, even though mul has nothing to do with sum. The larger the program becomes, the worse the analysis becomes for all uses of higher-order functions.

To be fair, the type system can offer some relief from this global weakening, as functions of different types need to be segregated into separate apply functions, so the joining of results of different functions is limited to functions of the same type. Nevertheless, the
fact that an analysis can be weakened by completely unrelated code makes this approach a poor substitute for a true higher-order analysis on un-transformed program code.

9.3 Overview

In this section, we give an informal overview of the changes to the analysis to ready it for higher order functions. We consider two separate classes of higher-order function: downward higher-order and upward higher-order.

In a downward higher-order function, a function value is passed in as an argument. Common downward examples are `map`, `foldl`, `foldr` and `filter`. In Mars, the signature of `map` is:

$$\text{map}(f :: a \rightarrow b, v :: \text{List}(a)) :: \text{List}(b)$$

The first parameter, $f$, is a downward function argument, and hence `map` is considered a downward higher-order function. Under the first-order analysis, the call to $f$ would be considered a call to an unknown function, and so the compiler would conservatively assume that all parts of all elements of the resulting list are aliased to all parts of all elements of the input list, as well as the global state (but note that even this simple analysis would realise that the spine of the result does not alias the spine of the input). We would like the analysis of `map` to reflect the specific function that was passed in as parameter $f$.

In an upward higher-order function, a function value is returned as the result. Common examples include any function written in the curried style. For example, the function `const` takes a value and returns a function that returns the original value when called. In Mars, this has the signature:

$$\text{const}(k :: a) :: b \rightarrow a$$

The result of this function is a function, and hence `const` is considered an upward higher-order function. Under the first-order analysis, the result of `const` would be considered an unknown function, and so any subsequent usage of the result would imprecisely assume too much aliasing. We would like the analysis of `const` to give precise details about the aliasing and destructive update behaviour of the resulting function. Note that partial application can be considered a form of upward higher-order programming, and is affected in the same way.

Note that the terms are not mutually exclusive. An obvious example of a downward

---

3This terminology is reminiscent of the so-called “downward funarg” and “upward funarg” problems; see [http://en.wikipedia.org/wiki/Funarg_problem](http://en.wikipedia.org/wiki/Funarg_problem).
and upward higher-order function is a curried version of \texttt{map}, with the following signature:

\[
\texttt{map}_\text{curried}(f :: a \to b) :: \text{List}(a) \to \text{List}(b)
\]

Another common function that is both downward and upward higher-order is \texttt{compose} (in Haskell, known as \texttt{.(.)}):

\[
\texttt{compose}(f :: b \to c, g :: a \to b) :: a \to c
\]

The applicability of these terms becomes much more widespread when one considers that any type-parameterised variable could hold a function. Consider the innocuous \texttt{id} function:

\[
\texttt{id}(x :: a) :: a
\]

This seems like a simple first-order function until one realises that the type variable \(a\) could be a function type. In light of this, we must consider \texttt{id}, and many other type parameterised functions, to be both downward and upward higher-order functions. That means we need to do more work when analysing these apparently first-order functions, so as to preserve information about function values. (For example, the expression \texttt{id}(f)(x) should be equivalent to the expression \(f(x)\), which means \texttt{id} needs to preserve aliasing information about \(f\).) Furthermore, higher-order functions with type parameters may become even higher order — for example, \texttt{map} is apparently a second-order function, but if the second argument is a list of functions, then \texttt{map} is actually a third-order function, and the higher-order analysis needs to deal with not only the unknown function parameter, but also the functions in the list.

We can further divide the downward higher-order functions into two classes. Any function that calls a function that it accepts as a parameter is called an \textit{applicative downward higher-order function}. For example, \texttt{map} is applicative downward higher-order because it actually calls the function \(f\), whereas \texttt{id} (given a function value) is not applicative because it merely passes the function value along without using it. The simplest\(^4\) applicative downward higher-order function is \texttt{apply}:

\[
\texttt{apply}(f :: a \to b, x :: a) :: b
\]

It should be noted that applicative downward higher-order functions are significantly harder to deal with than non-applicative downward or upward higher-order functions. This is because as we pass a function value upward or downward without calling it, we only need to algebraically track the aliasing behaviour of the function. When it does get

---

\(^{4}\)There is a simpler one, \texttt{apply}_0(f :: () \to a) :: a, but that is too trivial to be interesting.
called, it has known aliasing behaviour. It is when an unknown function is called that we really start to run into trouble, because at that point, it is unclear which edges need to be inserted into the local sharing graph. Therefore, we will tackle the three problems in increasing order of difficulty:

1. the upward higher-order problem, then
2. the non-applicative downward higher-order problem, then
3. the applicative downward higher-order problem.

The solution to the non-applicative cases is to extend the abstract domain to algebraically store the aliasing behaviour of every function variable. In the previous chapter, we introduced the concepts of a sharing graph (which represented the entire local state during the abstract interpretation of a procedure) and a uniqueinfo (which represented the result of analysis of a particular function). A uniqueinfo is an abstract function, mapping return and parameter names onto a sharing graph. When calling a function, we use the uniqueinfo to determine what new edges to add to the local sharing graph. In the previous chapter, only global functions had uniqueinfos; when any local function variable was called, we assumed the worst. In this chapter, we extend the local abstract state to include a uniqueinfo map, a map from variable names to uniqueinfos. Whenever a function value is created (by applying a closure template), the resulting uniqueinfo is stored in the variable’s entry in the uniqueinfo map. When that variable is called, the analysis can use the uniqueinfo map to determine the aliasing behaviour of the function, rather than assuming the worst.

When different code paths assign different uniqueinfo values to a function variable, we find the least upper bound of the different uniqueinfos (the most precise uniqueinfo that satisfies the aliasing behaviour of all of the joined uniqueinfos). We also extend the concept of a uniqueinfo to cover structures and arrays, allowing us to represent nested abstract functions (for example, a pair or array of functions).

This handles the analysis of local function variables, but it does not yet solve the upward higher-order problem. We also need to extend the abstraction of a function to include the uniqueinfo of the result as well as the sharing graph. This allows us to describe an upward higher-order function in terms of not just the aliasing it creates, but also the aliasing behaviour of the function it returns. This is recursive: given a function that returns a function that returns a function, we can capture the aliasing information of each function in the sequence.

We solve the non-applicative downward higher-order problem by introducing uniqueinfo parameters. In the abstraction of a function, each parameter is given a name for its (as
yet unknown) uniqueinfo. That name may be used in place of a known uniqueinfo in the uniqueinfo map or the result of the function. In this way, we can handle any function that accepts function parameters and treats them opaquely, such as id.

Lastly, we solve the applicative downward higher-order problem. This problem occurs when we encounter a call to a function with a uniqueinfo parameter. We solve it by extending the algebra of uniqueinfos to the sharing graph itself. When a call to a uniqueinfo parameter occurs, we create all possible new sharing graph edges (as with the “assume the worst” scenario), but we introduce conditions on those edges that depend upon the uniqueinfo parameter. When a real uniqueinfo is substituted for the uniqueinfo parameter (at the point where the applicative downward higher-order function is called), we resolve all of these edge conditions according to the real uniqueinfo. This lets us continue with the bottom-up analysis, yet makes usage of applicative downward higher-order functions just as precise as with first-order functions.

This chapter will introduce a number of new symbols to refer to abstract locations in higher-order procedures. For reference, Table 9.1 gives a summary of the Name symbols used in this chapter. The last three symbols are qualifiers, applied to other names.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>return value (convention)</td>
</tr>
<tr>
<td>%</td>
<td>me (convention)</td>
</tr>
<tr>
<td>@</td>
<td>static data</td>
</tr>
<tr>
<td>[]</td>
<td>array contents</td>
</tr>
<tr>
<td>()</td>
<td>closure data</td>
</tr>
<tr>
<td>#</td>
<td>result data</td>
</tr>
</tbody>
</table>

Table 9.1: Summary of symbols used to construct sharing graph names

9.4 The upward higher-order problem

As in the previous chapter, we first present some worked examples of the analysis before describing it formally. For clarity, this section presents a simplified view that does not include uniqueness parameters; those will be introduced in Section 9.5, where we will revisit these examples.

9.4.1 const

The const function is a simple example of an upwards higher-order function. The function is defined in MIR using closure templates in Example 9.3.

Under the first-order scheme of the previous chapter, we would assign the following
Example 9.3 The const function

```python
def const₁(y :: a)(z :: b):
    return y

def const(x :: a):
    return const₁{x}
```

abstract values to these procedures:

\[
\begin{align*}
\text{const₁} & : \{a\}(b) \to a \\
\text{const₁α} & : \text{Name} \to \text{Name} \to \text{Name} \to \text{Sharing} \\
\text{const₁α} y z $ & = \{\{\$, y\}\} \\
\text{const} & : a \to (b \to a) \\
\text{constₐ} & : \text{Name} \to \text{Name} \to \text{Sharing} \\
\text{constₐ} x$ & = \{\{\$, x\}\}
\end{align*}
\]

The analysis of \text{const} only tells us that \text{x} cannot be modified while the resulting closure is alive — it does not tell us anything about what happens when we \text{call} the result. To obtain a more precise result for \text{const}, we change the analysis so that as well as returning a Sharing, it also returns the uniqueinfo of the result.

Here is the higher-order analysis for \text{const}:

\[
\begin{align*}
\text{constₐ} & : \text{Name} \to \text{Name} \to \langle\text{Sharing, Name} \to \text{Name} \to \text{Name} \to \text{Sharing}\rangle \\
\text{constₐ} x$ & = \langle\{\{\$, x\}, \{\$, x\}\}, \lambda z \$. \ {\{\$, \%\}\
\end{align*}
\]

Quite a complex result! Let us break it down. First, to the Sharing:

\[
\{\{\$, x\}, \{\$, x\}\}
\]

We have introduced a new type of name: for each function \text{f}, there is a name \text{f#} (pronounced “\text{f}'s result data”). It has an edge to things that will be aliased to the result of \text{f}, when \text{f} is applied. This is essentially the same as the closure data (\(\text{f}()\)), but it is typed: the type of \text{f#} is \text{f}'s return type. The fact that \text{f#} is typed means it has sub-components and can alias names at the appropriate level, in contrast to the monolithic “value soup” that is \(\text{f}()\). In our \text{const} example, there is an edge between \text{x} and \$, because calling \$
9.4. THE UPWARD HIGHER-ORDER PROBLEM

\[ f = \text{const}(p) \]

This results in the sharing graph \( \{ \{ f(), p \}, \{ f\#, p \} \} \) (shown in Figure 9.1a).

The closure-data node \( (f()) \) is still necessary, because it is possible for a closure to hold a reference to an object that is not included in its result — for example, the closure created by \text{array\_length}(a, \ldots)\). However, in the \text{const} case, the edge \( \{ f(), x \} \) is redundant.

Now, to the uniqueinfo part of \text{const}\_a:

\[ \lambda \ z \ \$'. \ % \ . \ \{ \$', \%\# \} \]

This uniqueinfo is a little different to top-level uniqueinfos we have seen previously: it takes an extra Name argument, by convention called \% (pronounced “me”). The % name refers to the function with this uniqueinfo — it is the function known as $ in the context of \text{const}\_a (the result of \text{const}). The result of this lambda is another sharing graph: \( \{ \{ \$', \%\# \} \} \), which says that the result of this function will alias %# (pronounced “my-result”), which is $# in the context of \text{const}\_a, which is synonymous with x.

Why do we have such a long-winded way of saying that \$' aliases x? Uniqueinfos must be portable: the Names they refer to cannot be free variables. When analysing a function, a uniqueinfo map keeps track of the uniqueinfo of any function variables. After our example call \( f = \text{const}(p) \) above, the uniqueinfo map will include:

\[ f \Rightarrow \lambda \ z \ \$'. \ % \ . \ \{ \$', \%\# \} \]
The notation \( f \Rightarrow u \) means “\( f \) has uniqueinfo \( u \).”

The uniqueinfo of \( f \) is directly copied from the result of \( \text{const}_\alpha \); notice that it does not include references to \$ or \( x \). This is important, because both \$ and \( x \) are now out of scope.

If we now consider a call to \( f \):

\[
    r = f(q)
\]

This call can now be analyzed like any other call to a static function. Taking the uniqueinfo of \( f \), we substitute \( q \) for \( z \), \( r \) for \$‘ and \( f \) itself for \%. This gives us the sharing edge \( \{ r, f\# \} \). Using the propagation rule (\( \star \)), we obtain the new edge \( \{ r, p \} \) (shown in Figure 9.1b). The value of \( p \) was passed through to \( x \) inside \( \text{const} \), stored in \( f \)’s closure data, returned as the result of \( f \) and stored in \( r \); our sharing graph accurately reflects this result.

It is worth noting why the explicit edges between \$‘ and \%# (and, potentially, their corresponding components) are required: it allows us to be more precise when the result, or part thereof, does not alias a closure variable. For example, consider a closure \( g \) with uniqueinfo \( \lambda ( \) \$ \%. \( \emptyset \) (which creates no aliasing). If we saw the statement \( t = g() \), we would know that \( t \) has no aliases, as opposed to keeping \( t \) tied to \( g\# \) for the lifetime of \( g \).

This is a much more precise result for \( \text{const} \) than our first-order analysis. However, we have so far avoided the possibility that the \( x \) argument to \( \text{const} \) may itself be a function. That possibility will be handled when we return to this example in Section 9.5.2.

### 9.4.2 Closures with local variables

Example 9.4 is another upwards higher-order function, but in this example, the resulting closure returns a local variable which the caller has no access to. If not carefully tracked, some aliasing can be missed.

**Example 9.4 A closure with hidden local memory cells**

```python
def const_array():
x = [1, 2, 3]
return const(x)
def problem():
g = const_array()
y = g(0)
z = g(0)
y1 = array_add(y, 4)
return z
```

The sharing graph of \( \text{const}_\text{array} \) after Line 3 is \( \{ \{\$, x\}, \{\$, x\} \} \) (from the definition

---

5The \( f \Rightarrow u \) notation is used in favour of the general map notation \( \{ f \mapsto u \} \), because the former can be used for a selected set of variables, whereas the latter is not valid unless all entries in the map are given.
of \( \text{const}_a \). However, as usual, we remove \( x \) from the sharing graph because it is not an argument or result node; hence, the final sharing graph for \( \text{const-array} \) is \( \{ \} \). The uniqueinfo map for \( \text{const-array} \) is as follows:

\[
\Rightarrow \lambda \ z \ z' \ %, \{ \{\} \ %\} \\
\]

Thus the value of \( \text{const-array}_a \) is:

\[
\text{const-array}_a \ ::= \ () \rightarrow \text{Name} \rightarrow \langle \text{Sharing, Name} \rightarrow \text{Name} \rightarrow \text{Name} \rightarrow \text{Name} \rightarrow \text{Sharing} \rangle \\
\text{const-array}_a \ ( ) \ \Rightarrow \ \{\}, \lambda \ z \ z' \ %, \{ \{\} \ %\} \}
\]

Note the aliasing behaviour of \( \text{problem} \): the result of \( \text{const-array} \) is stored in \( g \), and then \( g \) is called twice, assigning the resulting array to \( y \) and \( z \). As both \( y \) and \( z \) are assigned the same array, we must ensure that the analysis considers \( y \) and \( z \) to be aliased or we may inadvertently mutate \( z \) when we update \( y \). The problem, in this case, is that the sharing graph of \( \text{const-array} \) is empty, which implies that the result of \( g \) has no aliases.

While it is true that the result is not aliased to any known variable names, there is a memory cell that is shared between all results of \( g \).

This aliasing is automatically taken care of by the result-node feature described previously. Figure 9.2 shows the internal sharing graph for \( \text{problem} \) after each line of analysis. We see that \( y \) and \( z \) share a common link to \( g\# \), and therefore become aliases of one another via the alternating closure rule.
This example shows that the result-data node can genuinely represent an actual memory cell that would otherwise not be given a name. By referring to the result-data node in the uniqueinfo, the anonymous memory cells inside the closure can be successfully tracked if they escape from the closure. Note that the result-data node has a full type hierarchy, so the system can precisely track aliasing with specific components of the result.

9.4.3 Structured uniqueinfos

So far, we have discussed uniqueinfos as abstractions of function values, and in this chapter, we have treated uniqueinfos as first-class abstract values. However, this view limits us to describing functions that are stored directly in variables or passed directly as arguments — we will lose all sharing information about a function if it is placed inside a data structure or array. In this section, we rectify this by extending uniqueinfos to cover all data types, much as we extended sharing graphs to deal with structured data in Section 8.2.5.

Consider the Pair data type. Until now, we have not thought of a Pair as having a uniqueinfo. Indeed, there is no need for one in a simple case like Pair(1, 2). But if we ever encounter a Pair of functions, then in order to precisely recall the sharing behaviour of those functions, we need to remember the uniqueinfo of the pair itself.

To this end, we extend uniqueinfos so that they apply to all values. Primitive values other than functions have no abstract data. Container types hold recursive uniqueinfos, so that we can precisely represent the uniqueinfos of functions that may be contained therein. Specifically:

- The uniqueinfo of a function is, as described above, an abstract function mapping Names to (Sharing, uniqueinfo) pairs.

- The uniqueinfo of a user-defined type value is a tuple of uniqueinfos, with one element for each field (flattened across all constructors, with ⊥ in any element that could not possibly exist). For example, the uniqueinfo of a Pair value Pair(f, g) is (α(f), α(g)). The uniqueinfo of an Either value Right(f) is (⊥, α(f)). The uniqueinfo of any enumeration value (any type with no fields in any constructor) is ()

- The uniqueinfo of an array is simply the join of the uniqueinfo of all of the elements of the array. We use square brackets to denote an array with elements of a given uniqueinfo: [x] is an array whose elements have uniqueinfo x. For example, the uniqueinfo of an Array value [f, g] is [α(f) ⊔ α(g)]. The uniqueinfo of the empty array [] is [⊥].
• The uniqueinfo of a Num or other possible built-in types that cannot hold functions is (), denoting the empty tuple (equivalent to a user-defined type with no fields in any constructor).

• The uniqueinfo \( \uparrow^n \) denotes a cyclic uniqueinfo caused by type folding. For example, the uniqueinfo \((a, (b, \uparrow^2))\) is equivalent to \((a, (b, (a, \ldots)))\). The value \( \uparrow^2 \) means that the value is equivalent to the uniqueinfo two levels above, and allows the uniqueinfo to be represented in finite space.

Up until now, functions that return non-function values have produced a Sharing, while functions that return functions have produced a \(\langle\text{Sharing}, \text{uniqueinfo}\rangle\) pair. Now that all values have uniqueinfos, we can generalise this definition — all functions now produce a \(\langle\text{Sharing}, \text{uniqueinfo}\rangle\) pair. For example, the extended uniqueinfo for \texttt{add}\ is:

\[
\texttt{add}_\alpha :: \text{(Name, Name)} \to \text{Name} \to \langle\text{Sharing}, ()\rangle
\]

\[
\text{add}_\alpha (x, y) $ = \langle\{\}, ()\rangle
\]

Our ongoing \texttt{const}\ example is tricky because the type of the result of the result is polymorphic. At this stage, all we can say about it is “\(\top\)” (i.e., we do not know anything).

\[
\texttt{const}_\alpha :: \text{Name} \to \text{Name} \to \langle\text{Sharing, Name} \to \text{Name} \to \text{Name} \to \langle\text{Sharing, a}\rangle\rangle
\]

\[
\text{const}_\alpha x $ = \langle\{\{\}_x, \{\}_x\}, \lambda z $ \% $. \langle\{\{\}_x, \% \}_x\}, \top\rangle\rangle
\]

This deficiency will be rectified in the next section, when we consider a function like \texttt{const} to be dealing with unknown function values.

### 9.5 The non-applicative downward higher-order problem

In this section, we continue with the examples, introducing the concept of uniqueness parameters briefly discussed in Section 9.3.

#### 9.5.1 \texttt{id}

The \texttt{id} function, shown in Example 9.5, is the simplest example of a non-applicative downward higher-order function, if its argument is considered to be a function. Because we do not know whether the argument will be a function or not, we need to assume it is.

As with \texttt{const}, the type of the result is polymorphic. Because we have no idea what function could be in the parameter variable \(x\), we cannot give it a “real” uniqueinfo.
Example 9.5 The id function

\[
def \text{id}(x :: a):
    \text{return } x
\]

Instead, we modify the abstract parameters to \(\text{id}_\alpha\) such that rather than taking a Name for each concrete parameter, we take a \((\text{Name}, \text{uniqueinfo})\) pair. Our abstract functions now explicitly pass the uniqueinfos of all arguments when they call each other.

Our convention will be to give the uniqueinfo parameter the concrete parameter’s name with a “\(ui\)” subscript. Hence, in the abstract view, \(x\), as always, refers to the name the caller has given to the concrete parameter \(x\), while \(x_{ui}\) refers to the uniqueinfo of \(x\). We initialise our uniqueinfo map with the uniqueinfo of each parameter:

\[
x \Rightarrow x_{ui}
\]

Uniqueinfo parameters can be passed around just like any other uniqueinfo. The only difficulty is how to make use of them if the function is called; by definition, this only occurs in an applicative downward higher-order case, which we will deal with in Section 9.6.

For now, it is simple: the uniqueinfo of \(x\) is copied to the result of \(\text{id}\), and so the analysis result is:

\[
\begin{align*}
\text{id} & :: a \rightarrow a \\
\text{id}_\alpha & :: (\text{Name}, a) \rightarrow \text{Name} \rightarrow (\text{Sharing}, a) \\
\text{id}_\alpha (x, x_{ui}) \$ & = \langle \{ \{\$, x\} \}, x_{ui} \rangle
\end{align*}
\]

This can be read as follows: “\(\text{id}_\alpha\) is an abstract function, given \(x\) (the name of the argument passed to \(\text{id}\) in the caller’s sharing graph), \(x_{ui}\) (the uniqueinfo of the argument passed to \(\text{id}\)) and \(\$\) (the name of the variable assigned the result of \(\text{id}\) in the caller’s sharing graph). As a result of calling \(\text{id}\), an alias will be introduced between the variables named \(x\) and \(\$\), and the variable named \(\$\) will have the uniqueinfo \(x_{ui}\).”

9.5.2 \ const

Now that we have a framework for dealing with non-applicative downward higher-order functions, we return to the example of \(\text{const}\) introduced in Section 9.4.1. Recall that we analysed the second-order case for \(\text{const}\), but our previous result would lose precision if the parameter to \(\text{const}\) was itself a function. We can now analyse \(\text{const}\) in full using the same technique as with \(\text{id}\):
const₁ :: \{a\}(b) \rightarrow a
const₁ₐ :: ⟨Name,a⟩ \rightarrow ⟨Name,b⟩ \rightarrow Name \rightarrow ⟨Sharing,a⟩

\text{const₁ₐ}\langle y,y_{ui}\rangle \langle z,z_{ui}\rangle $\text{=}\langle\{\{\$\},y\}\},y_{ui}\rangle$

const :: a \rightarrow (b \rightarrow a)

\text{constₐ} :: ⟨Name,a⟩ \rightarrow Name \rightarrow ⟨Sharing,⟨Name,b⟩ \rightarrow Name \rightarrow Name \rightarrow ⟨Sharing,a⟩⟩

\text{constₐ}\langle x,x_{ui}\rangle $\text{=}\langle\{\{\$\},x\}\},λ\langle z,z_{ui}\rangle \text{%'s}.\langle\{\{\$\}',\%\}\},x_{ui}\rangle\}$

The analysis of \text{const₁} is very much like \text{id} — we report not only the aliasing of \text{y} to the result, but also that the result will have the same uniqueinfo as \text{y}. The analysis of \text{const} works much the same as it did in Section 9.4.1 except we now report the uniqueinfo of the result of the result, which is the same uniqueinfo as \text{x}.

It is acceptable to have a free uniqueinfo variable inside a uniqueinfo, as we have here with \text{x_{ui}}, because, unlike Names, uniqueinfos are independent of the context in which they appear.

### 9.5.3 Uniqueinfo algebra

With the introduction of uniqueinfo parameters, we find that several of the operations on uniqueinfos will not work when the uniqueinfos are unknown. Consider the simple case of a field-reference expression, \text{x.Cons/0}. If \text{x} has uniqueinfo \text{(a_{ui},⊥)}, then the result of the expression will have uniqueinfo \text{a_{ui}}. However, if \text{x} has uniqueinfo \text{x_{ui}}, then we cannot simply open up the tuple and pull out its first element. We must preserve this expression algebraically, as \text{x_{ui0}}. Our analysis must be prepared to handle arbitrarily large algebraic uniqueinfo terms involving field access, array indexing, and join (⊔) operations, which can only be resolved once the value of \text{x_{ui}} is supplied by the caller. The only type of operation we do not (yet) need to handle is function application, since in this section, we do not attempt to solve the applicative downward higher-order problem — that will be tackled in the next section.

Consider Example 9.6, which retrieves either the first or second field of a pair \text{p}, depending on the \text{i} parameter. As \text{p} is a parameter, its uniqueinfo is known only as \text{p_{ui}} throughout this function. Thus, we cannot access the components of its uniqueinfo until the caller supplies one. Because we are performing a bottom-up analysis, we need an answer before then, so we give an algebraic one.
Example 9.6 Accessing a field of a Pair

```python
def getfield(p :: Pair(a, a), i :: Num):
    if i:
        r = p.Pair/1
    else:
        r = p.Pair/0
    return r
```

After Line 3, we say the uniqueinfo of $r$ is $p_{ui1}$, and after Line 5, we say the uniqueinfo of $r$ is $p_{ui0}$. When joining uniqueinfos before Line 6, we would normally join the individual components of the uniqueinfo (for example, taking the set union of their sharing graphs), but since we have no real uniqueinfo, we again preserve the join algebraically, giving $r$ the uniqueinfo $p_{ui0} \sqcup p_{ui1}$. This gives us the final uniqueinfo for this function:

$$
\text{getfield}_a \left( \langle p, p_{ui} \rangle, \langle i, i_{ui} \rangle \right) = \{ \{\$, p.0\}, \{\$, p.1\} \}, p_{ui0} \sqcup p_{ui1}
$$

Similarly, the built-in function `array_ref` could be considered a downward non-applicative higher-order function (if the input array contained functions), and thus, it must have a higher-order-aware uniqueinfo as well:

$$
\text{array_ref}_a \left( \langle a, a_{ui} \rangle, \langle i, i_{ui} \rangle \right) = \{\$, a[\]}, a_{ui[\]}
$$

9.6 The applicative downward higher-order problem

Recall that an applicative downward higher-order function is one that a) accepts function values as parameters, and b) calls those function values. While we are analysing a function in a context-insensitive manner (without knowing the uniqueinfos of the function’s arguments), it is impossible to accurately update the sharing graph data structure with the result of the call to the unknown function.

In this section, we introduce by example the techniques for analysing such functions precisely in a context-insensitive analysis. We extend the sharing graph data structure as outlined in Section 9.3.

Before proceeding, however, it should be noted that, from this point onwards, we do not actually extend the abstract semantics in any way. The concepts we have covered so far are sufficient to mathematically describe the abstraction of any function. The difficulty is that an implementation attempting to follow the abstract semantics, upon encountering an application of an unknown function, will need to be able to represent the resulting data structures algebraically, treating the unknown function’s uniqueinfo as an
unknown value for the duration of the analysis. This section introduces the techniques that our compiler uses to represent such algebraic terms, and a visual representation for the compiler’s internal data structures.

We first introduce two helper functions, sg and ui, which extract the two components of a \(\text{\langle Sharing, UniqueInfo \rangle}\) pair:

\[
\begin{align*}
\text{sg} \langle s, u \rangle &= s \\
\text{ui} \langle s, u \rangle &= u
\end{align*}
\]

### 9.6.1 apply

We begin with an analysis of the simplest applicative higher-order function possible, apply, which simply takes a function and its argument, and applies the function to the argument, returning the result. The code for this simple function is shown in Example 9.7.

**Example 9.7** The apply function

1: `def apply(f :: a -> b, x :: a):
  2:     return f(x)`

We begin our analysis with an empty sharing graph and the uniqueinfo map:

\[
\begin{align*}
f & \Rightarrow f_{ui} \\
x & \Rightarrow x_{ui}
\end{align*}
\]

With \(f\) having concrete type \(a \to b\), \(f_{ui}\) is a member of the set:

\[
\langle \text{Name}, a \rangle \to \text{Name} \to \text{Name} \to \langle \text{Sharing}, b \rangle
\]

The expression on Line 2 produces the sharing graph edges \(\text{sg} (f_{ui} \langle \{x\}, x_{ui} \rangle \{[$] \{f\})\). Because we are performing a bottom-up analysis, and do not know the value of \(f_{ui}\), we must preserve the above expression algebraically until the point that \(\text{apply}\) is itself applied. The sharing graph after Line 2 can only be represented algebraically:

\[
\text{completion } (\text{sg} (f_{ui} \langle \{x\}, x_{ui} \rangle \{[$] \{f\}))
\]

As we will be seeing a lot of expressions of this form, we introduce a function:

\[
\text{sgapply} = \text{completion} \circ \text{sg}
\]
This reduces our sharing graph expression to:

\[
\text{sgapply} \ f \ u_i \langle [x], \ x_{u_i} \rangle \ [\$] \ [f]
\]

Similarly, after Line 2, we can assign a uniqueinfo to the result variable, $, but it can only be an algebraic uniqueinfo:

\[
\$ \Rightarrow u_i \langle [x], \ x_{u_i} \rangle \ [\$] \ [f]
\]

Note that if we take the \(u_i\) of an algebraic function application, the result will be entirely independent of the names \([x], \ [\$]\) and \([f]\) (they only appear in the sharing graph part). To simplify these expressions, we introduce a function \(u_iapply\) (defined in Section 9.8.1), which applies a uniqueinfo function to a sequence of uniqueinfos, making up arbitrary names for the arguments and result, so we do not have to supply them:

\[
\$ \Rightarrow u_iapply \ f \ u_i \ x_{u_i}
\]

Hence, the final uniqueinfo for apply is:

\[
\text{apply}_a :: (\langle \text{Name}, \langle \text{Name}, \text{a} \rangle \rightarrow \text{Name} \rightarrow \text{Name} \rightarrow \langle \text{Sharing}, \text{b} \rangle \rangle, \langle \text{Name}, \text{a} \rangle) \rightarrow \\
\text{Name} \rightarrow \langle \text{Sharing}, \text{b} \rangle
\]

\[
\text{apply}_a (\langle f, f_{u_i}, \langle x, x_{u_i} \rangle \rangle \ [\$] = \langle \text{sgapply} \ f \ u_i \ x_{u_i} \rangle \$ \ f, \ u_iapply \ f \ u_i \ x_{u_i})
\]

That was too easy: this example created an algebraic sharing graph, but did not subsequently do anything with it. To make things more interesting, Example 9.8 copies \(x\) to a variable, assigns \(f\)’s result to a variable, and then copies the variable to the result.

**Example 9.8** The apply function, copying the input and result from a variable

1: \textbf{def} apply\(f :: a \rightarrow b, x :: a\):
2: \hspace{1em} \textit{y} = \textit{x}
3: \hspace{1em} \textit{r} = \textit{f(y)}
4: \hspace{1em} \textbf{return} \textit{r}

Clearly, the sharing graph after Line 2 is \(\{ \{ x, y \} \} \). The uniqueinfo for \(y\) is \(x_{u_i}\). As in the previous example, the expression on Line 3 produces the edges:

\[
q = \text{sgapply} \ f \ u_i \langle [y], \ x_{u_i} \rangle \ [r] \ [f]
\]
We name this edge set \( q \), as it will be used extensively. The sharing graph after Line 3 is:

\[
\{ \{ x, y \} \} \trianglerighteq q
\]

If we were to continue the analysis, we would keep growing this algebraic expression, with the final sharing graph being \( \{ \{ x, y \} \} \trianglerighteq q \trianglerighteq \{ \{ $, r \} \} \), but this would turn into quite a messy strategy indeed, and our answer would be virtually meaningless to a human. Instead, we break it down further to try and assign meaning to the results of the alternating closure expressions.

We return to the sharing graph after Line 3. Our goal is to factor \( q \) so that the \( \trianglerighteq \) operator can meaningfully manipulate algebraic sharing graphs. A side-goal is to allow a human to inspect an algebraic sharing graph and see what aliases might exist between any pair of nodes. To this end, we break down \( q \) into three disjoint sets:

1. both \( \{ [[r]], [[y]] \} q \): Edges between \( r \) and \( y \) (and the components thereof),
2. both \( \{ [[r]], [[f]] \} q \): Edges between \( r \) and \( f \) (and the components thereof),
3. remove \( \{ [[y]], [[f]] \} q \): All other edges involving \( r \) (and the components thereof). This includes a) aliasing between components of \( r \), b) local variables in the caller (unknown to apply), and c) global variables (aliasing from components of \( r \) to @).

Recall that the remove function projects away any edges that contain components of any of the given names, on either end. We also introduce a new function both, which takes exactly two names, \( \{ x, y \} \), and projects away any edges that do not contain a component of \( x \) on one end and a component of \( y \) on the other. Clearly, the three sets above are disjoint. Because the only edges in \( q \) involving \( [[y]] \) or \( [[f]] \) must also involve \( r \), the union of the three sets is \( q \).

We call these sets *pseudo-edges*, because each can be thought of like an edge between two known names. Unlike a real edge, however, a pseudo-edge from \( x \) to \( y \) represents unknown aliasing between arbitrary components of \( x \) and \( y \). Formally, we write pseudo-edges with the special notation:

\[
(x, y)^c
\]

This represents a pseudo-edge from \( x \) to \( y \) with modifier \( c \). In our new notation, the above pseudo-edge both \( \{ [[r]], [[y]] \} q \) is written as:

\[
([y], [r])^{c_1}, \text{ where } c_1 = \lambda x \ y. \text{ both } \{ x, y \} (\text{sgapply } f_{ui} \langle x, x_{ui} \rangle \ y \ ?))
\]

The name literal \( \? \) is a “don’t care” value that never escapes the expression. Note that
the modifier is totally independent of the actual names \([y]\) and \([r]\), so it can be used with other name pairs (and this is the key to our algebraic \(\bowtie\) operator). Similarly, our second pseudo-edge is written as:

\[
(\lbrack f \rbrack, \lbrack r \rbrack)^{c_2}
\]

where \(c_2 = \lambda x. y. \text{both} \{x, y\} (\text{sgapply f}_{\text{ui}} \langle \lbrack ? \rbrack, x_{\text{ui}} \rangle y x)\)

Our third pseudo-edge has a slightly different form. It is a self-pseudo-edge, which means that rather than being an edge between two nodes, it is applied to a single node:

\[
\lbrack r \rbrack^{c_3}
\]

where \(c_3 = \lambda x. \text{remove} \{\lbrack ?_1 \rbrack, \lbrack ?_2 \rbrack\} (\text{sgapply f}_{\text{ui}} \langle \lbrack ?_1 \rbrack, x_{\text{ui}} \rangle x \lbrack ?_2 \rbrack)\)

A self-pseudo-edge on \(x\) represents unknown aliasing between \(x\) and any names other than the closure and arguments. This includes the static data (@), hidden local variables, and between two components of \(x\) itself.

In a diagram, a pseudo-edge \((x, y)^c\) is represented as a directed edge from \(x\) to \(y\) annotated with \(c\). A self-pseudo-edge \(x^c\) is represented as a loop from \(x\) to itself annotated with \(c\).

For simple types (those with no components, such as an array of numbers), a modifier is essentially a Boolean function that denotes whether there is a real edge between the two names. For more complex types, the modifier can produce arbitrary aliasing between the components of the two names.

The sharing graph after Line 3 can now be expressed as:

\[
\{ \{x, y\} \bowtie ((y, r)^{c_1} \cup (f, r)^{c_2} \cup r^{c_3})
\]
We apply a generalised version of \( \bowtie \) designed to handle pseudo-edges. It essentially works as normal, but any new pseudo-edges have modifiers from the pseudo-edges that they were derived from. Here, we need to generate a new pseudo-edge from \( x \) to \( r \), with the modifier \( c_1 \). The resulting algebraic sharing graph (pictured in Figure 9.3b) is:

\[
\{ \{ x, y \} \} \cup (y, r)^{c_1} \cup (x, r)^{c_1} \cup (f, r)^{c_2} \cup r^{c_3}
\]

which we simplify by removing \( y \):

\[
(x, r)^{c_1} \cup (f, r)^{c_2} \cup r^{c_3}
\]

The uniqueinfo map after Line 2 is:

\[
f \Rightarrow f_{ui} \\
x \Rightarrow x_{ui} \\
r \Rightarrow uiapply f_{ui} x_{ui}
\]

Line 3 simply adds the new edge \( \{ \{ \$, r \} \} \). Using the algebraic \( \bowtie \), we derive the final sharing graph (pictured in Figure 9.3c):

\[
\{ \{ \$, r \} \} \cup (x, r)^{c_1} \cup (x, \$)^{c_1} \cup (f, r)^{c_2} \cup (f, \$)^{c_2} \cup r^{c_3} \cup \$^{c_3}
\]

which we simplify by removing \( r \):

\[
(x, \$)^{c_1} \cup (f, \$)^{c_2} \cup \$^{c_3}
\]

The uniqueinfo for \( \$ \) is copied from \( r \):

\[
\$ \Rightarrow uiapply f_{ui} x_{ui}
\]

Hence, the final uniqueinfo for apply is:

\[
apply_a ((f, f_{ui}), (x, x_{ui})) \$ = ((x, \$)^{c_1} \cup (f, \$)^{c_2} \cup \$^{c_3}, uiapply f_{ui} x_{ui})
\]

where \( c_1 = \lambda x. y. \text{both} \{x, y\} (\text{sgapply} f_{ui} \langle x, x_{ui} \rangle y \, [?]) \)

\( c_2 = \lambda x. y. \text{both} \{x, y\} (\text{sgapply} f_{ui} \langle [?], x_{ui} \rangle y \, x) \)

\( c_3 = \lambda x. \text{remove} \{[?1], [?2]\} (\text{sgapply} f_{ui} \langle [?1], x_{ui} \rangle x \, [?2]) \)

Consider, for example, a call to apply:

\[
r = apply(f, m)
\]
in a context with the following sharing graph:

\[
\{ \{f#, k\}\}
\]

and the following uniqueinfo map:

\[
m \Rightarrow m_{ui} \\
f \Rightarrow \lambda \langle z, z_{ui} \rangle \% \langle \{\{\$, \%\}\}, k_{ui}\rangle
\]

The evaluation of \(\text{apply}_a\) follows:

\[
c_1 = \lambda \ x \ y. \ \text{both} \ \{x, y\} \ \{y, [?]#\} \\
= \lambda \ x \ y. \ \emptyset
\]

\[
c_2 = \lambda \ x \ y. \ \text{both} \ \{x, y\} \ \{y, x\#\} \\
= \lambda \ x \ y. \ \{y, x\#\}
\]

\[
c_3 = \lambda \ x. \ \text{remove} \ \{[?]_1, [?]_2\} \ \{x, [?]_2\#\} \\
= \lambda \ x. \ \emptyset
\]

\[
\text{apply}_a (\langle [f], f_{ui}\rangle, \langle [m], m_{ui}\rangle) \ [r] = \langle([m], [r])^{\circ1} \cup ([f], [r])^{\circ2} \cup [r]^{\circ3}, \text{uiapply} \ f_{ui} \ m_{ui}\rangle \\
= \langle\{\{r\}, [f]\#\}\}, k_{ui}\rangle
\]

Combining this with the existing sharing graph results in the graph:

\[
\{ \{f#, k\}, \{r, f\#\}, \{r, k\}\}
\]

with \(r\) having uniqueinfo \(k_{ui}\). This demonstrates how a call to an algebraic higher-order function with concrete arguments (i.e., a known function) results in a nice simple first-order result, without any loss of precision.

### 9.6.2 compose

This next example is an extension of \(\text{apply}\) that demonstrates the possibility of a pseudo-edge that is the intersection of two (or more) applications. Example 9.9 is the closure template used by the \(\text{compose}\) function — it takes two functions \(f\) and \(g\) and applies them, in reverse order, to the value \(x\). The \(\text{compose}\) function simply reifies this closure template with \(f\) and \(g\), producing a function of type \(a \rightarrow c\), but for this example, we just focus on the closure template.

---

6This describes the state after \(f\) is assigned \(\text{const}(k)\).
Example 9.9 The closure template used by the compose function

1: `def compose(f :: b → c, g :: a → b)(x :: a):`
2: `t = g(x)`
3: `return f(t)`

We begin the analysis with the ordinary uniqueinfo map:

\[ f \Rightarrow f_{ui} \]
\[ g \Rightarrow g_{ui} \]
\[ x \Rightarrow x_{ui} \]

After Line 2, we have the uniqueinfo for \( t \):

\[ t \Rightarrow \text{uiapply } g_{ui} x_{ui} \]

and we have the following pseudo-edges (visualised in Figure 9.4a):

\[(x, t)^c_1 \cup (g, t)^c_2 \cup t^c_3\]
\[\text{where } c_1 = \lambda x y. \text{both } \{x, y\} (\text{sgapply } g_{ui} \langle x, x_{ui} \rangle y [?])\]
\[ c_2 = \lambda x y. \text{both } \{x, y\} (\text{sgapply } g_{ui} \langle [?], x_{ui} \rangle y x)\]
\[ c_3 = \lambda x. \text{remove } \{[?1], [?2]\} (\text{sgapply } g_{ui} \langle [?1], x_{ui} \rangle x [?2])\]

After Line 3, we have the uniqueinfo for \( $ \):

\[ $ \Rightarrow \text{uiapply } f_{ui} (\text{uiapply } g_{ui} x_{ui}) \]

Line 3 introduces the following new sharing graph edges (visualised in Figure 9.4b):

\[(t, $)^c_4 \cup (f, $)^c_5 \cup $^c_6\]
\[\text{where } c_4 = \lambda x y. \text{both } \{x, y\} (\text{sgapply } f_{ui} \langle x, \text{uiapply } g_{ui} x_{ui} \rangle y [?])\]
\[ c_5 = \lambda x y. \text{both } \{x, y\} (\text{sgapply } f_{ui} \langle [?], \text{uiapply } g_{ui} x_{ui} \rangle y x)\]
\[ c_6 = \lambda x. \text{remove } \{[?1], [?2]\} (\text{sgapply } f_{ui} \langle [?1], \text{uiapply } g_{ui} x_{ui} \rangle x [?2])\]

As in the apply example, adding these edges into the existing edges (via the \( \bowtie \) operator) requires propagation of pseudo-edges. However, in this example, we see new pseudo-edges derived from more than one existing pseudo-edge. The resulting modifier is the alternating closure of the source pseudo-edges’ modifiers (visualised in Figure 9.4c):

\[(x, t)^c_1 \cup (g, t)^c_2 \cup t^c_3 \cup (t, $)^c_4 \cup (f, $)^c_5 \cup $^c_6 \cup (x, $)^c_{1\bowtie 4} \cup (g, $)^c_{2\bowtie 4} \cup $^c_{3\bowtie 4}\]
Here we see pseudo-edges with modifiers such as $c_1 \mathbin{\bowtie} c_4$; this will perform the equivalent alternating closure over the real edges, once the pseudo-edges are resolved. The final uniqueinfo for \texttt{compose}$_2$ is:

\[
\text{compose}_2 \left( (f \, \text{ui}, (g, \text{ui}) \, (x, x_{ui}) \, \text{ui} \right) =
\left( (x, \text{ui}) \, c_1 \mathbin{\bowtie} (g, \text{ui}) \, c_2 \, c_3 \, c_4 \, c_5 \, c_6 \, \text{ui} \right)
\]

\[\text{where } c_1 = \lambda x y. \text{both } \{x, y\} \text{ (sgapply } g_{ui} \langle x, x_{ui} \rangle \, y \, [?]\text{)}
\]
\[c_2 = \lambda x y. \text{both } \{x, y\} \text{ (sgapply } g_{ui} \langle [?], x_{ui} \rangle \, y \, x)\]
\[c_3 = \lambda x. \text{remove } \{[?], [?]\} \text{ (sgapply } g_{ui} \langle [?], x_{ui} \rangle \, x \, [?]\}
\]
\[c_4 = \lambda x y. \text{both } \{x, y\} \text{ (sgapply } f_{ui} \langle x, \text{uiapply } g_{ui} x_{ui} \rangle \, y \, [?])\]
\[c_5 = \lambda x y. \text{both } \{x, y\} \text{ (sgapply } f_{ui} \langle [?], \text{uiapply } g_{ui} x_{ui} \rangle \, y \, x)\]
\[c_6 = \lambda x. \text{remove } \{[?], [?]\} \text{ (sgapply } f_{ui} \langle [?], \text{uiapply } g_{ui} x_{ui} \rangle \, x \, [?]\}
\]

This is, no doubt, a complicated result, and difficult for a human to parse. Fortunately, most of the commonly used higher-order functions use only a single function variable.

### 9.6.3 Polymorphic recursion

A nasty edge case occurs when a recursive or iterative function builds a closure by repeatedly redefining it in terms of its previous value. Consider Example 9.10:

**Example 9.10** Repeatedly redefining a closure

```python
def apply_1(f :: a → b)(x :: a):
    return f(x)
def wrap_f(f :: a → b, i :: Num):
    if i ≤ 0:
        return f
    else:
        i' = i - 1
        f' = wrap_f(f, i')
    return apply_1(f')
```
9.6. THE APPLICATIVE DOWNWARD HIGHER-ORDER PROBLEM 217

Semantically, the \texttt{wrap}_f function just returns \( f \). However, operationally, it builds \( i \) nested closures, each passing its argument down to the next closure before finally calling \( f \). Ideally, we could determine that the uniqueinfo of the result of \texttt{wrap}_f is \( f_{ui} \). However, we cannot, as a dutiful application of the analysis rules does not terminate.

On the first iteration, the analysis for \texttt{wrap}_f gives:

\[
\text{wrap}_f \alpha (\langle f, f_{ui} \rangle, \langle i, i_{ui} \rangle) \Downarrow = \{ \{\$, f\}, \{\$, f\} \} \cup (f, \$#)_{c_2} \ni \lambda \langle x, x_{ui} \rangle \% \ni \{\}, \bot \}
\]

On the second iteration, the analysis for \texttt{wrap}_f gives:

\[
\text{wrap}_f \alpha (\langle f, f_{ui} \rangle, \langle i, i_{ui} \rangle) \Downarrow = \begin{cases} \\
\{\{\$, f\}, \{\$, f\} \cup (f, \$#)^2, \\
 f_{ui} \cup \lambda \langle x, x_{ui} \rangle \% \ni \langle x, \$, \$\rangle_{c_2} \cup \{\}, \bot \}
\end{cases}
\]

(\text{The modifiers} \ c_1, c_2 \text{ and} \ c_3 \text{ are omitted for brevity.)}

Unfortunately, we can see a pattern emerging in which each iteration’s uniqueinfo features a lambda containing the uniqueinfo of the result of the previous iteration. This analysis will grow without end!

To be clear, this is not a problem with the UniqueInfo data structure defined in our abstract semantics — the uniqueinfo lattice for a given type has a finite height and so our non-algebraic analysis is guaranteed to terminate. This is a problem with the algebraic uniqueinfo we have introduced in order to deal with downward higher-order functions in a bottom-up analysis. By introducing algebraic terms such as \( \sqcup \) and \texttt{uiapply}, we have made it possible to construct arbitrarily tall uniqueinfos.

Fortunately, functions such as this are rare in practice, so we can afford to lose some precision in these circumstances (but we must still ensure the analysis terminates). To this end, we explicitly \textit{widen} the result when part of a uniqueinfo corresponding to a specific node in the type tree grows to a sufficient height; that part of the uniqueinfo is simply set to \( \top \), assuming full aliasing as we did for all closure applications in Chapter 8. For this example, we terminate with:

\[
\text{wrap}_f \alpha (\langle f, f_{ui} \rangle, \langle i, i_{ui} \rangle) \Downarrow = \{ \{\$, f\}, \{\$, f\} \cup (f, \$#)^2, \top \}
\]

This widening is discussed further in Section 9.8.6.
CHAPTER 9. HIGHER-ORDER UNIQUENESS INFERENCE

9.7 Abstract memory denotational semantics

In this section, we “patch” the semantics given in Section 8.3 to formally specify how upward higher-order functions may be precisely analysed. Note that the semantics only directly addresses the upward higher-order problem as described above. The downward higher-order problem is not a problem with the semantics, but rather with the context-insensitive interpretation of functions. These will be addressed formally in a later section.

9.7.1 Abstract cell names

We add a new type of name, the result-data node introduced in Section 9.4.1.

\[
\text{Name} \rightarrow \text{Var} \\
\quad | \quad \text{Name} \cdot \text{Int} \\
\quad | \quad \text{Name}[] \\
\quad | \quad \text{Name}() \\
\quad | \quad \text{Name#} \\
\quad | \quad @
\]

The name \(x#\) (“\(x\)’s result-data”) is a placeholder node that has the same type as the result of the function \(x\). It is used to represent the memory cells that will be aliased to the result of calling the function \(x\).\(^7\) This is commonly applied to the \% uniqueinfo parameter (which represents the closure having that uniqueinfo): anything aliased to \%# (“my-result”) will be aliased to the result of the current function.

We introduce a new clause in the \text{names} function defined in Section 8.3.1 to handle this new type of name:

\[
\text{names} : \text{State} \rightarrow \text{Name} \rightarrow \wp(\text{Ref})
\]

\[
\text{names} (\eta, \sigma) [x#] = \left\{ r : \\
\quad r_0 \in \text{names} (\eta, \sigma) [x] \land \\
\quad (c, f) = \eta r_0 \land \\
\quad \text{args} \in \text{Val}^* \land \\
\quad (r, \eta') = f \eta (\eta c) \text{ args} \right\}
\]

This rather complex looking rule simply states that, assuming \(x\) is a function (which it must be if it has a # child), \(r\) is any value that could possibly be the result of applying \(x\) to any arguments. The resulting set is all possible references \(r\).

\(^7\)The function application semantics do not automatically alias the result of \(x\) with \(x#\); this will only happen if \(x\)’s uniqueinfo includes \{$, %#\}. 

9.7. ABSTRACT MEMORY DENOTATIONAL SEMANTICS

9.7.2 Uniqueinfos

In the first-order analysis, we described the domain Func_\alpha and related domains as “unique-infos.” As we have seen in this chapter, the definition of a uniqueinfo has now expanded to represent an abstract value for any concrete value. We formally define uniqueinfos in this section.

Rather than defining uniqueinfos as a set of values, we define a “set abstraction” function, \( \alpha \), that denotes, for a concrete type \( T \), the corresponding set of uniqueinfos:

\[
\alpha : \text{Type} \rightarrow \wp(\text{UniqueInfo})
\]

\( \alpha' : \text{Type}^* \rightarrow \text{Type} \rightarrow \wp(\text{UniqueInfo}) \)

\( \alpha'' : \text{Type}^* \rightarrow \text{Type} \rightarrow \wp(\text{UniqueInfo}) \)

\[
\alpha t = \alpha' \langle t \rangle t
\]

\[
\alpha' ts ((a_1, \ldots, a_n) \rightarrow r)] = (\langle \text{Name}, \alpha'' ts a_1 \rangle \times \cdots \times \langle \text{Name}, \alpha'' ts a_n \rangle) \rightarrow \\
\text{Name} \rightarrow \text{Name} \rightarrow (\text{Sharing}, \alpha'' ts r)
\]

\[
\alpha' ts [\text{Num}] = \{()\}
\]

\[
\alpha' ts [\text{Array}(a)] = \{[u] : u \in \alpha'' ts a\}
\]

\[
\alpha' ts [\text{usertype}] = \text{let } e_1, \ldots, e_n = \text{fields usertype in } \alpha'' ts e_1 \times \cdots \times \alpha'' e_n
\]

\[
\alpha'' \langle t_0, \ldots, t_n \rangle s = \begin{cases} 
\{^n+1−i\}, & \text{if } 0 \leq i \leq n \land s = t_i \\
\alpha' \langle t_0, \ldots, t_n, s \rangle s, & \text{otherwise}
\end{cases}
\]

where fields denotes the types of all fields of a given user-defined type, flattened, in order of constructor.

Collectively, these sets are the uniqueinfos. They record the aliasing behaviour of first-class functions, parameterised by the aliasing behaviour of the input, and recording the aliasing behaviour of functions nested within arrays, user-defined types, and the results of functions. For completeness, we define the full set UniqueInfo (which is used only for closures, not global functions or closure templates):

\[
\text{UniqueInfo} = \langle \text{Name}_\perp, \text{UniqueInfo} \rangle^* \rightarrow \text{Name} \rightarrow \text{Name} \rightarrow (\text{Sharing}, \text{UniqueInfo})
\]

\[
\cup \{[u] : u \in \text{UniqueInfo}\}
\]

\[
\cup \text{UniqueInfo}^*
\]

\[
\cup \{^i : i \in \mathbb{N} \land i > 0\}
\]
Uniqueinfos are \textit{type folded} such that fields that have the same type as any ancestor in the type graph are joined with the same-typed ancestor, and replaced with a $\uparrow^n$ symbol (denoting \(n\) levels up to the same-type ancestor). For example, a List\((a)\) has a uniqueinfo in \(a(a) \times \{\uparrow}\), with the second element never having any value other than $\uparrow$.

The domain UniqueInfo forms a set of lattices, with tuples ordered element-wise and functions ordered pointwise. Specifically:

\[
\begin{align*}
  f & \subseteq g \iff \forall as \% . (f as \%) \subseteq (g as \%)
  \\
  [f] & \subseteq [g] \iff f \subseteq g
  \\
  (f_1, \ldots, f_n) & \subseteq (g_1, \ldots, g_n) \iff f_1 \subseteq g_1 \land \cdots \land f_n \subseteq g_n
  \\
  \langle sh_1, ui_1 \rangle & \subseteq \langle sh_2, ui_2 \rangle \iff sh_1 \subseteq sh_2 \land ui_1 \subseteq ui_2
  \\
  f \sqcup g & = \lambda as \% . (f as \%) \sqcup (g as \%)
  \\
  [f] \sqcup [g] & = [f \sqcup g]
  \\
  (f_1, \ldots, f_n) \sqcup (g_1, \ldots, g_n) & = (f_1 \sqcup g_1, \ldots, f_n \sqcup g_n)
  \\
  \langle sh_1, ui_1 \rangle \sqcup \langle sh_2, ui_2 \rangle & = \langle sh_1 \sqcup sh_2, ui_1 \sqcup ui_2 \rangle
\end{align*}
\]

Note that uniqueinfos are only compatible if they share the same concrete type (which is why the domain forms a set of lattices and not a single lattice). For example, the concrete type \(\text{Num} \rightarrow \text{Num}\) corresponds to a uniqueinfo lattice whose bottom is:

\[
\lambda \langle x, x_{ui} \rangle \% . (\{\}, ())
\]

and whose top is:

\[
\lambda \langle x, x_{ui} \rangle \% . (\{\{\$, x\}, \{$, @\}\}, ())
\]

The concrete type \(\text{Num}\) corresponds to a uniqueinfo lattice whose bottom and top are (\()\).

It is now shown that each of the individual uniqueinfo lattices is finite. We make the observation that uniqueinfo functions do not inspect the values of the supplied names; they only use them verbatim in the output. Thus, despite accepting an infinite number of possible names, a uniqueinfo function in:

\[
\langle \text{Name}_\bot, \text{UniqueInfo} \rangle^* \rightarrow \text{Name} \rightarrow \text{Name} \rightarrow \langle \text{Sharing}, \text{UniqueInfo} \rangle
\]

is isomorphic to a function in:

\[
\text{UniqueInfo}^* \rightarrow \langle \text{Sharing}, \text{UniqueInfo} \rangle
\]

with an implicit name in place of each name parameter. If we assume that UniqueInfo is
a finite set, then this gives us a function mapping finite inputs to finite outputs. Hence
the function space is finite.

9.7.3 The abstract domain

We extend the abstract domain of the previous chapter to include the uniqueinfo map. This means that during the course of analysing a function, we explicitly assign a unique-
info to each variable.

\[
\begin{align*}
\text{State}_\alpha &= \langle \text{Sharing}, \text{UIMap} \rangle \\
\text{Sharing} &= \wp(\text{Name} \otimes \text{Name}) \\
\text{UIMap} &= \text{Var} \mapsto \text{UniqueInfo}
\end{align*}
\]

The new State\(_\alpha\) domain, like the UniqueInfo domain, forms a set of lattices, such that
any two states are compatible (part of the same lattice) if their variables have the same
types. This means that any two states generated by the same function are compatible.

Given that each uniqueinfo lattice has a finite height, the State lattice for the fixed set
of variables in a given function also has a finite height.

We also redefine the function domains to include uniqueinfos:

\[
\begin{align*}
\text{Func}_\alpha &= \langle \text{Name}_\perp, \text{UniqueInfo} \rangle^* \rightarrow \text{Name} \rightarrow \langle \text{Sharing}, \text{UniqueInfo} \rangle \\
\text{CT}_\alpha &= \langle \text{Name}_\perp, \text{UniqueInfo} \rangle^* \rightarrow \langle \text{Name}_\perp, \text{UniqueInfo} \rangle^* \rightarrow \text{Name} \rightarrow \\
& \quad \langle \text{Sharing}, \text{UniqueInfo} \rangle \\
\text{CGC}_\alpha &= \text{Name} \rightarrow \langle \text{Sharing}, \text{UniqueInfo} \rangle \\
\text{Den}_\alpha &= (\text{CGCName} \cup \text{FuncName} \cup \text{CTName}) \mapsto (\text{CGC}_\alpha \cup \text{Func}_\alpha \cup \text{CT}_\alpha)
\end{align*}
\]

9.7.4 Abstraction and concretisation

Recall from Section 8.3.3 that the abstraction (\(\alpha\)) and concretisation (\(\gamma\)) functions are de-
defined in terms of \(\beta : \text{State} \rightarrow \text{State}_\alpha\), as follows:

\[
\begin{align*}
\alpha : \wp(\text{State}) &\rightarrow \text{State}_\alpha \\
\gamma : \text{State}_\alpha &\rightarrow \wp(\text{State})
\end{align*}
\]
\[ \begin{align*}
\alpha S &= \bigsqcup_{s \in S} \beta s \\
\gamma g &= \{ s : \beta s \sqsubseteq g \}
\end{align*} \]

In this section, we redefine \( \beta \) with respect to the higher-order state model. We already have a \( \beta \) function that produces a sharing graph from a concrete state, from Section 8.3.3:

\[ \beta_{\text{sh}} : \text{State} \to \text{Sharing} \]

\[ \beta_{\text{sh}} s = \{ \{ x, y \} : \mu x \cap \mu y \neq \emptyset \} \]

where \( \mu = \text{names } s \)

We now define a \( \beta \) function that takes a memory heap and concrete value and produces a corresponding uniqueinfo:

\[ \beta_{\text{ui}} : \text{Heap} \to \text{Val}_{ \perp } \to \text{UniqueInfo} \]

\[ \beta_{\text{obj}} : \text{Heap} \to \text{Object} \to \text{UniqueInfo} \]

\[ \begin{align*}
\beta_{\text{ui}} \eta \perp &= \perp \\
\beta_{\text{ui}} \eta n &= (\), \text{ if } n \in Q \\
\beta_{\text{ui}} \eta r &= \beta_{\text{obj}} \eta (\eta r), \text{ if } r \in \text{Ref} \\
\beta_{\text{obj}} \eta (c, e_1, \ldots, e_n) &= (\perp, \ldots, \perp, \beta_{\text{ui}} \eta e_1, \ldots, \beta_{\text{ui}} \eta e_n, \perp, \ldots, \perp) \\
\beta_{\text{obj}} \eta [e_1, \ldots, e_n] &= [\beta_{\text{ui}} \eta e_1 \sqcup \cdots \sqcup \beta_{\text{ui}} \eta e_n] \\
\beta_{\text{obj}} \eta (c, f) &= \lambda (\langle a_1, a_{ui1} \rangle, \ldots, \langle a_n, a_{uin} \rangle) \$. \langle \text{sg}, \text{ui} \rangle \\
& \quad \text{where } R = \\
& \quad \{ (\eta'', \sigma, r) : \eta' \in \text{Heap} \land \\
& \quad \quad v_1 \sqsupseteq \beta_{\text{ui}} \eta' a_{ui1} \land \\
& \quad \quad \cdots \land \\
& \quad \quad v_n \sqsupseteq \beta_{\text{ui}} \eta' a_{uin} \land \\
& \quad \quad (\eta'', r) = \eta' (\eta' c) (v_1, \ldots, v_n) \land \\
& \quad \quad \sigma = \{ a_1 \mapsto v_1, \ldots, a_n \mapsto v_n, \$ \mapsto r \} \}
\end{align*} \]

where \( \text{afterfields}(c) \) is the number of fields that appear in all subsequent constructors after \( c \) in the same type (similar to \( \text{firstfield}(c) \)).

The final clause of \( \beta_{\text{obj}} \) is fairly complex, as it denotes the uniqueinfo of a concrete clo-
sure. It computes a function that takes input names $a_1 \ldots a_n$ and corresponding unique-infos $a_{ui1} \ldots a_{uin}$, as well as the result name $. For all possible heaps $\eta'$, for all possible argument values $v_1 \ldots v_n$ (that must satisfy the input unique-infos), it computes the resulting heap $\eta''$, the environment of the input and result names $\sigma$, and the resulting concrete value $r$. It then computes the sharing graph, which is the lub of $\beta_{sh}$ of all possible output states, and the uniqueinfo, which is the lub of $\beta_{ui}$ of all possible result values. Note that this does not make use of the supplied heap $\eta$, because the closure needs to be considered in the context of a future heap where it will be applied ($\eta'$), not the current heap.

Hence, it follows that the $\beta$ function for the full state is:

$$
\beta : \text{State} \rightarrow \text{State}_\alpha
$$

$$
\beta (\eta, \sigma) = \langle \beta_{sh} (\eta, \sigma), \lambda v. \beta_{ui} \eta (\sigma v) \rangle
$$

### 9.7.5 Abstract semantics

This section presents the abstract semantics, derived from those in Section 8.3.6, augmented with support for higher-order functions.

$$
\Theta_\alpha = \{ f \mapsto \text{abstract semantics of } f : 
\begin{align*}
\text{f is the name of a primitive function}
\end{align*}
$$
\[ A_a[n] \sigma = \langle \bot, () \rangle, \quad \text{if } n \in \text{Num} \]
\[ A_a[v] \sigma = \langle [v], \sigma v \rangle, \quad \text{if } v \in \text{Var} \]

\[ E_a v[a \in \text{Atom}] \rho \sigma = \text{let } (n, u) = A_a[a] \sigma \text{ in } \langle s \{ \{ [v], n \} \}, u \rangle \]
\[ E_a v[c \in \text{CGCName}] \rho \sigma = (\tilde{p} [c]) v \sqcup \langle s \{ \{ [v], @ \} \}, \bot \rangle \]
\[ E_a v[s \cdot c / n] \rho \sigma = \langle s \{ \{ [v], [s] . (n + \text{field } c) \} \}, (\sigma s)_{(n + \text{field } c)} \rangle \]

\[ \text{let } ((n_1, u_1), \ldots, (n_n, u_n)) = \]
\[ E_a v[a_1, \ldots, a_n] \rho \sigma = \langle s \{ \{ [v], [a_1], \ldots, [v], [n] \} \}, \rangle \]
\[ \text{let } ((n_1, u_1), \ldots, (n_n, u_n)) = \]
\[ E_a v[c(a_1, \ldots, a_n) \text{ hint}] \rho \sigma = \langle s \{ \{ [v], ([\text{field } c], n_1), \ldots, [v], (n + \text{ field } c), n_n \} \}, \rangle \]
\[ E_a v[f \in \text{Var}(a_1, \ldots, a_n)] \rho \sigma = (\sigma f) (A_a[a_1] \sigma, \ldots, A_a[a_n] \sigma) [v] [f] \]
\[ E_a v[f \in \text{FuncName}(a_1, \ldots, a_n)] \rho \sigma = (\tilde{p} f) (A_a[a_1] \sigma, \ldots, A_a[a_n] \sigma) [v] \]
\[ E_a v[f \{ a_1, \ldots, a_n \}] \rho \sigma = \text{reify } v (\tilde{p} f) (A_a[a_1] \sigma, \ldots, A_a[a_n] \sigma) \]

where field = firstfield

\[ S_a[e] \rho \sigma = \langle \emptyset, \emptyset \rangle \]
\[ S_a[s_1 ; s_2] \rho \sigma = \langle s_2, s_1 \rangle \rho \sigma \]
\[ S_a[v = e] \rho \sigma = \text{let } (\eta_1, \sigma_1) = S_a[s_1] \rho \sigma \text{ in } \langle \eta_1 \triangleright \eta_2, \sigma_1 \cup \sigma_2 \rangle \]
\[ S_a[\text{if } a: s \text{ then } else: s \text{ else}] \rho \sigma = S_a[s_{\text{then}}] \rho \sigma \sqcup S_a[s_{\text{else}}] \rho \sigma \]
\[ S_a[\text{switch } v; (c_1 : s_1, \ldots, c_n : s_n)] \rho \sigma = S_a[s_1] \rho \sigma \sqcup \cdots \sqcup S_a[s_n] \rho \sigma \]
\[ S_a[\text{while } a [u_1 = \phi(v_1, w_1), \ldots]: s] \rho \sigma = \text{lfp } \lambda(\eta, \sigma'). \]

\[ \text{let } \sigma_1 = \sigma'[u_1 \mapsto \sigma v_1 \sqcup \sigma' u_1, \ldots] \]
\[ \langle \eta_1, \sigma_2 \rangle = S_a[s] \rho \sigma_1 \]
\[ \eta_2 = \eta \cup \{ u_1, v_1 \}, \ldots \triangleright \eta_1 \]
\[ \eta_3 = \text{remove } u_1, \ldots \eta_2 \]
\[ \eta_4 = \eta_3 \triangleright \{ u_1, w_1 \}, \ldots \]
\[ \sigma_3 = \sigma_2[u_1 \mapsto \sigma_2 w_1, \ldots] \]
\[ \text{in } \langle \text{remove } (\text{assumed } s) \rangle \eta_4, \sigma_3 \]
9.7. ABSTRACT MEMORY DENOTATIONAL SEMANTICS

\[ F_\alpha[f \ s \ e] = \{ f \mapsto \lambda r. \}
\]
\[ \text{let } \langle \eta, r_{ui} \rangle = \text{procbody } [s][e] r \rho \odot \odot \]
\[ \text{in } \langle \text{keep } \{ r \} \eta, r_{ui} \rangle \]

\[ F_\alpha[f(a_1, \ldots, a_n) \ s \ e] = \{ f \mapsto \lambda (x_1, x_{ui1}), \ldots, (x_n, x_{uin}) \ r. \}
\]
\[ \text{let } \eta = s \{ \{ a_1, x_1 \}, \ldots, \{ a_n, x_n \} \}
\]
\[ \sigma = \{ a_1 \mapsto x_{ui1}, \ldots, a_n \mapsto x_{uin} \}
\]
\[ \langle \eta', r_{ui} \rangle = \text{procbody } [s][e] r \rho \eta \sigma \]
\[ \text{in } \langle \text{keep } \{ r, x_1, \ldots, x_n \} \eta', r_{ui} \rangle \]

\[ F_\alpha[f\{v_1, \ldots, v_n\}(a_1, \ldots, a_n) \ s \ e] = \{ f \mapsto \lambda ((w_1, w_{ui1}), \ldots, (w_n, w_{uin})) \}
\]
\[ ((x_1, x_{ui1}), \ldots, (x_n, x_{uin})) \ r. \]
\[ \text{let } \eta = s \{ \{ v_1, w_1 \}, \ldots, \{ v_n, w_n \} \}
\]
\[ \{ a_1, x_1 \}, \ldots, \{ a_n, x_n \} \}
\]
\[ \sigma = \{ v_1 \mapsto w_{ui1}, \ldots, v_n \mapsto w_{uin} \}
\]
\[ \{ a_1 \mapsto x_{ui1}, \ldots, a_n \mapsto x_{uin} \}
\]
\[ \langle \eta', r_{ui} \rangle = \text{procbody } [s][e] r \rho \eta \sigma \]
\[ \text{in } \langle \text{keep } \{ r, x_1, \ldots, x_n, w_{ui1}, \ldots, w_{uin} \} \eta', r_{ui} \rangle \}

\[ \mathcal{P}_\alpha[p] = \text{lfp } \bigg( \bigcup_{f \in p} (F_\alpha f) \sqcup \lambda r. \Theta \bigg) \]

\[ \text{procbody } [s][e] r \rho \eta \sigma \]
\[ = \langle \eta', \sigma' \rangle = S_\alpha[s] \rho \sigma \]
\[ \langle \eta_r, r_{ui} \rangle = \mathcal{E}_\alpha[r][e](\sigma \cup \sigma') \]
\[ \text{in } \langle \eta \bowtie \eta' \bowtie \text{completion } \eta_r, r_{ui} \rangle \}

The semantics looks considerably more complex than it did in the previous chapter, but it boils down to essentially two changes:

1. Everything has been augmented to deal with uniqueinfos in addition to sharing graphs. For example, the variable atom semantics was \( [v] \), but is now \( \langle [v], \sigma v \rangle \).

2. The rules for variable application and closure template reification now precisely describe the aliasing, rather than assuming the worst in both cases. These new rules are discussed in the next section.

9.7.6 Reify

The reify function implements the semantics for a closure template reification statement:
reify : \( \text{Var} \rightarrow \text{CT}_\alpha \rightarrow \langle \text{Name}, \text{UniqueInfo} \rangle^* \rightarrow \text{UniqueInfo} \)

\[ \text{reify } c \ f \ (\langle x_1, u_1 \rangle, \ldots, \langle x_n, u_n \rangle) = \langle \eta', \lambda (\langle y_1, v_1 \rangle, \ldots, \langle y_m, v_m \rangle) \% \langle \eta'', r_{ui} \rangle \rangle \]

where

\[ \lambda (\langle y_1, v_1 \rangle, \ldots, \langle y_m, v_m \rangle) \% \langle \eta, r_{ui} \rangle = f (\langle x_1, u_1 \rangle, \ldots, \langle x_n, u_n \rangle) \]

\[ \eta' = \{ \{c(), x_1\}, \ldots, \{c(), x_n\} \} \cup \]

\[ \eta'' = \{ \{k \% #, q\}, \text{if } p = z \ x_i \land q = k \% \}

\[ \text{otherwise} \}

\[ \{p, q\} \in \eta \land 0 \leq i < n \land \]

\[ z \in \text{descendants (type } x_i) \land \]

\[ k \in \text{descendants (type } \$) \land \]

\[ p = z \ x_i \land q = k \% \}

\[ \{p, q\} \in \eta \land 0 \leq i < n \land \]

\[ z \in \text{descendants (type } x_i) \land \]

\[ k \in \text{descendants (type } \$) \}

The reify function is hideously complex, but essentially just takes care of binding closure variables to the closure’s () and # nodes, and creating the closure’s uniqueinfo based on that of the closure template.

Informally, we partially apply the closure template’s uniqueinfo, and then find two sharing graphs: \( \eta' \), which contains the new edges that will be created immediately as a result of reifying the closure template, and \( \eta'' \), which contains the edges that will be created by applying the closure. Assuming that \( \eta \) is the sharing graph given by partially applying the closure template:

- To compute \( \eta' \), take each edge in \( \eta \) between \$ and a closure variable and replace \$ with \( c\# \). Ignore any other edges in \( \eta \). Also alias \( c() \) to each closure variable.

- To compute \( \eta'' \), take \( \eta \), and for each edge between \$ and a closure variable, replace the variable with the relevant component of \% #. Leave all other edges unchanged.

For example, consider a closure template \( f \) with the following uniqueinfo:

\[ \rho \ f = \lambda \langle w, w_{ui} \rangle \langle x, x_{ui} \rangle \% \langle \{\{0.0, w.1\}, \{0.2, x.3\}\} \rangle \langle w_{ui1}, \bot, x_{ui3} \rangle \]

If we encounter the reification statement “\( v = f[a] \)”, this will lead us to evaluate:

\[ \text{reify } [v] \ (\rho \ f) \langle [a], a_{ui} \rangle \]

Note the somewhat strange notation in the definition of reify — we “pattern match” on the partially applied CT\(_\alpha\), binding the parts of the lambda to the variables \( y_1, \ldots, y_m, \)

\( v_1, \ldots, v_m, \% \), \( \$ \), \( r_{ui} \). While such a binding would normally be uncomputable, we can do this in the compiler because the CT\(_\alpha\) is stored as a lambda with a fixed syntax.
Therefore, we learn that:

\[
\eta = \{\{$.0, a.1\}, \{$.2, x.3\}\}
\]

\[
r_{ui} = (a_{ui1}, \perp, x_{ui3})
\]

When computing \(\eta'\) and \(\eta''\), note that the set builder naturally matches \(\{p, q\}\) against each edge in the sharing graph in both directions. We evaluate the remainder of \(\text{reify}\) as follows:

\[
\eta' = \{\{v(), a\}\} \cup \{\{v#.0, a.1\}\}
\]

\[
\eta'' = \{\{$.0, % #.0\}, \{$.2, x.3\}\}
\]

\[
\text{reify } [v] (\overline{p} \ f) \langle [a], a_{ui} \rangle = \left\langle \{\{v(), a\}, \{v#.0, a.1\}\}, \lambda \langle x, x_{ui} \rangle \% . \langle\{\{$.0, % #.0\}, \{$.2, x.3\}\}, (a_{ui1}, \perp, x_{ui3})\rangle \right\rangle
\]

This gives us the resulting sharing graph edges between \(v\) and \(a\), as well as the uniqueinfo of the closure \(v\). Note that the new sharing graph edge \(\{v#.0, a.1\}\) is derived from the closure template’s sharing graph edge \(\{$.0, a.1\}\), substituting \$ with \(v#\). Similarly, in the uniqueinfo, the edge \(\{$.0, % #.0\}\) is derived from that same edge, substituting \(a.1\) with \#.0 (which was aliased to \(a.1\) in the sharing graph).

### 9.8 Algebraic analysis

Unlike our first-order analysis, the higher-order semantics is non-condensing (Definition 6.5), which means that we can no longer be oblivious to the function’s inputs if we want a context-insensitive analysis.

Although statement semantics is still computed independently of the existing sharing graph state, we now require information about the uniqueinfos of existing variables when analysing a statement. For a function, we need to know the uniqueinfos of its arguments. A counter-example of Definition 6.5 is \(S_a \ [x = f()]\), which gives an undefined result if given a uniqueinfo map of \(\emptyset\) instead of a uniqueinfo map containing \(f\). Therefore, the semantics is not condensing.

In Section 6.6, we gave a number of options for context-insensitive analyses of non-condensing semantics. We could weaken the semantics so that functions do not have access to a uniqueinfo map, and assume all arguments are \(\top\). That would at least make the function semantics condensing and allow a bottom-up analysis, but would only be sufficient to handle the upward higher-order problem described in Section 9.4. To handle the non-applicative and applicative downward higher-order cases, we develop an
algebraic framework to precisely represent unknown information whilst performing a context-insensitive analysis of a function.

### 9.8.1 Algebraic uniqueinfos

An algebraic uniqueinfo (UniqueInfo\(_\text{alg}\)) is the uniqueinfo data structure with additional alternatives for algebraically representing manipulations of uniqueinfos.

\[
\text{UniqueInfo}_{\text{alg}} = \langle \text{Name}_\bot, \text{UniqueInfo}_{\text{alg}} \rangle^* \rightarrow \text{Name} \rightarrow \text{Name} \rightarrow
\langle \text{Sharing}_{\text{alg}}, \text{UniqueInfo}_{\text{alg}} \rangle
\cup \{[u] : u \in \text{UniqueInfo}_{\text{alg}} \}
\cup \text{UniqueInfo}_{\text{alg}}^*
\cup \{ \downarrow^i : i \in \mathbb{N} \land i > 0 \}
\cup \text{UIVar}
\cup \{ u_i : u \in \text{UniqueInfo}_{\text{alg}} \land i \in \mathbb{N} \}
\cup \{ u[] : u \in \text{UniqueInfo}_{\text{alg}} \}
\cup \{ \text{uiapply} \ u (a_1, \ldots, a_n) : \{ u, a_1, \ldots, a_n \} \subseteq \text{UniqueInfo}_{\text{alg}} \}
\cup \{ u_1 \sqcup \cdots \sqcup u_n : \{ u_1, \ldots, u_n \} \subseteq \text{UniqueInfo}_{\text{alg}} \}
\]

The first three alternatives are the “definite” (non-algebraic) uniqueinfo alternatives: abstract functions, abstract arrays, and abstract tuples, respectively. The remaining alternatives represent algebraic uniqueinfos.

We do not define the set UIVar; it represents the name of a uniqueinfo variable in the current context. Its implementation is non-trivial, as the implementation must deal with matters such as variable capture (having the same variable name used in two contexts), requiring that variables be alpha-renamed during various operations. This is outside of the scope of this thesis, as it is a standard problem involved in implementing the lambda calculus. For the purpose of this analysis, we consider a UIVar to be an opaque symbol representing a uniqueinfo variable name.

Once we have uniqueinfo variables, all of the basic operations on uniqueinfos might have to be stored algebraically. The remaining uniqueinfo alternatives represent these operations on algebraic uniqueinfos. In order, they are:

- **Field access of a uniqueinfo**: \( u_i \) refers to field \( i \) of tuple uniqueinfo \( u \).
- **Array contents of a uniqueinfo**: \( u[] \) refers to the contents of array uniqueinfo \( u \).
- **Application of a uniqueinfo**: \( \text{uiapply} \ u (a_1, \ldots, a_n) \) refers to the uniqueinfo part of the application of the function uniqueinfo \( u \) to arguments with unspecified names.
and uniqueinfos \( a_1 \) through to \( a_n \), with an unspecified result name. Note that the \( a_i \) uniqueinfos may be either definite or algebraic uniqueinfos.

- Join (least upper bound) of a set of uniqueinfos: \( u_1 \sqcup \cdots \sqcup u_n \) refers to the least upper bound of uniqueinfos \( u_1 \) through to \( u_n \).

The creation and manipulation of these algebraic uniqueinfos follows reasonably straightforwardly from the abstract semantics. If field access expression \( s . c / n \) finds that the uniqueinfo of \( s \) is algebraic, it constructs a field-reference uniqueinfo, \((\sigma s)(n+\text{firstfield } c)\).

Array contents are accessed via the primitive function \( \text{array_ref} \), not via a primitive, so there is no special semantics for arrays; the primitive function itself is hand-coded with an algebraic uniqueinfo:

\[
\text{array_ref}_a (\langle a, a_{ui} \rangle, \langle i, i_{ui} \rangle) \sim \langle \{\$, a[\]\}, a_{ui}[\]\rangle
\]

This automatically propagates and produces other array-contents uniqueinfos if \( \text{array_ref} \) is applied. There are several situations which join uniqueinfos: constructing an array, constructing a member of a recursive type (via type folding), and the result of analysing a conditional statement. In all such cases, if \( \sqcup \) is applied to an algebraic uniqueinfo, the operation is preserved algebraically. The most complicated case is function application. From the abstract semantics, function application produces a uniqueinfo of the form:

\[
\text{ui} \left( f \left( \langle [a_1], a_1 \rangle, \ldots, \langle [a_n], a_n \rangle \right) [\$] \right)
\]

Because names do not affect the output aside from the names that appear in the resulting sharing graph, the \( \text{ui} \) function (which selects the uniqueinfo and discards the sharing graph) will discard all of the names. This means that an expression of the above form has the same value regardless of the names of the arguments and result. Hence, we simplify the expression to:

\[
\text{uiapply} \ f \ (a_1, \ldots, a_n)
\]

via the use of the \( \text{uiapply} \) function, defined as:

\[
\text{uiapply} \ f \ (a_1, \ldots, a_n) = \text{ui} \left( f \left( \langle [?_1], a_1 \rangle, \ldots, \langle [?_n], a_n \rangle \right) [\$] \right)
\]

This function makes up arbitrary names to pass to the uniqueinfo \( f \); the names chosen have no effect on the output, as the sharing graph is discarded.

Lastly, after a definite uniqueinfo function is applied, the resulting uniqueinfo may contain unnecessary algebraic uniqueinfos that should be reduced wholly or partially to
definite uniqueinfos. For example, when analysing the expression:

```java
array_ref(x, 0)
```

if `x` has uniqueinfo `[y_{ui}]`, the result of applying `array_ref` will be `[y_{ui}][]`. This can (and should) be simplified to `y_{ui}.

### 9.8.2 Algebraic sharing graphs

An *algebraic sharing graph* is a sharing graph where some or all of the edges are the result of applying an unknown function. It is comprised of zero or more pseudo-edges.

\[
\text{Sharing}_{\text{alg}} \rightarrow \{ \text{PEdge} \ast \}
\]

\[
\text{PEdge} \rightarrow (\text{Name}, \text{Name})^{\text{Modifier}}
\]

| \[ \text{Name}^\ast \text{SelfModifier} \] |

A pseudo-edge is a term of the form:

\[
(x, y)^c
\]

This is said to be a pseudo-edge from `x` to `y` with *modifier* `c`. In a diagram, this would be represented as a line from `x` to `y` annotated with `c`. The names `x` and `y` are both literal names, whereas `c` is an algebraic term. Conceptually, a modifier is a function in the following domain:

\[
\text{Modifier} \equiv \text{Name} \rightarrow \text{Name} \rightarrow \text{Sharing}
\]

It maps two names of a pseudo-edge onto a set of zero or more real edges. The simplest modifier is called *id*, the identity modifier:

\[
\text{id} = \lambda \, x \, y. \{ \{x, y\}\}
\]

The identity modifier is how definite (non-algebraic) edges are represented in an algebraic sharing graph; in a diagram, if a line has no annotation, it is implicitly id. Another important modifier is called *null*, the null modifier:

\[
\text{null} = \lambda \, x \, y. \emptyset
\]

An edge with a null modifier is equivalent to having no edge at all. For pseudo-edges between simply typed names (types with no child nodes), every modifier is either id or null (although during the analysis, we may not know which). The modifier is essentially
9.8. ALGEBRAIC ANALYSIS

a Boolean term, answering the question, “is there an edge between \(x\) and \(y\), or not?” For structured types, the modifier may denote arbitrary edges between the components of the two names.

Note that pseudo-edges are ordered pairs, unlike regular edges which are unordered. This is important, because the modifier may treat its two arguments differently. For example, this modifier takes a pseudo-edge from \(x\) to \(y\) and modifies it so that the first element of \(x\) aliases the second element of \(y\):

\[
\lambda x y. \{ x.0, y.1 \}
\]

Applying this modifier to the pair \((\llbracket a \rrbracket, \llbracket b \rrbracket)\) is different to applying it to the pair \((\llbracket b \rrbracket, \llbracket a \rrbracket)\). The order of the two names is arbitrary, but important. Pseudo-edges are reversible, which means that wherever a pseudo-edge \((x, y)^c\) is present, the reverse edge \((y, x)^\text{flip}^c\) is also considered to exist.

However, modifiers are not represented in the compiler as arbitrary functions (they would be impossible to manipulate). Instead, they are represented with the following grammar:

\[
\text{Modifier} \quad \rightarrow \quad \text{id} \\
\quad \quad \quad \quad | \quad \lambda \ x \ y. \text{both } \{ x, y \} \ (\text{sgapply UniqueInfo}_{\text{alg}}(\llbracket ? \rrbracket, \ldots, x, \ldots, \llbracket ? \rrbracket) \ y \ \llbracket ? \rrbracket) \\
\quad \quad \quad \quad | \quad \lambda \ x \ y. \text{both } \{ x, y \} \ (\text{sgapply UniqueInfo}_{\text{alg}}(\llbracket ? \rrbracket, \ldots, \llbracket ? \rrbracket) \ y \ x) \\
\quad \quad \quad \quad | \quad \text{Modifier} \sqcup \text{Modifier} \\
\quad \quad \quad \quad | \quad \text{Modifier} \triangleright \text{Modifier}
\]

The second and third forms allow modifiers that are the partial result of applying an unknown function, represented by an algebraic uniqueinfo. The former is used on a pseudo-edge from an argument to a result; the latter on a pseudo-edge from a function to its result.

Here, \(\text{both} : (\text{Name} \otimes \text{Name}) \rightarrow \text{Sharing} \rightarrow \text{Sharing}\) takes two names \(\{x, y\}\), and filters out any edges of a sharing graph that do not feature a descendant of \(x\) (or \(x\) itself) on one side, and a descendant of \(y\) (or \(y\) itself) on the other.

The remaining forms allow modifiers to be composed using special overloads of \(\sqcup\) or alternating closure for modifiers:

\[
\sqcup : \text{Modifier} \rightarrow \text{Modifier} \rightarrow \text{Modifier}
\]

\[
c_1 \sqcup c_2 = \lambda x y. (c_1 x y) \sqcup (c_2 x y)
\]
The \( \star \) of two modifiers is given in Section 9.8.4.

A self-pseudo-edge is a term of the form:

\[ x^c \]

This is said to be a self-pseudo-edge on \( x \) with self-modifier \( c \). In a diagram, this would be represented by the node \( x \) having a self-loop with annotation \( c \). Conceptually, a self-modifier is a function in the following domain:

\[
\text{SelfModifier} \equiv \text{Name} \rightarrow \text{Sharing}
\]

It maps a single name onto a set of zero or more real edges. A special self-modifier is called \texttt{self\_null}, equivalent to having no additional edges:

\[
\texttt{self\_null} = \lambda x. \emptyset
\]

For example, this self-modifier specifies that the first and second elements of a given node are aliased:

\[
\lambda x. \{\{x.0, x.1\}\}
\]

Self-modifiers are represented in the compiler with the following grammar:

\[
\text{SelfModifier} \rightarrow \text{id} \\
| \; \lambda x. \text{remove} \{[?1], \ldots, [?n]\} \\
| \; (\text{sgapply UniqueInfo}_{\text{alg}} ([?1], \ldots, [?n-1]) \; x \; [?n]) \\
| \; \text{SelfModifier} \; \sqcup \; \text{SelfModifier} \\
| \; \text{SelfModifier} \; \star \; \text{SelfModifier}
\]

The second form allows self-modifiers that are the partial result of applying an unknown function, represented by an algebraic uniqueinfo. This captures any remaining edges that are not captured by the regular pseudo-edges; specifically, edges between:

- two components of \( x \),
- a component of \( x \) and \( @ \), and
- a component of \( x \) and a local variable in a different scope.

Recall from Section 8.3.5 that \texttt{remove} : \( \varphi(\text{Name}) \rightarrow \text{Sharing} \rightarrow \text{Sharing} \) filters a sharing graph, removing any edges with components of any name in the given set, on either end. Therefore, we are left with only edges that do not involve the parameters or the closure.

The remaining forms allow self-modifiers to be composed using special overloads of
9.8. ALGEBRAIC ANALYSIS

lub or alternating closure for modifiers:

\[ \square : \text{SelfModifier} \rightarrow \text{SelfModifier} \rightarrow \text{SelfModifier} \]

\[ c_1 \uplus c_2 = \lambda x. (c_1 x) \uplus (c_2 x) \]

The \( \sqsubseteq \) of a self-modifier and a modifier is given in Section 9.8.4.

9.8.3 Algebraic function application

A sharing graph application is an algebraic expression denoting the sharing graph of a function application:

\[
\text{Application} = \begin{cases} 
\text{sgapply} f_{ui} (\langle n_1, u_1 \rangle, \ldots, \langle n_n, u_n \rangle) r f : \\ 
\{ f_{ui}, u_1, \ldots, u_n \} \subseteq \text{UniqueInfo}_{\text{alg}} \land \{ n_1, \ldots, n_n, r, f \} \subseteq \text{Name} 
\end{cases}
\]

This assumes that \( f \) is an algebraic uniqueinfo, and all of the names and uniqueinfo variables are present in the enclosing scope of the sharing graph. The \text{sgapply} function is short-hand for the completion of the sharing graph of the application of a uniqueinfo:

\[ \text{sgapply} = \text{completion} \circ \text{sg} \]

Consider an unknown function \( f \) with uniqueinfo \( f_{ui} \). The sharing graph resulting from the statement \( r = f(x_1, \ldots, x_n) \) is:

\[ \text{sgapply} f_{ui} (\langle [x_1], x_{1ui} \rangle, \ldots, \langle [x_n], x_{nui} \rangle) \ [r] \ [f] \]

From this, we generate the following pseudo-edges and self-pseudo-edges:

\[ ([x_1], [r])^c, \quad \text{where } c = \lambda x y. \text{both } \{ x, y \} \ (\text{sgapply} f_{ui} (\langle x, x_{1ui} \rangle, \ldots, \langle [?], x_{nui} \rangle)) \ y \ [?] \]

\[ (\ldots) \]

\[ ([x_n], [r])^c, \quad \text{where } c = \lambda x y. \text{both } \{ x, y \} \ (\text{sgapply} f_{ui} (\langle [?], x_{1ui} \rangle, \ldots, \langle x, x_{nui} \rangle)) \ y \ [?] \]

\[ ([f], [r])^c, \quad \text{where } c = \lambda x y. \text{both } \{ x, y \} \ (\text{sgapply} f_{ui} (\langle [?], x_{1ui} \rangle, \ldots, \langle [?], x_{nui} \rangle)) \ y \ x \]

\[ (\ [r] \ [?], \text{where } c = \lambda x. \text{remove } \{ [?], \ldots, [?], [?]_n, [?]_{n+1} \} \]

\[ (\text{sgapply} f_{ui} (\langle [?], x_{1ui} \rangle, \ldots, \langle [?], x_{nui} \rangle)) \ x \ [?]_{n+1} \]

Note that the modifiers contain no literal names (such as \([x_1]\) or \([r]\)). The names are contained in the pair that the modifier is applied to. This makes the modifier portable, allowing it to be applied to different name pairs.
9.8.4 Algebraic alternating closure

The alternating closure operator \((A \bowtie B)\) is extended to work on algebraic sharing graphs:

\[
X \bowtie Y = \begin{cases} 
\exists a_0, \ldots, a_n \in S, \\
\{ (a_0, a_1)^{c_1}, (a_1, a_2)^{c_2}, \ldots, (a_{n-1}, a_n)^{c_n} \} \subseteq X \cup Y \land \\
\text{type_compatible}(\text{type } a_0, \text{type } a_n) \land \\
\{(a_{i-1}, a_i)^{c_i} \in X \Rightarrow (a_i, a_{i+1})^{c_{i+1}} \in Y \land \\
(a_{i-1}, a_i)^{c_i} \in Y \Rightarrow (a_i, a_{i+1})^{c_{i+1}} \in X \} 
\end{cases}
\]

\[
\cup \begin{cases} 
\exists a_0, \ldots, a_n \in S, \\
\{ (a_0, a_1)^{c_0\text{self}}, (a_1, a_2)^{c_1}, \ldots, (a_{n-1}, a_n)^{c_n} \} \subseteq X \cup Y \land \\
\text{type_compatible}(\text{type } a_0, \text{type } a_n) \land \\
\{(a_{i-1}, a_i)^{c_i} \in X \Rightarrow (a_i, a_{i+1})^{c_{i+1}} \in Y \land \\
(a_{i-1}, a_i)^{c_i} \in Y \Rightarrow (a_i, a_{i+1})^{c_{i+1}} \in X \} 
\end{cases}
\]

This works almost the same way as regular alternating closure, but any new edges it creates have a modifier that is the alternating closure of all of the modifiers that were used to derive that new edge. The alternating closure of modifiers is as follows:

\[
\bowtie : \text{Modifier} \rightarrow \text{Modifier} \rightarrow \text{Modifier} \\
c_1 \bowtie c_2 = \lambda x, y. \text{both } \{ x, y \} \ ( (c_1 \ x \ [t]) \bowtie (c_2 \ [t] \ y)) \\
\bowtie : \text{SelfModifier} \rightarrow \text{Modifier} \rightarrow \text{SelfModifier} \\
c_{\text{self}} \bowtie c = \lambda x. \text{remove } \{ [t] \} \ ( (c_{\text{self}} \ [t]) \bowtie (c \ [t] \ x))
\]

This definition uses a new name \([t]\) which must be a different name than \(x\) or \(y\).

The alternating closure of modifiers \(c_1\) and \(c_2\) is a new modifier that, when applied to \(x\) and \(y\), gives the new edges that would be obtained from a regular alternating closure over the edges from \(x\) to \([t]\) with modifier \(c_1\) and the edges from \([t]\) to \(y\) with modifier \(c_2\).

For example:

\[
(x, y)^{c_1} \bowtie (y, z)^{c_2} = (x, y)^{c_1} \cup (y, z)^{c_2} \cup (x, z)^{c_1\bowtie c_2}
\]

The modifier on the edge from \(x\) to \(z\) is the alternating closure of the edge from \(x\) to \(y\) (modified by \(c_1\)) and the edge from \(y\) to \(z\) (modified by \(c_2\)).

The following axioms naturally arise concerning the special modifiers given above:

\[
\forall c \in \text{Modifier}. \ c \bowtie \text{id} = c \\
\forall c \in \text{Modifier}. \ c \bowtie \text{null} = \text{null}
\]
If we think of \( \text{id} \) and \( \text{null} \) as 1 and 0, respectively, alternating closure is a logical AND. In the context of the above example, if \( x, y \) and \( z \) have simple types, we can think of \( c_1 \) and \( c_2 \) as Boolean conditions, and consider there to be an edge between \( x \) and \( z \) if and only if \( c_1 \land c_2 \).

The alternating closure of a self-modifier \( c_{self} \) with a regular modifier \( c \) is a new self-modifier that, when applied to \( x \), gives the new edges that would be obtained from a regular alternating closure over \( [I] \) with self-modifier \( c_{self} \) and the edges from \( [I] \) to \( x \) with modifier \( c \).

For example:

\[
x^{c_{self}} \bowtie (x, y)^c = x^{c_{self}} \cup (x, y)^c \cup y^{c_{self} \land c}
\]

### 9.8.5 Algebraic reify

An algebraic version of the \( \text{reify} \) operator is required to deal with closure templates that may return algebraic sharing graphs. This is a very simple translation of the original \( \text{reify} \) function, replacing edges with pseudo-edges and copying the modifiers across. For completeness, it is defined here:

\[
\text{reify}_{\text{alg}} : \text{Var} \rightarrow \text{CT}_{\text{alg}} \rightarrow \langle \text{Name}_\perp, \text{UniqueInfo}_{\text{alg}} \rangle^* \rightarrow \text{UniqueInfo}_{\text{alg}}
\]

where

\[
\text{reify}_{\text{alg}} c f ((x_1, u_1), \ldots, (x_n, u_n)) = \langle \eta', \lambda ((y_1, v_1), \ldots, (y_m, v_m)) \% \eta''. \ (r_{ui}) \rangle
\]

\[
\eta' = \{ \{ c(), x_1 \}, \ldots, \{ c(), x_n \} \} \cup \{ (p, k\, c\#)^c | \begin{align*}
\ (p, q)^c &\in \eta \land 0 \leq i < n \land \\
\ &z \in \text{descendants (type } x_i) \land \\
\ &k \in \text{descendants (type } \$) \land \\
\ &p = z \land q = k\$
\end{align*} \}
\]

\[
\eta'' = \{ \begin{align*}
\ (k\ %\#, q)^c, \ &\text{if } p = z \land q = k\$
\ (k\ %\#, p)^c, \ &\text{if } q = z \land p = k\$
\ (p, q)^c, \ &\text{otherwise}
\end{align*} | \begin{align*}
\ (p, q)^c &\in \eta \land 0 \leq i < n \land \\
\ &z \in \text{descendants (type } x_i) \land \\
\ &k \in \text{descendants (type } \$)
\end{align*} \}
\]

### 9.8.6 Widening

As discussed in Section 9.6.3, the UniqueInfo_{\text{alg}} data structure has unbounded height, unlike UniqueInfo which has a bounded height for any given type. In rare circumstances, an analysis may not terminate because the uniqueinfo grows larger on each iteration, never reaching a fixed point.

To deal with this, we \textit{widen} \cite{14} an algebraic uniqueinfo once it has reached a certain
height within a given node of the type tree, replacing it with $\top$. Importantly, we do not limit the height of uniqueinfo nodes with separate types (as that would weaken our previous analysis); we only limit the growth of uniqueinfo nodes that do not descend the type tree. Our widening operation does not descend into lambda, array and tuple nodes unless they are cancelled out by corresponding indexing or $\text{uiapply}$ nodes.

\[
\text{ui}_\text{height} \, d \, (\lambda \, as \, %, \langle \eta, u \rangle) = 1 + \begin{cases} 
0, & \text{if } d = 0 \\
\text{ui}_\text{height} \, (d - 1) \, u, & \text{if } d > 0 
\end{cases}
\]

\[
\text{ui}_\text{height} \, d \, [u] = 1 + \begin{cases} 
0, & \text{if } d = 0 \\
\text{ui}_\text{height} \, (d - 1) \, u, & \text{if } d > 0 
\end{cases}
\]

\[
\text{ui}_\text{height} \, d \, (u_1, \ldots, u_n) = 1 + \begin{cases} 
\max \{ \text{ui}_\text{height} \, (d - 1) \, u_i : 1 \leq i \leq n \}, & \text{if } d > 0 
\end{cases}
\]

\[
\text{ui}_\text{height} \, d \, ^\uparrow^n = 1
\]

\[
\text{ui}_\text{height} \, d \, u = 1, \quad \text{where } u \in \text{UIVar}
\]

\[
\text{ui}_\text{height} \, d \, u_i = 1 + \text{ui}_\text{height} \, (d + 1) \, u
\]

\[
\text{ui}_\text{height} \, d \, u[] = 1 + \text{ui}_\text{height} \, (d + 1) \, u
\]

\[
\text{ui}_\text{height} \, d \, (\text{uiapply} \, u_1 \, (u_2, \ldots, u_n)) = 1 + \max \{ \text{ui}_\text{height} \, (d + 1) \, u_i : 1 \leq i \leq n \}
\]

\[
\text{ui}_\text{height} \, d \, (u_1 \sqcup \cdots \sqcup u_n) = 1 + \max \{ \text{ui}_\text{height} \, d \, u_i : 1 \leq i \leq n \}
\]

Any uniqueinfo for which $\text{ui}_\text{height} \, 0$ is greater than a given threshold (say, 5) is replaced with $\top$. The threshold is arbitrary, and should be large enough to avoid false positives in practical cases.

9.9 Analysing non-Mars language features

In Section 5.3, a number of non-Mars features were discussed with respect to their translation into MIR or (in the case of lazy evaluation) an extension of MIR. In this section, we revisit these features and discuss how our higher-order analysis would cope upon encountering them.

9.9.1 Type classes

In Section 5.3.3, we briefly addressed the notion of a language with type classes, such as Haskell or Mercury, being transformed into MIR. This is possible without any changes
to the underlying representation, via the use of *type dictionaries*. Any function accepting a parameter constrained by a type class would, in MIR, accept a tuple containing one or more closures — the implementations of the methods for the parameter’s concrete type.

Any function that calls one of these methods would be considered an applicative downward higher-order function during the aliasing analysis, and analysed appropriately. At the function’s call site, the concrete type is known, and therefore, the concrete method implementations are too. Therefore, we can achieve a precise aliasing analysis of type-class-constrained functions, incorporating details about the aliasing behaviour of the methods of the concrete types passed to the function.

Note that the same benefits can be obtained with a compiler that specialises implementations for specific concrete types (as discussed in Section 9.2, specialising higher-order programs to first-order ones can, in general, achieve many of the benefits of a full higher-order analysis, but has several drawbacks).

### 9.9.2 Existential and virtual types

Some languages allow *existential types*\(^8\) (in object-oriented languages, these are called virtual classes and are ubiquitous). Unlike conventional use of type classes, where a static type variable is bound to a concrete type at runtime, each object of an existential type may have a different concrete type. This allows, for example, a heterogeneous list, where each element has a different concrete type.

To implement existential types in MIR, rather than attaching type dictionaries to type constraints, a type dictionary (or *virtual method table* in object-oriented parlance) is attached to each object.

As with type classes, our aliasing analysis can naturally handle this implementation without any special rules. Because the existentially typed objects appear as tuples with closures, our applicative downward higher-order analysis can treat the closures algebraically and evaluate them at call time.

We can achieve a much more precise result than a specialising compiler, because unlike the conventional usage of type classes (which specialises very well), it is unlikely that a compiler could specialise the method calls on, say, a collection of existentially typed objects. Note that due to the possibility of heterogeneous collections, some precision may be lost, but we can still do better than nothing; the analysis will consider the aliasing behaviour of only the methods which *may* be in a given heterogeneous collection of objects.

---

\(^8\)In Haskell, these are available as an extension to the Glasgow Haskell Compiler.
9.9.3 Lazy evaluation

In Section 5.3.4, we sketched out potential changes to MIR which could be made to accommodate implementation of a lazy evaluation strategy; in particular, the ability to create a thunk value that could hold a future computation. In this section, we revisit this language extension with respect to the higher-order aliasing analysis.

As previously noted, thunk templates can be treated as closure templates for the purpose of analysis (making sure that the result of a thunk is always aliased to the thunk’s # node, representing the memory cell of the thunk’s cache). Therefore, our higher-order analysis can easily be modified to deal with lazy evaluation.

The major problem with this is the memory usage characteristics of programs executed under a lazy evaluation strategy: values can stay referenced for longer than they would under an eager evaluation strategy, meaning there will be fewer opportunities for reuse. Consider the setLast function of Example 9.11 (adapted from Section 7.2), presented in a hypothetical lazy version of Mars.

Example 9.11 The setLast function in “lazy Mars”

```python
def setLast(xs :: Array(a), y :: a):
    i = array_length(xs) - 1
    return array_replace(xs, i, y)
```

In regular Mars, this code would destructively update xs if it were unique, but in “lazy Mars,” as in Clean or Haskell, the thunk value i retains a pointer to xs, and will only evaluate it inside the call to array_replace. This means we cannot call the destructive version of array_replace.

In some cases, a strictness analysis [50] may improve the opportunities for reuse by forcing the evaluation of thunks early (and hence removing pointers to their closure data). For example, in the setLast case, we can tell trivially that array_replace will always use the value of its second argument, and hence, we can force its evaluation before the call without changing the semantics. This removes the problematic thunk i from the sharing graph before calling array_replace, which means that xs will be considered unique if it was passed in as a unique value.
9.10 Conclusion

In this chapter, the aliasing analysis of Chapter 8 was extended with full support for higher-order functions. We began with a justification of this approach: in the past, program transformations such as specialisation of higher-order functions, and defunctionalisation, have been proposed as a way of statically analysing higher-order functions using first-order techniques. However, this approach has a number of significant drawbacks, including being fairly limited in the set of cases it can handle, as well as being coupled to a heuristic in a separate part of the compiler.

We focused on analysing three major categories of higher-order function, in increasing order of complexity:

1. Upward higher-order functions (those that return functions, but do not accept functions as arguments),
2. Non-applicative downward higher-order functions (those that accept functions as arguments, but do not call them),
3. Applicative downward higher-order functions (those that call functions passed as arguments).

The key idea in this chapter is a data structure known as a uniqueinfo, which records the aliasing behaviour for any first-class value. The uniqueinfo of a closure is the sharing graph relating its inputs to its outputs. Non-closure values have uniqueinfos too — for example, the uniqueinfo of a pair is a pair of uniqueinfos, one for each element. In this way, we preserve aliasing information about closures nested within other data structures.

We handle upward higher-order functions by having the analysis record not just the sharing graph for each function, but also the uniqueinfo of its result. That way, when a function returns a closure, the caller is aware of what aliasing it might create when called, and does not need to simply assume maximal aliasing.

We handle non-applicative downward higher-order functions by allowing the uniqueinfo in a function’s result to refer to the uniqueinfos of the function’s parameters. This allows us to perfectly preserve knowledge about the aliasing behaviour of a closure that is passed down into a function, and then returned as part of the result.

Finally, we handle applicative downward higher-order functions by extending the sharing graph representation with “pseudo-edges.” A pseudo-edge between two nodes has a modifier, which is like a condition expressed in terms of the unknown uniqueinfos of the function’s closure parameters. We describe algebraic versions of the abstract operations, allowing us to preserve algebraic information in the sharing graph.
As a result of these changes, we have created a context-insensitive analysis that does not throw away any information from the caller as a result of dealing with unknown closure values. The algebraic analysis faithfully preserves all operations on both the unique-infos and sharing graph, so that once substituted with concrete unique-infos, it produces the same results as a context-sensitive analysis would.\footnote{There is a rare exception to this, which is if an algebraic uniqueinfo becomes excessively large, within the same node of the type tree, it is \textit{widen}, giving a weaker result. This is only performed in degenerate cases, as discussed in Section 9.6.3.}

One major drawback of this approach is that the resulting reports (the algebraic sharing graphs and unique-infos that come out of the analysis of each function) can be extremely large and difficult to read (\textit{e.g.}, see the result for \texttt{compose} in Section 9.6.2 — a relatively simple higher-order function). This goes somewhat against our goal of being able to produce human-readable reports of the aliasing of these functions. To some extent, this could be mitigated by a good user interface that presents the user with an algebraic sharing diagram and allows them to explore areas of interest, but it is still questionable whether these tools can be useful enough to help diagnose erroneously detected or legitimate, but unexpected aliasing in higher-order functions.

### 9.10.1 Implementation status

The Mars compiler currently only implements part of what is described in this chapter. What we have implemented is sufficient to handle the upward higher-order and non-applicative downward higher-order cases, but not the applicative case (which requires a significant amount of extra implementation work).

The implementation includes support for algebraic unique-infos (§9.8.1), excluding the \texttt{uiapply} case. We have not implemented the remainder of Section 9.8, as it was not required for the non-applicative cases. When an unknown function is applied, the current implementation simply assumes maximal aliasing, as in Chapter 8.

In the next chapter, we finally make use of the aliasing results: a new analysis is introduced, which determines where it is possible to safely optimise code to perform a destructive update, based on whether the value being updated is definitely not aliased.
Chapter 10

Destructive update optimisation

10.1 Introduction

In the previous two chapters, a precise higher-order uniqueness inference system was presented, allowing us to know whether any given variable is definitely not aliased at any given program point. The reason we need this information is that if a variable is featured in a pure operation that copies and modifies a data structure, if that variable is known not to be aliased or needed again in the future, the copy-and-modify operation can be automatically upgraded to a destructive update, making the program more efficient (in some cases by a linear strength reduction) without changing the semantics. In this final chapter, we explore the use of our aliasing information in actually performing this optimisation.

The first part of Chapter 8 gave some simple examples (Examples 8.1–8.3) of the destructive update optimisation in practice. We introduced the notation $\Omega\{x\}$ to indicate that the variable $x$ is dead (no longer to be used within the current procedure), and the idea of specialisation, whereby a function such as $\text{array}_\text{replace}$ could have a destructive version $\text{array}_\text{replace}_d$ that can be called if its array parameter is dead and unique. We will explore several other opportunities for reuse, and formalise the above concepts.

There are two separate optimisations which together form the reuse system:

1. Structural reuse, in which construction operations (including those syntactically represented as field-replace operations) are converted into destructive update operations, and

2. Call mode selection, in which arguments passed to functions are annotated for uniqueness, to allow the called procedures to take advantage of reuse on those arguments.
The two optimisations complement one another: structural reuse relies on knowledge about which objects are unique, and that information is supplied by the call mode selection optimisation. These concepts correspond to Mazur’s direct reuse and indirect reuse, respectively: direct reuse is reusing dead cells in the current context, whereas indirect reuse is calling procedures with direct or indirect reuse opportunities [65].

Only part of the material in this chapter has been implemented in the Mars compiler at the time of writing. Therefore, while we have concrete theoretical approaches to solving some of these problems, we do not have experimental results for all of our solutions.

The remainder of this chapter is organised as follows. In Section 10.2, we describe the liveness analysis algorithm for determining when variables become dead. In Section 10.3, we formally define the call signature and call mode data structures, and the main algorithm for determining the conditions for calling destructive versions of procedures. In Section 10.4, we describe a runtime system for dynamically determining which version to call based on the static information, and in Section 10.5, a system for creating specialised versions at compile time is described; these two systems can be used interchangeably or in conjunction. In Section 10.6, extensions to the call mode system for handling higher-order functions are described. In Section 10.7, we discuss the algorithm for converting constructor calls into field-update instructions. Finally, Section 10.8 discusses the parts that have not yet been implemented, and possible future improvements to this system. Section 10.9 concludes.

10.2 Live-variable analysis

The first step in determining whether it is legal to destructively update a variable, or component thereof, is live-variable analysis, which determines whether the variable is dead. A dead variable is one that will not be read from again. This relatively simple analysis should not be confused with the aliasing analysis covered in the previous chapters (which is concerned with whether there are any remaining pointers to a memory cell). The live variable analysis is used in conjunction with the aliasing analysis: in order to guarantee a safe destructive update of variable $x$ at program point $p$, it must be the case that $x$ has no aliases at $p$, and that $x$ is not live after $p$.

The problem description for live-variable analysis follows:

Given a variable $x$ at program point $p$, will $x$ be read before it is written at any program point after $p$ on any code path? If so, $x$ is live at $p$; otherwise, $x$ is dead at $p$. 
10.2. **LIVE-VARIABLE ANALYSIS**

Note that a variable is considered dead on the final line that it is used. Liveness is a local property — there is no need for an inter-procedural analysis. The basic algorithm is described by Aho *et al.* [4] in terms of a control-flow graph, comprised of basic blocks; it is adapted to suit MIR, which is in nested statement form.

We first assign two sets of variables to each statement: def and use. These are defined as follows:

**Definition 10.1.** $\text{def}_s$ is the set of variables defined (assigned) by the statement $s$ before any use of the variable.

**Definition 10.2.** $\text{use}_s$ is the set of variables that are used (read from) by the statement $s$ before any definition of the variable.

Note that for any given statement, the two sets are mutually exclusive: the reassignment rule (Rule 5.1) prevents a statement from both using and defining a variable.\(^1\) The formal definition of the use and def sets for MIR is given in Figure 10.1.

From these sets, we can compute two useful results for each statement $s$:

1. $\Omega_s$ (pronounced “last-use in $s$”), the set of variables that become dead during the execution of $s$, and
2. The liveness state of each variable $x$ after $s$.

Algorithm 10.1 shows how to compute the set of live variables at the end of each statement $s$ ($\text{live}_s$).\(^2\)

Algorithm 10.2 shows how to compute the set $\Omega$ for each statement $s$, which represents the set of variables that transition from live to dead directly during the execution of that statement. (This definition is not transitive; for example, an if statement’s $\Omega$ does not include all of the variables that die during the then or else branches, only while evaluating the condition.) It is convenient to show these visually in source code listings; the examples in this chapter will show the $\Omega$ set for each line that it is non-empty.\(^3\)

Finally, we embellish the liveness information for each variable with some additional information depending on whether the variable is a parameter. At the end of each statement $s$, for each variable $x$, we say that $x$’s liveness state is one of the following:

$$\{\text{Live, Dead, Dead-Parameter}\}$$

---

\(^1\)Even without the reassignment rule, if a statement both defined and used a variable, it would belong to the set of whichever comes first.

\(^2\)This algorithm is adapted from [4], which is block-based.

\(^3\)When asked to print out the MIR code for a program, our compiler also shows the $\Omega$ set for each line.
use_A : Atom → ϕ(Var)
use_E : Expr → ϕ(Var)
use_S : Stmt → ϕ(Var)
def_S : Stmt → ϕ(Var)

\[
\begin{align*}
\text{use}_A [n] &= \emptyset, \text{ where } n \in \text{Num} \\
\text{use}_A [v] &= \{v\}, \text{ where } v \in \text{Var} \\
\text{use}_E [a] &= \text{use}_A [a], \text{ where } a \in \text{Atom} \\
\text{use}_E [c] &= \emptyset, \text{ where } c \in \text{CtorName} \\
\text{use}_E [s \cdot c / n] &= \{s\} \\
\text{use}_E [(a_1, \ldots, a_n)] &= \text{use}_A [a_1] \cup \cdots \cup \text{use}_A [a_n] \\
\text{use}_E [c(a_1, \ldots, a_n) \text{hint}] &= \text{use}_A [a_1] \cup \cdots \cup \text{use}_A [a_n] \\
\text{use}_E [f \in \text{Var}(a_1, \ldots, a_n)] &= \{f\} \cup \text{use}_A [a_1] \cup \cdots \cup \text{use}_A [a_n] \\
\text{use}_E [f \in \text{FuncName}(a_1, \ldots, a_n)] &= \text{use}_A [a_1] \cup \cdots \cup \text{use}_A [a_n] \\
\text{use}_E [f\{a_1, \ldots, a_n\}] &= \text{use}_A [a_1] \cup \cdots \cup \text{use}_A [a_n]
\end{align*}
\]

\[
\begin{align*}
\text{use}_S [e] &= \emptyset \\
\text{use}_S [s_1 ; s_2] &= \text{use}_S [s_1] \cup (\text{use}_S [s_2] \setminus \text{def}_S [s_1]) \\
\text{use}_S [v = e] &= \text{use}_E [e] \\
\text{use}_S [\text{if } a: s_{\text{then}} \text{ else: } s_{\text{else}}] &= \text{use}_A [a] \cup \text{use}_S [s_{\text{then}}] \cup \text{use}_S [s_{\text{else}}] \\
\text{use}_S [\text{switch } v: (c_1 : s_1, \ldots, c_n : s_n)] &= \{v\} \cup \text{use}_S [s_1] \cup \cdots \cup \text{use}_S [s_n] \\
\text{use}_S [\text{while } a [u_1 = \phi(v_1, w_1), \ldots]: s] &= \{v_1, \ldots, v_n\} \cup \text{use}_A [a] \cup \\
& \quad (\text{use}_S [s] \setminus \{u_1, \ldots, u_n\})
\end{align*}
\]

\[
\begin{align*}
\text{def}_S [e] &= \emptyset \\
\text{def}_S [s_1 ; s_2] &= \text{def}_S [s_1] \cup (\text{def}_S [s_2] \setminus \text{use}_S [s_1]) \\
\text{def}_S [v = e] &= \{v\} \setminus \text{use}_E [e] \\
\text{def}_S [\text{if } a: s_{\text{then}} \text{ else: } s_{\text{else}}] &= \text{def}_S [s_{\text{then}}] \cap \text{def}_S [s_{\text{else}}] \\
\text{def}_S [\text{switch } v: (c_1 : s_1, \ldots, c_n : s_n)] &= \text{def}_S [s_1] \cap \cdots \cap \text{def}_S [s_n] \\
\text{def}_S [\text{while } a [u_1 = \phi(v_1, w_1), \ldots]: s] &= \{u_1, \ldots, u_n\}
\end{align*}
\]

Figure 10.1: The use and def sets
10.3 Call mode selection

Examples 8.1–8.3 gave a broad overview of call mode selection. The idea is to generate destructive versions of functions, which may assume that certain parameters are unique in the caller. A destructive version may take advantage of this in three ways:

1. It may be a primitive, such as array_replace_d, and may therefore be hard-coded to perform destructive update.

2. It may destructively update fields of its unique parameters. (The technique for
doing so is discussed in Section 10.7.)

3. It may recursively make call mode selections, taking advantage of the uniqueness of its parameters.

We now introduce formal data structures for specifying how procedures are called with respect to uniqueness information.

### 10.3.1 Simple per-argument call modes

As a first approximation, we assume that we want to know the uniqueness of all parameters to a procedure, and that we do not need to know the uniqueness of any components of a parameter. For each function call, the caller must communicate information to the callee about which arguments are safe to destructively update. A variable (i.e., non-literal) actual parameter is safe to destructively update if, and only if:

1. It is **dead** (DEFAP or DEAD-PARAMETER and the parameter is not aliased) at the program point of the function call, and
2. It is not aliased to a live variable at this program point.

If both of these conditions are met, that actual parameter’s **mode** is D (destructive); otherwise, the parameter’s mode is N (non-destructive or normal). A number literal actual parameter always has mode N. Each procedure call has a **call mode**, a vector of argument modes.

**Example 10.1** A call to array_replace with no aliasing

```latex
\begin{align*}
b &= \text{array}\_\text{replace}(a, 0, 42) \\
\text{return } b
\end{align*}
\triangleright \Omega\{a\}; \langle D, N, N \rangle
\triangleright \Omega\{b\}
```

**Example 10.2** A call to array_replace with a live

```latex
\begin{align*}
b &= \text{array}\_\text{replace}(a, 0, 42) \\
\text{return } (a, b)
\end{align*}
\triangleright \langle N, N, N \rangle
\triangleright \Omega\{a, b\}
```

**Example 10.3** A call to array_replace with a aliased

```latex
\begin{align*}
b &= a \\
c &= \text{array}\_\text{replace}(a, 0, 42) \\
\text{return } (b, c)
\end{align*}
\triangleright \Omega\{a\}; \langle N, N, N \rangle
\triangleright \Omega\{b, c\}
```

\[4\text{Recall that number literals are considered aliased to the static data cells. We do not currently support destructive update operations for numbers, but if we did, it would be unsound to allow it on literals.}\]
10.3. CALL MODE SELECTION

Example 10.1 shows a call to `array_replace` where both of the above conditions are satisfied, and hence the first argument’s mode is $D$. The call mode is $\langle D, N, N \rangle$. In Example 10.2, the variable $a$ is LIVE after the call (being used in the following `return` statement), so it cannot be destructively updated, and the call mode is $\langle N, N, N \rangle$. In Example 10.3, $a$ is aliased to the live variable $b$ (as determined by our aliasing analysis), so it cannot be destructively updated.

Recall that Rule 5.2 states that no variable may appear more than once in a function argument list, preventing MIR code such as “`array_concat(a, a)`”. Without Rule 5.2, we would need an additional rule stating that a variable must have mode $N$ if it is passed as more than one of the arguments.

**Example 10.4**
A call to `array_replace` with a formal parameter

```
def set42(a :: Array(Num)):  
    b = array_replace(a, 0, 42)  \triangleright \Omega\{a\}; \langle \{a\}, N, N \rangle 
    return b  \triangleright \Omega\{b\}
```

In Example 10.4, a function passes its formal parameter $a$ to `array_replace`. Since $a$ is not aliased at that program point, nor repeated in the argument list, and it is not live after that program point, it may be eligible for destructive update. However, it is not DEAD; since it is a formal parameter, its liveness state is DEAD-PARAMETER. This means that its mode cannot be determined statically — it is conditional on the call mode of formal parameter $a$ at the call site of `set42`. We therefore express the call mode to `array_replace` as $\langle \{a\}, N, N \rangle$. We will have to decide whether to choose $D$ or $N$ at a later time; techniques for resolving call mode conditions are discussed in Sections 10.4 and 10.5.

In general, a parameter’s mode may be conditional on more than one variable. In Example 10.5, the variable $x$ may alias $a$ or $b$ on Line 6 where it is passed to `array_replace`. Hence, its mode is $\{a, b\}$. This is a logical conjunction: $\{a, b\}$ resolves to $D$ if and only if both $a$ and $b$ are $D$.$^5$

**Example 10.5** A multi-variable conditional mode

```
def cond_set42(c :: Num, a :: Array(Num), b :: Array(Num)):  
    if c:  
        x = a  \triangleright \Omega\{a\} 
    else:  
        x = b  \triangleright \Omega\{b\} 
    y = array_replace(x, 0, 42)  \triangleright \Omega\{x\}; \langle \{a, b\}, N, N \rangle 
    return y  \triangleright \Omega\{y\}
```

$^5$This may result in sub-optimal behaviour. For example, `cond_set42` may be called with $a$ (but not $b$) eligible for destructive update, and $c = 1$. It will not destructively update $a$, even though it could.
CHAPTER 10. DESTRUCTIVE UPDATE OPTIMISATION

10.3.2 Interestingness and call signatures

Not all argument modes are useful: an argument mode will be ignored by the callee unless some call mode in the callee is conditional on the parameter. In fact, the majority of parameters are not used in a way that could take advantage of destructive update. To simplify call modes and avoid unnecessary specialisation, we define the concept of interestingness.

**Definition 10.3.** A formal parameter is **interesting** if and only if there is a code path in which the parameter’s memory cell could be destructively updated, were its argument not aliased.

An interesting parameter is one for which passing an argument with mode $D$ (as opposed to $N$) may result in additional destructive update; a non-interesting parameter would see no difference between $D$ and $N$. For example, `array_replace` has no business destructively updating its second or third parameters, so only the first parameter is interesting. Similarly, the first parameter of `array_concat` is interesting, but the second is not — even if the second array is unique, it still has to be copied onto the end of the first array, so using mode $D$ for the second array would not result in an improvement.

The property of interestingness is transitive, and is determined by a simple bottom-up static analysis. In each user-defined function, a parameter $x$ is marked interesting if any call mode within the function is conditional on $x$. In `cond_set42` of Example 10.5, arguments $a$ and $b$ are interesting, while $c$ is not. Built-in functions are hand-annotated for interestingness, based on whether destructive update is possible for each parameter.6

To record which parameters are interesting, we introduce **call signatures**. Each function has a call signature, a sequence of argument positions that are interesting. The call signature of `array_replace` and `array_concat` is $\langle 0 \rangle$. The call signature of `array_length` is $\langle \rangle$ (no parameter is interesting). The call signature of `cond_set42` is $\langle 1, 2 \rangle$.

We redefine call mode as a vector of argument modes for each interesting argument. Non-interesting arguments are assumed to have the mode $N$. Therefore, in Example 10.1, the call to `array_replace` has call mode $\langle D \rangle$, with the remaining two parameters implicitly having mode $N$. Note that the call mode corresponds to the call signature: for a given call signature $\langle p_0, \ldots, p_{n-1} \rangle$ and corresponding call mode $\langle m_0, \ldots, m_{n-1} \rangle$, argument mode $m_i$ is the mode for parameter position $p_i$, for $i \in \{0, \ldots, n - 1\}$.

---

6Ideally, user-defined foreign language functions would also be hand-annotated, but currently are assumed to have no interesting parameters.
10.3. CALL MODE SELECTION

10.3.3 Structural call modes

We have so far described only argument modes, ignoring the destructibility of the components of a parameter. Example 10.6 highlights the weakness of this approach. This example, `element_array_add`, takes a list of arrays and updates one of the arrays in the list. It returns the updated array (not modifying the list). Ideally, if the arrays in the list are unique, we want to perform a destructive call to `array_add`. The problem is that while our aliasing analysis accurately tracks the aliasing of components of variables (§8.2.5), we have no way for the caller to communicate that a component of a parameter is unique.

Example 10.6 Destructive update of a component of a parameter

```python
def element_array_add(arrays :: List(Array(a)), i :: Num, x :: a):
    if i:
        arrays′ = arrays.Cons/1
        i′ = i − 1
        r = element_array_add(arrays′, i′, x)
    else:
        array = arrays.Cons/0
        r = array_add(array, x)
    return r
```

The problem is on Line 8, where our aliasing analysis has computed the sharing graph `{ {array, arrays.0} }`. Because we do not have call mode information for components of parameters, we must assume that `arrays.0` is non-unique in the caller, and therefore use the call mode ⟨N⟩ to call `array_add`. This is sub-optimal if all of the arrays are known to be unique at the call site.

To solve this, we consider the interestingness of all parameter components, and change call signatures to refer to components as well as top-level parameters. To avoid infinite call signatures, we use the naming scheme described in Section 8.2.8, which type folds names with ancestors of the same type into those ancestors. Call signatures now consist of a sequence of the interesting parameter components, similar to Names. Therefore, the call signature of `element_array_add` is ⟨0.0⟩, meaning that the elements of the first parameter are interesting.

Similarly, call modes are now vectors of Names. The call to `array_add` has call mode ⟨{arrays.0}⟩ — it is destructive if and only if `arrays.0` was called with mode D. The recursive call to `element_array_add` also has call mode ⟨{arrays.0}⟩.

7The `array_add` function appends a single element to the end of the given array. It has the call signature ⟨0⟩, being able to destructively update the array.
10.3.4 Formal definition

We now formally define call signatures and call modes, and give the algorithms for computing the call signature of a function and the call mode of a call.

A ParamComponent refers to a particular component of a parameter of the current function. The first part of a ParamComponent denotes the parameter position (where 0 refers to the first parameter), and subsequent parts denote child components of the argument. As with Names (§8.3.1), these are type-folded, so any ParamComponent \( p \) also refers to all of the descendant components with the same type as \( p \). A CallSignature is a sequence of ParamComponents.

\[
\begin{align*}
\text{ParamComponent} & \rightarrow \text{Int} \\
& \quad | \text{ParamComponent} \cdot \text{Int} \\
& \quad | \text{ParamComponent}[] \\
\text{CallSignature} & \rightarrow \langle \text{ParamComponent}^* \rangle
\end{align*}
\]

We speak of argument modes and call modes in two varieties: \textit{values} and \textit{expressions}. These are analogous to values and expressions in ordinary programming: call mode \textit{expressions} may include conditions dependent on function parameters, while call mode \textit{values} are known sequences of \( \text{N} \) and \( \text{D} \). As a result of the context-insensitive analysis of a function, the calls are annotated with call mode expressions. At some later time (either during a context-sensitive analysis or at runtime), the call mode expressions are \textit{resolved} (evaluated) and replaced with call mode values.

An ArgModeValue is either \( \text{N} \) (non-destructive, denoting that the argument may not be modified), or \( \text{D} \) (destructive, denoting that the argument may be destructively updated). A CallModeValue is a sequence of ArgModeValues.

\[
\begin{align*}
\text{ArgModeValue} & \rightarrow \text{N} | \text{D} \\
\text{CallModeValue} & \rightarrow \langle \text{ArgModeValue}^* \rangle
\end{align*}
\]

An ArgModeExpr is either \( \text{N} \) (non-destructive) or conditional based on the argument mode value of zero or more parameters. Each element of a condition is a Name (specifically, an element of a restricted subset of Name, ArgModeComponent), denoting either the name of a formal parameter of the current function, or a component thereof. Again, names are type-folded, so any ArgModeComponent \( p \) also refers to all of the descendant components with the same type as \( p \). A CallModeExpr is a sequence of ArgModeExprs.
10.3. CALL MODE SELECTION

\[
\begin{align*}
\text{ArgModeComponent} & \rightarrow \text{Var} \\
& \quad | \text{ArgModeComponent} \cdot \text{Int} \\
& \quad | \text{ArgModeComponent}[] \\
\text{ArgModeExpr} & \rightarrow \text{N} \\
& \quad | \{\text{ArgModeComponent}^*\} \\
\text{CallModeExpr} & \rightarrow \langle\text{ArgModeExpr}^*\rangle
\end{align*}
\]

ArgModeExprs are resolved to ArgModeValues by the following rules:

- The ArgModeExpr \text{N} resolves to the ArgModeValue \text{N}.

- A conditional ArgModeExpr \{a_0, \ldots, a_{n-1}\} resolves to the mode \text{D} if, and only if, all of the components \text{a}_0 through to \text{a}_{n-1} have mode \text{D}. Otherwise, it resolves to \text{N}.

There is no explicit ArgModeExpr \text{D} in the formal grammar, because the empty condition \{\} is equivalent. (\{\} trivially resolves to the ArgModeValue \text{D}.) However, we use \text{D} as an alias for the ArgModeExpr \{\}. A single context-insensitive analysis computes the call signature of each function and the call mode expression of each call.\(^8\) Algorithm 6.2 is used to perform an efficient bottom-up fixed point analysis. Recall from Section 6.7 that each function is analysed after all of its callees, and any recursive function or mutually recursive group of functions is analysed repeatedly until a fixed point is reached. Initially, the call signature for each function is \langle\rangle (no parameters are interesting). Therefore, during the analysis of each function, we can assume that the call signature of all callee functions is known.

Algorithm 10.3 computes the call mode for each call in a function, assuming that the call signature of all callees is known. Recall that @ is the name corresponding to the static program data that can never be mutated. Note that this algorithm will compute the argument mode expression \{\} (\text{D}) whenever a parameter and all of its aliases (if any) are \text{DEAD}. This algorithm assumes that all calls are to known functions (it does not handle higher-order functions that make calls to variables). Such functions are discussed in Section 10.6.

Algorithm 10.4 computes the call signature of a function, assuming that the call mode of each call in the function has already been computed by Algorithm 10.3. This algorithm simply collects all of the arguments mentioned in the call modes of the function (by argument position). The final step sorts the ParamComponents. The order is not important; it is just necessary for a CallSignature to have some order, for canonical representation.

\(^8\)In practice, this can be done during the same bottom-up analysis that computes aliasing information. This is beneficial, because aliasing information at each program point can be used immediately by the call mode analysis and then thrown away, instead of having to be stored between passes.
In the next two sections, we describe two orthogonal approaches for resolving call mode expressions into call mode values with known \{D, N\} argument modes. In Section 10.4, we describe an approach that resolves call modes at runtime, by dynamically tracking the mode of interesting formal parameter components. In Section 10.5, we describe a separate context-sensitive static analysis that produces specialised versions of functions at compile time. These two approaches can be used together, with a heuristic to determine which functions or arguments should use specialisation, and which should use dynamic mode tracking.

### 10.3.5 The one-mode-per-call-site problem

A downside of our data model is that each static call site can have only a single call mode expression. This prevents the compiler from taking advantage of knowledge about control flow. Consider Example 10.7, which has a call to array_replace with a conditional call mode inside a loop. If array_increment is called with call mode \langle D \rangle, it will only perform destructive updates, taking $O(n)$ time. However, if it is called with call mode \langle N \rangle, it will copy the array on every iteration, taking $O(n^2)$ time!
Example 10.7 Conditional destructive operation inside a loop

```python
def array_increment(a :: Array(Num)):
    i = 0
    while i < array_length(a1) [a1 = φ(a, a'_1), i1 = φ(i, i'_1)]:
        v0 = array_ref(a1, i1)
        v1 = v0 + 1
        a'_1 = array_replace(a1, i1, v1)
        i'_1 = i1 + 1
    return a1
```

Even if it must take a copy of $a$ on the first iteration, there is no need for `array_increment` to take a copy on every subsequent iteration. Ideally, it would call `array_replace` with $\langle\{a\}\rangle$ on the first iteration, and $\langle\mathcal{D}\rangle$ on the rest, but there is no way to specify this.

The solution is for the compiler to automatically unroll the loop. In this example, $a_1$ is assigned $\phi(a, a'_1)$. Since $a$ is a parameter, but $a'_1$ never has any aliases, the compiler should duplicate the body of the loop, using $a$ in the first body and $a'_1$ in the second. The call to `array_replace` in the second body will always have mode $\langle\mathcal{D}\rangle$, so `array_increment` will always have $O(n)$ behaviour. However, we have not implemented this functionality in the compiler, so users are currently required to manually unroll the loop.

### 10.4 Dynamic call mode vectors

We have so far described a static system for representing the flow of “safe mutability” information through function calls. A function’s call mode signature describes the parameter components for which the function expects mutability information. A call mode value gives the mutability information for a particular function call, and a call mode expression represents a computation that determines the call mode value at a later time.

The obvious approach for implementing this system is to actually pass the mutability information at runtime, using a bit vector to represent call mode values. This is the approach taken by Lee et al. [59]; our approach is similar but more expressive (the two approaches are compared at the end of this section). Under this approach, a call mode expression is an actual expression to be evaluated at runtime, and a call mode value is an actual value in memory. The approach is described as part of a code generator that takes MIR and produces machine-level code (represented here in C).

---

9While we could describe the approach as a source-to-source transformation that produces MIR, this would be prohibitively slow in practice, because Mars does not feature any bit-level operations. Therefore, we chose to make the code generator aware of call signatures and call modes, and describe the output in a C-like language. Note that the Mars compiler actually emits LLVM code, but C was chosen for readability.
The code generator is modified such that each global function with a non-empty call signature takes an additional parameter, \texttt{callmode}. This parameter is a 32-bit integer (C type \texttt{uint32_t}), representing up to 32 Boolean values, one per interesting parameter component of the function’s call signature. If the call signature has more than 32 parameter components, additional \texttt{callmode\_n} parameters may be added (but for simplicity, here we assume that all functions have a call signature of at most 32 parameters).

The \texttt{callmode} argument encodes the call mode value as a bit vector. The first argument mode value is encoded in bit #0, the second in bit #1, and so on. The argument mode value \texttt{N} is encoded with bit value 0; \texttt{D} is encoded as 1.

Each built-in function is implemented in the low-level machine representation, and hand-written to accept the \texttt{callmode} parameter and interpret it accordingly. For example, the implementation of \texttt{array\_replace} has an if statement that tests whether \texttt{callmode} is non-zero, and if so, destructively updates the array instead of copying it.

Each call to a function with a non-empty call signature is augmented with an expression to compute the call mode value and pass it to \texttt{callmode}. Algorithm 10.5 shows how a call mode expression is compiled into machine-level code (represented with C code).

\begin{algorithm}
\caption{Compiling a call mode expression $c$ into machine-level code}
\begin{verbatim}
const_part = 0
for each ArgModeExpr $a \in$ CallModeExpr $c$ (with 0-based index $i$):
    if $a = \texttt{N}$:
        \(\triangleright\) do nothing (implicitly taking the value 0)
    else if $a = \{\}$:
        \(\triangleright\) this is argument mode \texttt{D}
        const_part = const_part + 2^i
        \(\triangleright\) put a 1 in bit position $i$
    else:
        for ArgModeComponent $n \in a$ (with 0-based index $j$):
            $p = \text{0-based parameter component index for } n$
            $N_j = [(\text{callmode } \& \ 2^p) \gg (p - i)]$
            \(\triangleright\) or $[\ll (i - p)], \text{if } i > p$
            $A_i = [(N_0 \& \cdots \& N_{|a-1|}]$
            $C = [\text{const_part} \cup | A_0 | \cdots | A_{|c-1|}]$
            \(\triangleright\) omitting any undefined $A_i$
\end{verbatim}
\end{algorithm}

An explanation of Algorithm 10.5 follows. Each argument mode expression in the call mode expression is either constant (\texttt{N} or \texttt{D}) or conditional. The constant argument modes contribute to the \texttt{const_part}, which, by the end of the algorithm, contains a bit vector with a 1 in the bit position of each component that is unconditionally destructive. Each conditional argument mode results in the construction of an expression $A_i$ for execution at runtime. The first step involves obtaining the Boolean value of each relevant condition variable, using a constant mask with \& (bitwise AND), followed by a $\gg$ (bitwise right shift with zero-extension) or $\ll$ (bitwise left shift), to shift the Boolean value
into bit position $i$. The second step decides whether to perform a destructive update for
that argument mode ($D$ if and only if all of the condition variables are $D$), by using $\&$ (bitwise AND) on all of the condition values. The final call mode bit vector is computed
using $|$ (bitwise OR) to combine the bit patterns of the $\text{const\_part}$ and all of the computed
conditional modes. Obviously, if $\text{const\_part}$ is 0 and there is at least one conditional mode,
the $\text{const\_part}$ can be omitted from the expression.

For example, consider the following call and call mode expression:

$$\text{callee}(a, b, 13, c, d) \langle D, \{x, y.1[]\}, N, D, \{x\} \rangle$$

where $x$ is parameter component $\#0$ and $y.1[]$ is parameter component $\#3$. This call trans-
lates into the following machine-level code:

```c
callee(0x09 | 
((callmode & 0x1) << 1) & ((callmode & 0x8) >> 2)) | 
((callmode & 0x1) << 4), 
a, b, 13, c, d);
```

This example generated a complex expression, but many cases result in far simpler
expressions. In particular, if the call mode expression has no conditions, the argument
is an integer constant. If the call mode expression has a single conditional value, the
argument reduces to a bitwise AND and (possibly) a bit shift.

There are a number of possible low-level optimisations on the generated code which
have not been explored. The viability of these optimisations would have to be carefully
assessed, based on the target architecture and any compiler optimisations applied to the
generated code.

- The initial AND mask on $\text{callmode}$ is not needed if the caller’s call signature has
  a single component.

- Logical AND ($\&\&$) could be used instead of bitwise AND to compute conditional
  argument mode expressions. This would take advantage of short-circuit logic, but
  may result in additional CPU branching. In cases where a logical AND is used
  ($|a| > 1$), the bit shift for each $N_i$ would be omitted (since $\&\&$ implicitly shifts its
  output into bit $\#0$), and instead, a single left shift $[<< i]$ would be applied to $A_i$.

Note that the latter optimisation only matters when an argument mode is conditional
on more than one component, which ought to be rare (the majority of argument mode
expressions are constant or conditional on a single component).
CHAPTER 10. DESTRUCTIVE UPDATE OPTIMISATION

All of this applies only to direct function calls to global functions, not calls to closures. It is much more difficult to support higher-order programming, because type-compatible functions at the source level may have different call signatures, and therefore become type-incompatible when the above transformation is made in the code generator: we cannot call a closure with any call mode, because we do not know what the underlying function’s call signature is, or even if it accepts a callmode bit vector at all.

Fortunately, the fact that closures cannot accept call modes does not force us to change the way static functions are represented, because of a layer of indirection (closure templates). For now, closure templates assume that all of their arguments are non-interesting, even if the functions they call have interesting arguments. In Section 10.6, we discuss potential improvements to higher order programs with respect to call modes.

10.4.1 Comparison to runtime reference counting

The dynamic call mode approach is somewhat similar to runtime reference counting: both schemes involve managing some extra information at runtime to track the uniqueness of certain variables, and using this information to decide dynamically when to garbage-collect or reuse the memory. Note that a standard runtime reference count alone does not provide enough information to perform destructive update, because it does not take into account whether a variable is LIVE or DEAD. Therefore, it would need to be coupled with a liveness analysis to decrement the reference count as soon as possible (see [53], discussed in Section 7.8) in order to fulfil the role of our dynamic call mode system.

The runtime reference counting system would be much simpler than our approach, as it would not have a static aliasing analysis. It would also perform destructive update in a superset of the situations in which our approach works. However, it has several major disadvantages. From a performance standpoint, it appears to have a lot more runtime overhead than our dynamic call mode approach: it requires a memory read and write every time a pointer is copied or goes out of scope, whereas the overhead in our approach is just extra integer function arguments.

Also, the reference counting scheme has far less predictable performance than ours, because the same code path can produce vastly different performance characteristics depending on runtime conditions. In our approach, the same code path always performs the same destructive updates, making it easier to reason about the behaviour. In addition, because our system is based on static information, we can tell the user at compile time exactly what circumstances result in destructive update.
10.4.2 Related work

Dynamic call modes were described in a much simpler form by Lee et al. [59] (described in detail in Section 7.5). Their system was described for a language with unary functions, where the only data structure is a binary tree. The “call mode” is always a pair of Booleans: \( \beta \) represents the mutability of the top-level memory cell of the tree argument and \( \beta_{ns} \) represents the descendant memory cells. In our terminology, each function’s call signature has exactly two elements. The advantages of our system are an applicability to general data structures, a more accurate representation (compartmentalising mutability information about differently typed nodes of the data structure) and less time wasted computing call mode values for structures that cannot be updated (due to our interestingness analysis).

10.5 Multiple specialisation

An alternative approach is to resolve call modes at compile time with specialisation. Here, the compiler generates several versions of each procedure, potentially (but not necessarily) one for each possible call mode value. The specialised versions are hard-coded to perform destructive update, so there is no runtime overhead in checking whether it is allowed or not. At compile time, the call mode expression for each call is resolved, and the correct specialised version is selected.

For example, the built-in function `array_replace` is hand-written to take a 1-bit call mode vector, and uses the value of this bit to decide whether to create a shallow copy of the array. However, we can also provide two specialised versions, `array_replace_d` and `array_replace_n`. If we find a call to `array_replace` with mode \( \langle D \rangle \), we can replace it with a call to `array_replace_d` with mode \( \langle \rangle \), saving the overhead of generating, passing, and selecting on the call mode at run time.

For each function, there are several approaches for choosing which call mode values generate a specialised version:

- Generate a specialised version for all possible call mode values. This will generate \( 2^n \) versions of each function, where \( n \) is the size of the function’s call signature.

- Generate just two specialised versions: one with all argument mode values set to \( N \), and one with all set to \( D \). This avoids exponential growth in the size of the generated code, but it means that we cannot take advantage of destructive update unless all of the interesting parameter components are unique. This is the approach taken by Bruynooghe [11], Shankar [82] and Mazur [65].
Generate specialised versions on demand. When any call to a procedure is made, the call mode value is resolved, and at that point, the compiler generates the specialised version for that call mode value, if one does not already exist. This approach can potentially generate $2^n$ versions of any given function, but typically generates fewer.

We have chosen to generate special versions on demand, because the first approach has unacceptable code size, and the second approach was too limited (it would defeat the whole optimisation for any function that could potentially perform destructive update on $n$ arguments, but fewer than $n$ arguments were resolved to the mode $D$). This is the most complex approach, as it requires a simple context-sensitive static analysis to determine which call mode values are required.

A context-sensitive (or top-down) analysis requires that the compiler knows all call sites of a particular procedure before analysing it. This means that inter-module optimisation is difficult: in a program with multiple modules, either the whole program must be analysed as a single unit, or multiple specialisation cannot work across module borders. Fortunately, this is not an issue for Mars, because it always compiles all of the imported files as a single unit. However, this would be an issue for a more serious programming language. Note that this multiple specialisation optimisation is the only part of the entire compiler that is context sensitive.

Algorithm 10.6 Selecting the specialised call modes for each function

```
versions = \{ (\text{main}, \langle\rangle) \}
repeat:
    for \( \langle f, \text{mode} \rangle \) in versions:
        for each call to \( c \) in \( f \) with CallModeExpr \( e \):
            \( v = \) resolution of \( e \) with respect to \( \text{mode} \)
            \( \text{versions} = \text{versions} \cup \{ \langle c, v \rangle \} \)
until \text{versions} has not changed in the last iteration
```

Algorithm 10.6 shows how the compiler selects the specialised versions of each function. This algorithm works by building up a set of (FuncName, CallModeValue) pairs, denoting the call modes that each function may statically be called with, from anywhere in the program.

The algorithm begins with the set \{ (\text{main}, \langle\rangle) \}, representing initially the only known call mode, a call to main with no argument modes. We then analyse each call mode

---

10 This approach is quite similar to template instantiation in C++. By this analogy, the call mode values are Boolean template arguments. The C++ compiler automatically generates any required versions of the function, on demand.

11 Note that this optimisation assumes that the program will be launched via the main function. This will
of each function in the \textit{versions} set, using the call mode value to the current function to resolve the call mode value to each callee. We add the call mode value of each callee to the \textit{versions} set. This is repeated until a fixed point is reached, which is guaranteed to be finite, because there is a finite number of functions and a finite number of possible call mode values for each, and \textit{versions} is monotonically increasing. The resulting \textit{versions} set represents all functions statically reachable from \textit{main}, and all possible call modes that may ever arise at runtime from a call to \textit{main}.

The code generator now creates a copy of the function for each element of \textit{versions}. For example, a pair \(\langle \text{pop}, \langle D \rangle \rangle\) results in a specialised function called \texttt{pop}_D. In the specialised copy, the call signature is always \(\langle \rangle\), and each conditional call mode expression is replaced with the resolved call mode value. All built-in functions have a hand-coded specialised version for all call modes. In a separate pass, all non-conditional call mode expressions are replaced by a call to the specialised version, if it exists, with an empty call mode expression. For example, a call to \texttt{pop} with mode \(\langle D \rangle\) is replaced with a call to \texttt{pop}_D with mode \(\langle \rangle\). After this transformation, all calls reachable from \textit{main} will be to specialised versions and have an empty call mode.

\subsection{10.5.1 Specialisation and dynamic call mode vectors}

The algorithm in the previous section was carefully designed so as not to remove call mode expressions from the generated code, instead replacing most of them with the empty tuple. This is to ensure that a specialised program is still compatible with the dynamic call mode vector code generator described in Section 10.4. This is useful because it means we do not need to globally choose between specialisation and dynamic call mode vectors: we can tune the specialisation algorithm to back off in certain situations and let the dynamic vectors do their work.

If specialisation is not applied at all, the default “dynamic” version (with conditional call mode expressions) will be used. Note that the specialisation algorithm does not remove the dynamic version of a function; it merely creates specialised copies. To this end, the non-destructive version of a function is not always present — it is compiled only if the function needs to be called in a non-destructive manner. Also note that we do not need to modify Algorithm 10.5 to deal with specialised versions: it simply does not add any runtime overhead to them because their call signature and all of their call modes are empty.

This infrastructure means that we do not need to aggressively specialise all functions not be true if the user manually runs a function via the interactive prompt. In this case, it will not be possible to analyse the call mode in advance, and Mars will fall back to using dynamic call modes.
to get the benefit of destructive update. We can use a heuristic to determine when to specialise, trading off runtime performance for compiler performance and compiled code size. We have not experimented in any detail with specific heuristics, and suggest that this would make a good topic for future research. Some obvious ways to cut back the specialisation are:

- Place an upper bound on the number of versions generated for each function; if that limit is exceeded, arbitrarily drop some versions and let them be called dynamically.

- Simply generate a version with all argument mode values set to \( N \), and one with all set to \( D \), and let all other versions be called dynamically.

- Use profiling to dynamically determine which call modes are most frequently used, and specialise those.

- If Mars supported separate compilation, it may be an acceptable strategy to simply perform no inter-module specialisation beyond the two obvious modes.

In addition, the dynamic version of a function could be augmented to first switch over the call mode bit vector, and if it matches a specific pattern, call the specialised version of the function. Without this optimisation, any call to a dynamic function would result in non-specialised calls all the way down.

This section is not intended to contain the answers about when it is appropriate to specialise, but rather to demonstrate that a hybrid dynamic/specialised approach is possible due to the way we have defined the specialisation algorithm.

To give a brief idea of the performance trade-off, we benchmarked a program that repeatedly inserts integers into a hash table using `array_replace` with the call mode \( \langle D \rangle \). The program and benchmarking environment are described in detail in Section 11.3. We compared a version built with the LLVM back-end that uses dynamic call mode vectors against a manually optimised version that explicitly uses destructive update (to simulate specialisation). The latter does not use any call mode parameters at runtime. Both the dynamic call mode and specialised versions took 8.7 seconds to insert 20 million items. This is not surprising, since passing and processing the call mode bits is a relatively cheap operation. However, there may be more savings in a program that actually features complex call mode expressions.
10.6 Higher-order call modes

In this chapter so far, we have not addressed higher-order programming. Specifically, we have not discussed how closure templates or calls to closures are to be annotated and handled by the code generator. We have so far simply assumed that all calls to closures are non-destructive. Now, consider the function \texttt{array\_apply} in Example 10.8.

Example 10.8 Definition and usage of higher-order array function

```python
def \texttt{pop}(\texttt{stack} :: \texttt{Array}(a)):
    \quad i_0 = \texttt{array\_length}(\texttt{stack})
    \quad i = i_0 - 1
    \quad \textbf{return} \ \texttt{array\_remove}(\texttt{stack}, i)

def \texttt{pop}_0(\texttt{stack}):
    \quad \textbf{return} \ \texttt{pop}(\texttt{stack})

def \texttt{array\_apply}(f :: \texttt{Array}(a) \rightarrow \texttt{Array}(a), x :: \texttt{Array}(a)):
    \quad \textbf{return} \ f(x)

a = [1, 2, 3]
p = \texttt{pop}\{\}
array\_apply(p, a)
```

When compiling \texttt{array\_apply}, we do not know the call signature of \( f \). The call signature of a function cannot be derived from its type, for two reasons. Firstly, the type does not tell us which parameter components are interesting, and secondly, because of polymorphism, we do not know the exact structure of the function’s arguments anyway. For example, the \texttt{id} function could be passed to \texttt{array\_apply}, which has different interestingness to \texttt{pop}.

The simplest solution to this problem is to assume no mutability when calling a closure. The closure template \texttt{pop}_0 has a call signature of \( \langle 0 \rangle \), despite the interestingness of the \texttt{stack} parameter, and the call to \texttt{pop} is made with call mode \( \langle N \rangle \). All calls to a closure (such as \( f(x) \)) have an empty call mode. The opportunity to mutate \texttt{stack} is lost. Note that under this scheme, closure templates may still accept a full call mode for their partially applied closure variables (the closure template reification statement may supply a call mode, which would be stored in the closure). It is only the function parameters that are not allowed to accept a mode.

However, there is a better solution — not as strong as the first-order case, but an improvement on the above scheme. For closure calls, we assign exactly one Boolean value to each parameter. If a parameter has a more complex structure, all of the components are folded into a single mode Boolean. Furthermore, all parameters are considered interesting. As such, all unary closures have call signature \( \langle 0 \rangle \), all binary closures have call
signature \langle0,1\rangle, and so on. We do not perform multiple specialisation for closure calls, instead always using dynamic call mode vectors. The closure template then “translates” this simplified call mode into the real call mode expected by the function.

In this example, the closure template \texttt{pop\_0} has call signature \langle0\rangle, and calls \texttt{pop} with mode \langle\{stack\}\rangle. The function \texttt{array\_apply} has call signature \langle1\rangle, and calls \texttt{f} with mode \langle\{x\}\rangle (which is safe to do because all unary closures have call signature \langle0\rangle). If we used the id function instead, the closure template \texttt{id\_0} would also accept a Boolean argument mode, for compatibility, but it would ignore it, as the function id does not mutate its argument.

This approach yields strong optimisations for closure calls with simple arguments, but loses fidelity when structured arguments are introduced. For example, consider a function \texttt{pop\_all :: Array(Array(a)) \rightarrow Array(Array(a))}, with call signature \langle0,0[\]\rangle. If this function was passed to \texttt{array\_apply}, the components 0 and 0[\] would be combined into one Boolean value, for compatibility with the unary closure call signature, and therefore, if the outer array is non-unique, the inner arrays cannot be destructively updated (even though they may be unique).

### 10.7 Structure reuse

So far, we have only optimised calls to built-in functions such as \texttt{array\_replace}, which have been hand-written to perform destructive update when called with a specific call mode. In this section, we look at optimising calls to data constructors. These are quite different to other function calls, in a number of important ways:

- Constructor calls may not be \texttt{hinted}, which means that there is no obvious variable to attempt to reuse, and we must heuristically try to find one. The fact that many variables can potentially be reused by the same call means there is a much wider scope for optimisations.

- The compiler knows that a constructor will only reuse the top-level memory cell of a given variable, so we can relax the type folding rule somewhat.

The second point here is important for dealing with recursive types. We have used a “type folding” scheme for both the aliasing analysis and call modes, to ensure our compile-time data structures have finite size. This scheme creates an unfortunate problem for recursive types, which we need to work around with some additional rules.

Consider Example 10.9, a simple function that modifies the first element of a pair. As usual, this is semantically a non-destructive operation: it returns a new pair without
mutating the original. However, in the case that \( p \) is unique, we would like to save an allocation and garbage collection by reusing its memory cell.

Example 10.9 MIR function that modifies the first element of a pair

```python
def replace_fst(p :: Pair(a, b), f :: a):
    s = pPair/1
    return Pair(f, s)
```

This code may have been generated from either of two different Mars source expressions: a) Pair(\( f, p.snd \)), or b) \( p.fst := f \). In the latter case, the programmer has explicitly stated their intention to update the \( p \) variable, and the resulting MIR code will reflect that intention with a hint annotation on the call to the Pair constructor: \( \{ p \} \). We use the hint to tell us to attempt to update the variable \( p \). In the former case, where there is no hint, we heuristically select a variable to attempt to reuse, which in this case is also \( p \).

Once we have selected a variable to reuse, we compute a call mode expression for the constructor, just as with a call to a regular function. Constructor calls always take a single call mode argument, which determines whether it is safe to reuse the hinted variable’s memory cell. We use the same algorithm as the call mode selection to determine whether it is safe to mutate \( p \), based on the aliasing and liveness information for \( p \). In this case, \( p \) is a parameter, so the safety of mutation is conditional on this function’s call mode. Therefore, we annotate the call to Pair with the call mode expression \( \langle \{ p \} \rangle \). The call has a hint of \( \{ p \} \), regardless of whether it previously had a hint. Note that if the original hinted element is not reusable, our algorithm heuristically chooses a different variable, if possible, and updates the hint to the new variable.

The fact that the call mode is conditional on \( p \) also makes \( p \) interesting, so the call signature of replace_fst is \( \langle 0 \rangle \).

The code generator uses the hint and call mode expression to turn the constructor call into a destructive update. The dynamic call mode vector version includes an if statement that inspects the call mode bit vector, and if it contains a 1 in the \( p \) position, it modifies \( p \) instead of calling Pair. The multiple specialisation may result in a special destructive version being generated, as shown in Example 10.10.

Example 10.10 Destructive version of Example 10.9

```python
def replace_fst_d(p :: Pair(a, b), f :: a):
    s = pPair/1
    pPair/0 =! f
    pPair/1 =! s
    return p
```

\( \triangleright \Omega \{ f \} \)

\( \triangleright \Omega \{ s \} \)

\( \triangleright \Omega \{ p \} \)
CHAPTER 10. DESTRUCTIVE UPDATE OPTIMISATION

Note the new syntax: the \( =! \) instruction destructively updates a field of an object, and can only be generated by our optimisation.\(^\text{12}\) The generated destructive version of \texttt{replace\_fst} avoids an allocation and garbage collection, but it still has an inefficiency: Line 2 reads a value into \( s \) only to have Line 4 write it back to the second element of the pair; both lines can be removed. While we have not implemented a solution to this problem, it should be fairly easy to optimise away, by using a simple static analysis that records which local variables hold copies of other variables’ fields.\(^\text{13}\)

We must also take care that we do not attempt to reuse the same variable more than once. Consider Example 10.11, which features both a call to a potentially destructive function (\texttt{replace\_fst}, defined above) and a constructor call.

\begin{example}
\textbf{An attempt to reuse a memory cell twice}
\begin{verbatim}
def double_reuse(x :: Pair(a, b), y :: a, z :: b):
    fst = x.

3:   \( t_1 = \text{replace\_fst}(x, y) \) \( \triangleright \Omega\{x, y\}; \{\{x\}\} \)
4:   \( t_2 = \text{Pair}(fst, z) \) \( \triangleright \Omega\{fst, z\} \)
5:   \text{return} \text{Pair}(t_1, t_2) \( \triangleright \Omega\{t_1, t_2\} \)
\end{verbatim}
\end{example}

On Line 3, \( x \) is passed to \texttt{replace\_fst}, with call mode \( \{\{x\}\} \). If \( x \) is mutable, then it will mutate \( x \) and \( t_1 \) will be a reference to the same memory cell as \( x \). The sharing graph after Line 3 is:

\( \{\{fst, x.0\}, \{t_1.0, y\}, \{t_1.1, x.1\}\} \)

Note that there is no alias between \( x \) and \( t_1 \), because \( t_1 \) is semantically a fresh object. The simple algorithm described above would choose \( x \) for reuse on Line 4, because it is not aliased, but this would be a mistake, because it would result in \( t_1 \) and \( t_2 \) being aliases. We need to remember that we may have already reused \( x \) on Line 3, so that we do not attempt to reuse it again on Line 4. This particular constructor call cannot be optimised. In general, whenever we potentially reuse a variable, it must be blacklisted and not reused again.

However, this approach can be too restrictive in certain situations, specifically, when recursive types are concerned, as alluded to earlier. Consider the \texttt{append} function in Example 10.12. This is a straightforward recursive version of \texttt{append}, which uses the structural induction pattern on its first argument \( x \) — it is typical of a wide variety of structural induction algorithms on many data structures, including map, filter, substring, and binary tree insertion operations. Correctly performing automatic destructive update

\(^{12}\)Our Mars implementation also gives the programmer direct access to the \( =! \) operator (§4.2.7), but this should not be considered part of the language.

\(^{13}\)LLVM’s optimiser will automatically eliminate both the read and write if the write immediately follows the read, but a higher level analysis could fix this in the general case.
on this pattern is the key to optimising updates to non-array-based data structures.

**Example 10.12** The `append` function in MIR

```python
1. def append(x :: List(a), y :: List(a)):
2.     switch x:
3.         case Nil:
4.             r = y ▷ Ω{y}
5.         case Cons:
6.             h = x.Cons/0 ▷ Ω{x}
7.             t = x.Cons/1 ▷ Ω{t, y}; ⟨{x}⟩
8.             t1 = append(t, y) ▷ Ω{t, y}; ⟨{x}⟩
9.             r = Cons(h, t1) ▷ Ω{h, t1}
10.        return r ▷ Ω{r}
```

The ideal result is a specialised version of `append` that, when the first argument has mode D, iterates down the spine of `x` until it finds the Nil tail, and modifies it to point at `y`, making no new memory allocations. If we assume that `append` has call signature ⟨0⟩, we see that there is no problem recursively calling it with the destructive mode ⟨{x}⟩. Note that, due to type folding, `x` and `t` are aliased. However, because `x` is DEAD-PARAMETER before the call, it is safe to mutate `t` if the caller gave us permission to mutate `x`. However, in order to make use of this permission, the call to Cons must be able to reuse the dead memory cell of `x`. After Line 7, the sharing graph (after type folding) is:

\[
\{ \{h, x.0\}, \{t, x\}, \{t.0, x.0\} \}
\]

Unfortunately, due to the rule we introduced above, we cannot reuse `x`: since we have already reused `t`, `x` will have been blacklisted from reuse (as an alias of `t`). On the surface, this seems rational: we were allowed to modify `t` on Line 8 because its alias, `x`, was DEAD-PARAMETER, and therefore `t1` may reference the same memory cell as `x` did; therefore, as with `double_reuse`, we cannot reuse `x`.

The flaw in this argument is that in `append` (unlike in `double_reuse`), `t` is not a *direct* alias of `x`; it is `x`’s child. It is *not* generally safe to pass `x` to an arbitrary function, because it contains dangling pointers to reused cells, but we can make a special exception for constructor calls, because they only touch the top-level cell of `x`. Therefore, even though our type-folded aliasing analysis tells us “`x` may be aliased to `t`,” as a special rule, if we can show that `x` does not refer to a memory cell reachable from `t`, then it is safe to reuse `x` in a constructor call. We generalise this in Rule 10.1.
Rule 10.1. It is safe for a call to constructor C to reuse the memory cell referred to by a variable X if and only if

1. X is known to have been constructed with the constructor C, and
2. X is DEAD, or X is DEAD-PARAMETER and the call mode permits the mutation of X, and
3. at the time of X’s death, all live variables that it was aliased to were known to have been descendants of X, and
4. X has not previously been passed as an argument to a potentially destructive operation, or reused by a constructor call.

Condition 1 can be tested using the static analysis defined in Chapter 6, which determines the constructor used to construct a given variable. Determining Condition 3 begins with the sharing graph, but requires some new static knowledge: whether or not a given variable is known to be a descendant of another. Our sharing graph does not record such information — in the append example, it tells us that x and t, or components thereof, are aliased, but it cannot tell us whether x is an ancestor of t, t is an ancestor of x, or whether they are the same cell. The type folding applied to recursive types makes our aliasing analysis too weak to show that reuse of x in append is safe.

To test whether this rule applies and therefore x may be reused in this special case, we introduce two new static data structures known as the descendant set and the free list.

10.7.1 The descendant set and the free list

A simple static analysis is used to compute two new static data structures. These analyses are local — unlike the aliasing analysis, this does not propagate information across function calls, and is therefore more limited. It is designed solely to determine the above rule in certain cases.

The descendant set is a mapping from variable names to sets of variable names. Each variable x maps onto a set \{t_1, \ldots, t_n\}, indicating that x is definitely a strict ancestor of each of the variables t_1 through to t_n. This information serves as an addendum to the sharing graph: if the sharing graph contains a node \{x, t_1\}, the descendant set tells us that despite the sharing graph, x is not reachable from t_1, so it may be safe to reuse x.

Only field-reference instructions contribute to the descendant set. Therefore, for example, the instruction \(t_1 = x\).Cons/1 adds \(t_1\) to the descendant set for x, whereas the instruction \(t_1 = \text{tail}(x)\) does not (because descendant information does not propagate across function calls). The analysis is specifically designed to work with code generated by switch statements and field-reference expressions in the source language.
The *free list* allows us to track which variables are eligible for reuse over time. The free list is a collection of \( \langle \text{Var}, \text{ArgModeExpr} \rangle \) pairs that satisfy Rule 10.1.3. Each item \( \langle v, m \rangle \) indicates that the variable \( v \) is eligible for reuse if argument mode \( m \) can be satisfied (where \( m \) is either \( D \) or a set of parameter names, and never \( N \)). It is conceptually a set, but we consider it to be a list, to ensure deterministic results of the variable selection algorithms.

Whenever a variable \( x \) becomes *Dead* or *Dead-Parameter*, the descendant set for \( x \) is consulted. If every *Live* variable that is aliased to \( x \) also appears in \( x \)’s descendant set, and \( x \) is not being used in a potentially destructive update operation, \( x \) is appended to the free list. If \( x \) is aliased to one or more parameters (or is one), the set of parameters are given as the argument mode on the free list; otherwise, the argument mode is \( D \). This makes \( x \) eligible for reuse, despite the fact that any of its aliases may also be reused in the mean time.

### 10.7.2 Selecting a variable for reuse

We now apply a static analysis to each constructor call in a function. This selects which variable (if any) is to be reused, and the call mode expression that governs the condition of its reuse. Algorithm 10.7 describes the analysis, which ensures that all constructor calls satisfy Rule 10.1.

**Algorithm 10.7** Selecting a variable for reuse by a call to constructor \( C \)

\[
\text{filtered\_free\_list} = \{ \langle v, m \rangle : \langle v, m \rangle \in \text{free\_list} \land v \text{ was constructed with } C \} \\
\text{if } |\text{filtered\_free\_list}| = 0: \\
\quad \text{remove constructor call hint} \\
\quad \text{set constructor mode to } N \\
\text{else:} \\
\quad \text{if constructor call has hint } \{v_0\} \text{ and } \langle v_0, m_0 \rangle \in \text{filtered\_free\_list}: \\
\quad \quad \langle v, m \rangle = \langle v_0, m_0 \rangle \\
\quad \text{else:} \\
\quad \quad \langle v, m \rangle = \text{element of } \text{filtered\_free\_list} \langle v_0, m_0 \rangle \text{ with minimal } |m_0| \\
\quad \quad \text{replace constructor call hint with } \{v\} \\
\quad \quad \text{set constructor call mode to } \langle m \rangle \\
\quad \quad \text{mark the parameter corresponding to } v \text{ as interesting} \\
\quad \quad \text{remove } \langle v, m \rangle \text{ from free list}
\]

We first filter out any variables on the free list that are not statically known to have been constructed with \( C \). If the filtered free list is empty, we do not optimise the constructor call. Otherwise, we select the variable in the free list with the fewest constraints (ArgModeComponents). Ties are broken by whichever variable became available earli-
CHAPTER 10. DESTRUCTIVE UPDATE OPTIMISATION

This ensures that variables with mode $\mathcal{D}$ are prioritised over those with conditional modes, and variables with fewer conditions are prioritised over those with more.

As a special case, if a constructor already has a hint (because it was compiled from a field-replace expression) and the hinted variable is on the free list, we opt to reuse that variable, even if it imposes more conditions than another variable (we assume that the programmer knows that it is best to reuse that variable).

We must do two more things when we select a variable for reuse by a constructor call: first, make the corresponding parameter interesting, so that callers of this function will inform us when it is eligible for destruction, and second, remove the variable from the free list, so we do not reuse it again.

This simple heuristic may not be optimal, and we suggest the exploration of better heuristics in future work. In particular, selecting a variable with a conditional argument mode means that certain call modes may have been better off selecting a different variable. Solutions to this problem are discussed briefly in Section 10.7.3.

We now apply Algorithm 10.7 to append in Example 10.12. After Line 5, we know that $x$ was built with constructor Cons. After Line 7, $x$’s descendant set is $\{h, t\}$. When $x$ dies at this same program point, it has a single alias, $t$. Since $t$ is in $x$’s descendant set, we know that the top-level cell of $x$ is unique, and we add $\langle x, \{x\} \rangle$ to the free list. On Line 8, we annotate the recursive call to append with call mode $\langle \{x\} \rangle$, allowing the mutation of $t$. On Line 9, we search the free list for a variable of constructor Cons, and find $\langle x, \{x\} \rangle$. Therefore, we annotate the call to Cons with hint $\{x\}$ and call mode $\langle \{x\} \rangle$.

In the event that append is called with call mode $\langle \mathcal{D} \rangle$, it will allocate no new memory: each recursive call will use mode $\langle \mathcal{D} \rangle$, and all calls to Cons will reuse the memory cell of $x$. The specialised destructive version of append is shown in Example 10.13.

As with Example 10.10, Lines 6 and 9 are redundant, because they simply assign the same value to $x$.Cons/0 that it already contains. These could be removed by a simple analysis. Another source of inefficiency here is that for all but the last element of $x$, $t_1 == t$ and therefore the update statement $x$.Cons/1 $=! t_1$ is redundant. This is a much harder problem to optimise, requiring a deeper understanding and transformation of the program than our analysis provides. While we have not optimised away the redundant assignments, we have at least removed all memory allocations in the destructive version of append.
Example 10.13 Destructive version of Example 10.12

```python
def append_d(x :: List(a), y :: List(a)):
    switch x:
        case Nil:
            r = y
            \text{▷ } \Omega\{y\}
        case Cons:
            h = x.Cons/0
            t = x.Cons/1
            t_1 = append_d(t, y)
            \text{▷ } \Omega\{t, y\}
            x.Cons/0 =! h
            \text{▷ } \Omega\{h\}
            x.Cons/1 =! t_1
            \text{▷ } \Omega\{t_1\}
            r = x
            \text{▷ } \Omega\{x\}
        return r
        \text{▷ } \Omega\{r\}
```

10.7.3 Discussion

In this section, we have described a system for automatically converting calls to constructors (which would normally allocate new heap memory) with a definite or possible reuse of an existing memory cell. Unlike calls to built-in functions that may update data structures, we necessarily take a heuristic approach to constructor calls, because the programmer has not explicitly specified a variable to reuse. When the programmer does specify a reuse variable (via a field-replace expression), we honour that suggestion if it is at all possible, to maximise the amount of control the programmer has over the algorithm.

In Chapter 5, we opted to represent field-replace expressions as hinted constructor calls in MIR instead of as calls to synthesised functions. Having discussed the subtleties of reuse analysis, we can now justify this decision. If a field-replace was considered to be a call to a function, we would have to accept the general assumption that if any component of a variable \(x\) aliases a component of another variable \(t\), then neither \(x\) nor \(t\) may be destructively updated. As described above, this prohibits legitimate reuse in all structural induction cases, such as `append`. Representing field-replace instructions as constructor calls allows us to break that assumption and mutate \(x\) in the special case that \(t\) is known to be a strict descendant of \(x\).

This structure reuse system was partly inspired by Mazur’s [65] destructive update optimisation for Mercury. In that work, the compiler explicitly tracks which variables are available for reuse from a switch statement, and at a constructor call, it identifies a matching deconstruction-construction pair. This is analogous to our algorithm that finds a variable for reuse on the free list. Mazur’s version only detects that a variable is available for reuse if its last use is in a deconstruction (switch statement), whereas our system more generally tracks when the top-level memory cell of a variable is unaliased. It is also important to
note that this work focuses only on structure reuse, whereas we have developed a more
general system for describing reuse of arguments and their components, with structure
reuse as a special case. Furthermore, our system is more flexible in when and how it
selects whether to reuse a variable (with both dynamic and static selection), whereas
Mercury only generates a single static reuse version.

10.8 Future work

This chapter is the most open-ended in this thesis, because the task of selecting when
and how to reuse variables is quite heuristic in nature. We have described a number
of heuristics, many of which are alternatives, but we have not yet experimentally deter-
mined which settings are optimal. Furthermore, many of the algorithms described in this
chapter have not been implemented in our compiler.

We have a working implementation of Sections 10.2, 10.3, and 10.4, although the im-
plementation has just one call mode for each interesting parameter, omitting the struc-
tural call modes described in Section 10.3.3, and we have not implemented loop unrolling
described in Section 10.3.5. The three major features which have not been implemented
are multiple specialisation, higher-order call modes, and structure reuse, of Sections 10.5,
10.6, and 10.7, respectively. Each of these four enhancements is a good candidate for
future implementation work.

The free list selection heuristic described above is unaware of the current function’s
call mode, and is therefore unable to dynamically adapt to different parameters being
unique. For example, if the free list contains two viable entries, \(\langle x, \{a\} \rangle\) and \(\langle y, \{b\} \rangle\), the
heuristic essentially guesses. Whichever way it guesses, it will be sub-optimal for certain
call modes (e.g., if it chooses \(\langle x, \{a\} \rangle\), and then is called with \(b\) unique and \(a\) non-unique).
The problem could be solved by enhancing MIR. If we were to introduce a special “call
mode switch” statement, the generated code in the above example could select \(x\) for reuse
if \(a\) is unique, or \(y\) for reuse if \(b\) is unique, and otherwise allocate a new memory cell.
This would add some significant complexity to the infrastructure, but would allow the
specialised versions to select the appropriate variable to reuse in either case (and compile
away the switch statement), and allow the dynamic version to select which variable to
reuse at run-time. This idea is largely unexplored, and would make a good candidate for
future research.
10.9 Conclusion

In this chapter, we have presented a complete system for transforming calls to built-in functions and constructors in Mars programs into destructive update operations, making use of the precise aliasing analysis covered in previous chapters. The system described here presents various questions about how it might best be used, and these questions remain unanswered.

The underlying principle for deciding whether to perform destructive update is call modes. Each function (including built-in functions) has a call signature, which specifies which components of the function’s arguments might be destructively updated. Each function call is accompanied by a call mode expression, an expression whose value specifies which of those components are unique and are therefore allowed to be destructively updated. The call signatures and call mode expressions in a program are derived from the aliasing information.

We then require a mechanism for evaluating call mode expressions to decide whether each component is allowed to be updated. The first such mechanism is dynamic call mode vectors, in which the call mode expression is compiled into an expression to be evaluated at runtime. Under this mechanism, function calls involve a hidden extra argument, which contains information about which arguments may be destructively updated. The second such mechanism is multiple specialisation, in which each required call mode of a function is compiled into a separate version. Within each version, the call mode of each function call can be determined, so there is no need for a dynamic call mode vector.

We have fully specified the two mechanisms, but the main question deserving further research is how to decide between them. There is a non-trivial trade-off: the dynamic call mode vectors produce code with a runtime execution overhead, but the multiple specialisation approach produces larger code, and slows down the compiler (and potentially the execution too, due to the effects of caching a larger program). It is possible to decide between the two mechanisms on a per-function (or even per-argument) basis — any code that is specialised has no runtime call mode decisions, while any code that is not specialised simply has its call modes evaluated at runtime. Therefore, there is room for a heuristic to decide on a case-by-case basis whether or not to specialise any given function.

Up until this point, the only possible destructive updates are those performed by built-in functions, such as array_replace and array_add. This is a significant advantage, but we also want to perform destructive update to fields of user-defined types. The final contribution of this chapter is to define the mechanism for converting constructor calls into destructive update operations. There is a complication when dealing with structural induction over recursive types: due to the way our aliasing analysis type folds to make the
internal sharing graphs finite, we are unnecessarily prevented from updating an object while there is a live pointer to its descendant. We introduce a new analysis and internal structure for tracking the ancestor-descendant relationship between variables, and use it to discover when a variable’s top-level memory cell is unique. We use this information along with a heuristic to optimise constructor calls (as a special case), selecting a free variable and converting the call into a reuse of that variable.

Our Mars compiler provides a basic implementation of some of the algorithms presented in this chapter: per-argument interestingness detection with dynamic call mode selection. The remaining features described in this chapter are to be the subject of future implementation work.

This draws together everything discussed in this thesis into a practical optimisation, which, as originally intended, is able to automatically analyse a Mars program, and silently convert both array clone-and-update operations and constructor calls into efficient destructive updates that do not allocate new memory or copy data structures. In the next chapter, we recap the design of Mars and the overall architecture of the destructive update optimisation.
Chapter 11

Conclusion

Over the course of this thesis, we have detailed the exploration of a simple programming language concept: a language designed to be programmed imperatively, despite having a strict declarative semantics. Because such a language blurs the line between imperative and declarative programming, it is helpful to have a clear definition of what declarative programming is about without relying on functional concepts such as expressions, and this was tackled in the first part of this thesis. The obvious consequence of trying to use imperative concepts such as array updates within a purely declarative framework is that data structures will require inefficient copying upon modification. The latter part of this thesis is concerned with a static analysis that reclaims the efficiency of imperative data structure updates, without sacrificing the pure semantics.

11.1 A pure imperative programming language

We began our investigation into developing a hybrid language by looking at the benefits of both imperative and declarative programming. The benefits of imperative programming are disputed; after all, many programmers are content to use purely functional languages such as Haskell. However, we argue that many algorithms are better expressed as a sequence of steps. In addition, the fact that the vast majority of popular programming languages today are imperative means that a language programmable in an imperative style will be more familiar and comfortable to most of the world’s programmers — a property not to be overlooked when designing a language for use in the real world.

The benefits of purely declarative programming are more clear-cut, but also controversial. Restricting the possibility of side-effects means fewer surprises, with many authors recommending that objects be immutable wherever possible [8, 64]. A language which fully prevents procedures from causing side-effects is the logical conclusion of this
principle. In particular, a procedure that does not cause side-effects is bound by its interface — it cannot depend upon any data other than that given to it by its caller, nor can it influence the state of computation other than by the value it returns. However, there are down-sides to purely declarative programming: programs may be more complex due to the need to be explicit about all interactions, and programs may execute slower due to the prohibition against destructively updating objects. We did not address the former in this thesis (indeed, we see the explicit nature of declarative programming as a benefit), but we did address the performance problem later on.

The main contribution of Chapter 3 was to define the concept of interface integrity, which roughly corresponds to purely declarative programming. A language exhibits interface integrity if all of the interactions performed by a procedure written in the language (the procedure’s effective interface) are covered by the procedure’s explicit declaration and call syntax (its apparent interface). This definition allows us to judge whether a language gives us the benefits of declarative programming, without over-constraining the language designer, compared to other definitions which may require that program code is comprised only of expressions which have no effects at all. Our definition allows for a particular sort of imperative programming language — one that allows local effects (such as updates to local variables), but not global effects (such as object mutation or interaction with the operating system). It also allows for conventional declarative languages such as Haskell, Mercury and pure variants of Lisp.

Armed with this definition, we set out to create an imperative programming language that fulfils our concept of interface integrity. Mars, described in Chapter 4, is such a language, featuring statement sequences, local variable assignment, and while loops, but which does not allow objects to be mutated. For example, the array update functions produce a new array which is a copy of an existing array with the required modification. This Mostly simplifies imperative programming; for example, the user does not have to worry about the difference between “value types” and “reference types,” simply treating all values as values. Altering the value held by a variable requires that it be explicitly re-assigned, regardless of whether it is a number or an array.

Mars also features a number of advanced functional programming concepts, such as first-class functions (closures), partial application, algebraic data types, and a polymorphic Hindley-Milner type system. While these features were not strictly necessary for demonstrating our basic concept, they helped demonstrate the pleasant meshing of imperative and functional styles. They also added significant challenges for our static analysis; therefore, it was important to include them, to show that our analysis can handle non-trivial functional language features.
11.2 Automatic destructive update

For the second part of this thesis, we created a static compiler optimisation that automatically converts copy-and-update operations (such as the array replacement operator) into efficient destructive update operations, where it can be proven not to alter the semantics. This means that Mars remains a purely declarative programming language, but structure field and array update operations will have the performance of an imperative language, when the object in question is not aliased.

Specifically, the optimisation first determines whether or not each variable, at each program point, is unique (not aliased). This is conservative, allowing false negatives (a variable may be marked as aliased when in fact it is unique) but not false positives (a variable is never marked as unique if it is, in fact, aliased). Then, if the variable needs to be updated at a program point where it is known to be unique, the update operation is modified so that it is destructive, instead of making a copy. This can be thought of as a form of compile-time garbage collection — without this optimisation, a copy would be made and the old version of the object would eventually be collected by the garbage collector. By reusing the memory allocated to the old object, our compile-time optimisation obviates the need for runtime garbage collection in certain cases.

This technique is the subject of a large body of existing research. However, most past work has used relatively primitive data structures to account for aliasing, losing a lot of information — in particular, when variables become aliased and then unique again. Some past work does use a similar technique to ours, but none of the more precise analyses handle higher-order functions; they typically lose all information about any variable passed to an unknown function. Our analysis keeps all of the information about calls to unknown functions, so it is as precise for higher-order functions as for first-order.

The basic aliasing analysis is context-insensitive — before analysing a function, all of its callees are analysed, and recursive functions are analysed until a fixed point is found. The result is a “uniqueinfo,” a description of the function’s aliasing behaviour. This represents all of the inter-function analysis required, which means that each module or library may be analysed just once, and does not need to be re-analysed unless it or something it depends upon changes. Inter-module analysis is fully supported.

The aliasing analysis tracks which variables may be aliased to which other variables at any program point. Once a variable goes out of scope, any variables it was aliased to may become unique, allowing for previously aliased variables to be destructively updated. The analysis also tracks aliasing separately for each structure field (considering the fields of a recursive type with the same name to be a single entity). The details of the analysis are given in Chapter 8.
We then expanded upon the basic analysis to support higher-order functions. The technique used was to consider the results of calling an unknown function to be an algebraic expression, representing the callee and its inputs. The algebraic expression is substituted when the concrete function is known. The presence of algebraic expressions must be accounted for in all of the sharing graph operations. The details of these augmented operations are given in Chapter 9.

Given a sharing graph, it is a straightforward matter to decide whether to perform a destructive update. When updating an object that was created in the current procedure, this is the case, but when updating a parameter of the current procedure, we need to know whether the caller has supplied a unique object to that parameter. This cannot be known when compiling the procedure, and so in Chapter 10, we explore two techniques for determining this information. The first is to do a small amount of work at runtime, by keeping a uniqueness bit for each abstract object passed to a function. The second is multiple specialisation, in which we compile destructive and non-destructive versions of a procedure, and decide (at compile time) which version to call at each call site. In this chapter, we also described a mechanism for heuristically choosing a dead variable to reuse when a constructor is called, saving a memory allocation.

### 11.3 Benchmarks

To demonstrate that the Mars compiler really works (and is not just a proof of concept), several benchmarks are presented. The purpose was not to compare the automatic destructive update with the unoptimised version; the program transformation reducing \( O(n) \) operations to \( O(1) \) should speak for itself. Rather, this section is designed to compare the raw performance of Mars with other compilers in both the destructive and non-destructive modes. These benchmarks were run on Linux (Ubuntu 14.04) 64-bit with an Intel Core i5-4570 CPU (3.20GHz). All benchmarks are single-threaded.

We compare the performance of Mars programs with equivalent programs written in C and Python, chosen for being extremely popular and well established languages at the low and high level, respectively, of the language spectrum.

- Mars benchmarks were compiled with marsc 1.0, the Mars compiler developed in conjunction with this thesis, using the LLVM backend. The output was then optimised and compiled with LLVM 3.4 using `-std-compile-opts -O3`.

- C benchmarks were compiled with both the GCC 4.8.2 and Clang 3.4 compilers. Both were run with `-O3` for optimisation. Note that Clang outputs LLVM bitcode, like Mars, whereas GCC has its own backend.
11.3. BENCHMARKS

Python benchmarks were run with the CPython 3.4 interpreter, using -O for optimisation.

The first benchmark is a Mandelbrot set generator, written in C by Buchholz for the Computer Language Benchmarks Game [12]. The version by Buchholz is not the fastest in the competition by any means, but it was chosen because it is a simple, straightforward single-threaded version.\(^1\) The C version was ported to Python and Mars by us, keeping the same basic program structure and algorithms, but using the idioms of each language.

The results of running the benchmark for various output sizes are shown in Figure 11.1. The Python figures are not shown, because they are far too big to fit on the graph (on this benchmark, Python performs 60–70 times slower than Clang).

Mars is competitive with the C benchmark for this particular example. The program is basically floating point arithmetic and conditionals, with no memory allocation or complex objects, which means that the Mars compiler can output LLVM bitcode that is nearly identical to what a C compiler would output. Mars performs approximately 24% slower than Clang, due to Mars’ built-in putchar function, which does range checking, unlike C’s

\(^1\)Note that the purpose of our benchmarking is not to compete with hand optimisation or parallelism, but to show the performance of the compilers at full optimisation level on simple programs.
putc. (Clang itself is slightly slower than GCC, but it is fairer to compare Mars to Clang, since both use the same LLVM backend for optimisation and compilation.)

The second benchmark tests a hash table insert operation, derived from our earlier Example 4.12. The program inserts a large number of random integers into a very simple hash table (open addressing with linear probing is used to resolve collisions; the table is rehashed to double size when the load factor reaches 0.75).

In the Mars version of the benchmark, the insert operation uses \texttt{array\_replace} to update the array. To ensure a level playing field, we first disabled the automatic destructive update optimisation, and wrote comparable C and Python versions that use a naïve copy-and-update \texttt{array\_replace} function. Thus, in all three versions, each insert makes a copy of the array, requiring $O(n)$ time. The results are shown in Figure 11.2. Python performed 6–9 times slower than Clang on this benchmark.

We then enabled the automatic destructive update in Mars, and rewrote the C and Python versions so that the insert operation destructively modifies the input array. In

---

2The same randomly generated sequence of numbers is used across all tests.
3Again, the purpose is not to make the most efficient hash table implementation, but to have a simple program which works the same way across all languages.
4Note that the Clang results are nearly identical to those of GCC, so are hard to see on the graph.
11.4. FUTURE WORK

all three languages, the insert operation now takes $O(1)$ time. The results are shown in Figure 11.3. Python performed 12–15 times slower than Clang on this benchmark.

In the hash table benchmarks, Mars is 2.5–3 times slower than C (unlike in the simple arithmetic case). This is because of a number of inefficiencies such as bounds checking in built-in array operations and extra indirection in the implementation of Array. The performance of Mars in this benchmark could also improve if the structure update optimisation proposed in Section 10.7 was implemented.

The key outcome for us, and really the point of this whole thesis, is that the Mars code was identical for both Figures 11.2 and 11.3, whereas the C and Python code had to be largely rewritten. The author of the Mars code is able to enjoy side-effect free programming, yet still reap the potentially massive performance benefit of the destructive version of the code.

11.4 Future work

This research comprises a fairly rudimentary programming language (by the standards of modern “real” languages), coupled with an extensive exploration of an automatic de-
CHAPTER 11. CONCLUSION

Structive update optimisation. Therefore, most of the future research we would like to see involves expanding the Mars programming language itself.

Section 4.5 lists a number of features which we would like to incorporate into Mars. Of particular urgency is type classes — it is rather painful trying to define generic data structures without type-specific operations such as comparison and hashing functions. Also, working out how to incorporate other traditional language features such as object-oriented inheritance and optional arguments into our model would be an interesting challenge.

We also discussed (but did not implement) features that would make the purely declarative update semantics feel more like imperative programming. In particular, a syntax for a non-destructive field-update statement (Section 3.3.2) and the concept of out and inout subroutine parameters (Section 3.3.4) would be a good fit for Mars.

There are a number of algorithms presented in this thesis that have not yet been implemented in the compiler: the algebraic sharing graphs in the higher-order aliasing analysis (required to handle the applicative downward higher-order problem), multiple specialisation of call modes, higher-order call modes, and automatic reuse of objects other than arrays. All of these can be added to the existing compiler, given sufficient time.

We feel that the higher-order aliasing analysis is rather precise, and improving it in any significant way will not be easy. Future work on this analysis is likely to involve small incremental changes, chipping away at some of the limitations by identifying cases where it does not detect uniqueness and addressing them specifically.

The work on heuristically performing destructive update, on the other hand, is ripe for further research. Section 10.8 lists a number of areas where we would like to spend more time, in particular, performing rigorous experiments to determine an algorithm for choosing between dynamic call modes and multiple specialisation.

11.5 What we have built

The product of this research is a new programming language and its compiler. Mars is not a particularly expressive language, as it is lacking a number of features outside the scope of this research, but it is a demonstration of a new hybrid paradigm which we hope will gain traction in the future. The highly precise aliasing analysis and opportunistic destructive update optimisation allow Mars programs to be written in an imperative style, with imperative runtime performance, but without the surprises, bugs and poor design choices that often arise in programs where aliased destructive update is permitted.
Bibliography


Author/s:
GIUCA, MATTHEW

Title:
Mars: an imperative/declarative higher-order programming language with automatic destructive update

Date:
2014

Persistent Link:
http://hdl.handle.net/11343/43263

File Description:
Mars: An Imperative/Declarative Higher-Order Programming Language With Automatic Destructive Update