Portland State University

PDXScholar

Dissertations and Theses Dissertations and Theses

3-10-2022

Using Intrinsically-Typed Definitional Interpreters to Verify Compiler Optimizations in a Monadic Intermediate Language

Dani Barrack
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Computer Sciences Commons

Let us know how access to this document benefits you.

Recommended Citation

Barrack, Dani, "Using Intrinsically-Typed Definitional Interpreters to Verify Compiler Optimizations in a Monadic Intermediate Language" (2022). *Dissertations and Theses.* Paper 5923.
https://doi.org/10.15760/etd.7794

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Using Intrinsically-Typed Definitional Interpreters to Verify Compiler Optimizations in a Monadic Intermediate Language

by

Dani Barrack

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Mark P. Jones, Chair
Andrew Tolmach
Katherine Casamento

Portland State University
2022

Abstract

Compiler optimizations are critical to the efficiency of modern functional programs. At the same time, optimizations that unintentionally change the semantics of programs can systematically introduce errors into programs that pass through them. The question of how to best verify that optimizations and other program transformations preserve semantics is an important one, given the potential for error introduction. Dependent types allow us to prove that properties about our programs are correct, as well as to design data types and interpreters in such a way that they are correct-by-construction. In this thesis, we explore the use of dependent types and intrinsically-typed definitional interpreters in progressively larger subsets of Monadic Intermediate Language (MIL) to verify optimizations used in a compiler back end. In particular, we prove non-trivial program optimizations using the Agda proof assistant, and illustrate the benefits and challenges of this style of program verification.

Dedication

To all of those who were never given a chance to be themselves.

Acknowledgments

I would like to extend my deepest gratitude to Professor Mark P. Jones. His functional programming class, our independent study, and my thesis work were essential to developing my understanding of functional programming and type theory. I could not have hoped for a more helpful, understanding, kind, and brilliant person to be my advisor.

I am extremely grateful to Katie Casamento, a dear friend who first introduced me to category theory, advanced functional programming, who finally convinced me to read Godel, Escher, Bach, and whose late-night discussions about computer science, linguistics, and morality I still think of often. Without her help and encouragement with Agda, I would not have been able to complete this thesis.

I also wish to thank Professor Andrew Tolmach for his interest in my work and willingness to serve on my committee.

I would like to extend my sincere thanks to Professor Bart Massey, whose career and life advice was invaluable in guiding me to the path that I am on now and whose classes showed me the powerful intersection of functional programming techniques with software engineering.

I am deeply indebted to Professor Thomas Squier, who first showed me the beauty of research and biochemistry, allowed me to be legitimately creative and impactful in a research setting, helped show me how important it is to be oneself, and helped convince me to go to graduate school in the first place.

A special thanks to Michael Dixon, a mentor and friend who consistently pushes the boundaries of my understanding of computer science theory and computer security, and who gave me the encouragement and flexibility to work on this thesis while

working with him.

I also wish to thank my parents and sibling whose support and encouragement made this possible, as well as my friends who have been there for me.

My heart overflows with appreciation and gratitude for all of the people named above, and too many others to list. It is not an exaggeration to say that Portland State University was the first place I felt truly at home, and the past few years have been the best time of my life so far.

Table of Contents

Abstract i

Dedication ii

Acknowledgments iii

List of Tables viii

List of Figures ix

Chapter 1: Introduction 1
- 1.1 Fixing the Therac-25 1
- 1.2 Software correctness matters 3
- 1.3 Properties that should hold 5
- 1.4 Pipelines and transformations 9
- 1.5 Contributions and overview 11
  - 1.5.1 Contributions 11
  - 1.5.2 Overview 12

Chapter 2: Habit, MIL, and optimizations 16
- 2.1 Habit 16
- 2.2 MIL 17
- 2.3 Optimizations and transformations 20
  - 2.3.1 Monad laws 21
  - 2.3.2 Constant folding 23
  - 2.3.3 Newtype elimination 24
  - 2.3.4 Using algebraic identities 25
  - 2.3.5 Known constructors 26
- 2.4 Pipelines of optimizations 27

Chapter 3: A path to knowing 30
- 3.1 A scattered semantics 30
- 3.2 Wrapping types and structure together 34

3.3 Correct-by-construction programming and intrinsically safe design . . 37
3.4 Adding variables 40

Chapter 4: Curry, Howard, and friends 42
4.1 Types as propositions, programs as proofs 42
4.1.1 Proofs beyond propositional logic 48
4.2 A map forward 56

Chapter 5: A short introduction to Agda 57
5.1 Proofs, monads, and more 63
5.1.1 Writer 65
5.1.2 Monads in monads in monads in... 70

Chapter 6: SimpleLang 77
6.1 Abstract syntax and interpreter 78
6.2 Proofs of programs 81
6.3 A less simple simplifier 85
6.4 Approaches to optimization verification 96

Chapter 7: Featherweight (M)IL 104
7.1 Abstract syntax and interpreter 105
7.2 A change of type 109
7.3 A proof of correspondence 112

Chapter 8: Pure (M)IL 123
8.1 Abstract syntax and interpreter 123
8.2 Another simplification 126
8.3 Mountains of monoids 129

Chapter 9: Writer MIL 146
9.1 Abstract syntax and interpreter 146
9.2 Right monad law 149
9.3 Left monad law 154

Chapter 10: Block MIL 161
10.1 Abstract data types 161
10.2 Turing completeness and the logic loophole 170
10.3 A gas-powered evaluator 173
10.4 Optimization, termination, and proofs 178

Chapter 11: Conclusion 193
11.1 Future works 193

11.2 Discussion 195
11.2.1 Intrinsically-typed definitional interpreters 195
11.2.2 Broader applications of types in industry 196
11.2.3 What was not covered 199
11.3 Doing it right 201

References 205

List of Tables

7.1 Illustration of the $*^d$, $\wedge^d$correspondence 110

List of Figures

1.1 Strong types preventing the Therac-25 malfunction 3
1.2 An idealized stack 6

2.1 An annotated grammar for MIL 18
2.2 A commutative diagram illustrating the relation between optimization and evaluation 28
2.3 A pipeline of optimizations 29

3.1 The grammar of our language 31
3.2 Language typing rules 31
3.3 Big step semantics 31
3.4 Grammar and typing rules for `ExpVar` 41

4.1 Relations between type levels 55

5.1 An illustration of the Anytype using our example list. 63
5.2 Monad laws 64

6.1 Type introduction for Σ types 88
6.2 An intuition for how inspect works 92

7.1 The correspondence between collapseBool, simplifyEnv, and castDataVal 114

8.1 The monoid laws 130

11.1 A possible path forward with integrating intrinsically-typed and untyped interpreters. In subscripts: *ta* indicates a target, *so* indicates source, *it* indicated intrinsically-typed, *ut* indicates untyped. 197

Chapter 1

Introduction

1.1 Fixing the Therac-25

The Therac-25 was a radiation therapy machine in service from 1982-1987 that, due to problems in the software design and tools available at the time, resulted in the deaths of four patients, and left two others with life-altering injuries [Lev95]. The cause of this error was an intersection of bad design decisions, which included removing the hardware interlocks that were present on the previous model, and relying instead on software correctness to prevent catastrophic error. On a programming level, the designers did not sufficiently separate user input from device operation and used a very stateful design that did not prevent errors by design.

The Therac machine consisted of an electron gun that could operate in two different modes: a low-energy mode and a high-energy mode. The low-energy mode was intended to be used for direct electron-beam therapy, where the output of the electron gun was fired directly at the patient using magnets to distribute the beam over a safe area. The high-energy mode was used for X-ray therapy, in which the extremely high power electron-beam was first aimed at a target, which converted it to X-rays, which then passed through a metal flattening filter, a collimator used to

direct the beam, and finally an X-ray ion filter, before reaching the patient. The energy needed to overcome these multiple layers of material and transmit a sufficient dose was approximately 100 times higher than it needed to be in direct electron-beam therapy mode.

These filters and magnets were arranged on a motorized turntable, controlled by a computer. It should be obvious that, given the difference in energy output, it is absolutely essential that the flattening filter and target are in between the electron gun and the patient when in X-ray mode to avoid an over 100 times overdose that would result from a mismatch between table alignment and power settings. As mentioned above, hardware interlocks were not present on the device, so no hardware or physical controls prevented this mismatch from occurring.

If we wanted to fix the Therac-25 today, what tools could we use to solve not just this problem, but problems like it? In essence, the problem with the machine was that there was a traversable execution path that skipped around the checks meant to protect against these configuration mismatches, with fatal consequences. Unlike in the 1980s, we have more advanced tools to work with today. Instead of a custom real-time operating system, a modern iteration could use a unikernel model or simply a small Linux computer to manage console interactions and supply a scheduler, which were nontrivial causes of the race conditions that led to the Therac software being so error-prone. In addition, instead of the assembly language that was used in the Therac software, we have more advanced programming languages that give us the ability to reason more clearly and explicitly about limiting the set of valid paths in our programs.

Using a functional language with strong static types, we can imagine a pipeline that accepts user input and packages it in a “UserInput” type. We can then imagine

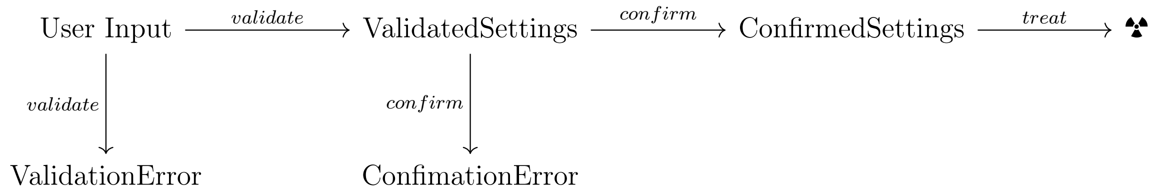

Figure 1.1: Strong types preventing the Therac-25 malfunction

that it passes next through a validation step, where the user input is either determined to be invalid (in which case an error is thrown), or it is determined to be valid and translated into a "ValidatedSettings" type. We can further imagine a process where the validated settings are displayed on a screen, thus eliminating concerns about user input that is not registered overwriting the actual consistent package of settings, and where, upon confirmation from the user that the settings are correct, a signal is given to the turntable to rotate into the proper position and ensure that the beam directing and transforming apparatus is in place. The program could then confirm that the actual settings match the hardware detected settings, and wrap this in an additional "ConfirmedSettings" type. The function that applies the treatment could then, via strong typing, only accept a ConfirmedSettings value. In doing this, we set up a composition of functions that separate input from execution via a series of validation steps, in which skipping a step results in a type error and thus compilation failure. By using richer types, we can rule that error as impossible in any code that compiles with this type system.

1.2 Software correctness matters

Software correctness matters. Computers are integrated into almost every critical system that exists, from nuclear reactors, to power plants, to pacemakers, to vehicle

control systems. Errors in our programs become errors in the physical world, and errors in our lives. Some of these errors are obvious, like overdosing one's patients with radiation, causing airplane crashes [MPŠ+20], or leaking confidential information to the world. Others are much less so, such as systematically excluding certain genes from analysis in biological research [ZEE16], or arbitrarily ruling candidates as undesirable due to specific word choice or order in applicant tracking systems [Web12].

Software is necessarily extremely complicated. Even the 1993 video game DOOM, a relatively simple piece of software compared to modern systems, is made up of over 60,000 lines of source code [Car97]. Conveniently, there is no catastrophic result if one's DOOM game crashes[1], it seems that this is the exception, not the rule of software errors. In addition, even 60,000 lines of code is far beyond one developer's ability to understand inside and out. And without some systematic guarantees, it will be very hard to reason about what any non-trivial program actually does. This, however, implies that there is some standard by which to measure what a program should do against what it actually does. What tools do we have to reason about programs in this way?

Ad-hoc testing is a simple approach in which a person manually runs functions or programs in an interpreter or via a terminal, attempting to judge the correspondence between what is output, and what they think should be output by a correct program [Bla02, p.94]. If one writes these down and generates an automatic way to run all of these tests, this becomes *unit testing* [Ham04, p.13]. The implication here, however, is that there is an idealized model of the program represented in the mind of the developer, and against which the program can be tested by picking individual inputs or sets of inputs manually and seeing if the outputs correspond to what is

[1] Possibly dependent on one's definition of catastrophic result.

expected. But there are many problems with this: the model may not be consistent over time, the method of testing against it passes through the programmer, which is the primary source of error in software development and which is rate limited and labor-intensive; and this mental model is not assured to be internally consistent. Each of these factors results in this modeling strategy being insufficient. One can attempt to write a model system down as a specification, but in effect that is what one is attempting to do by programming. Even when one can write down a formal model that can be tested against, how do we know that the model has the properties we care about, and that it behaves as we expect, and how do we even specify what "we expect"?

1.3 Properties that should hold

Consider a typical implementation of a stack in a C-like language. The stack itself is represented by two `int` values representing the number of members of the stack and the capacity, followed by a pointer to an array on the heap. We are supplied with functions that allow pushing an object onto, and popping the top object off the stack, as well as functions to clone and compare equality between two stacks. It seems reasonable that there are important properties about this stack and these operations that should hold. For example, pushing some object onto the stack and immediately popping it off should leave the stack in its original state. This property can be represented by stating that the following function, `pushPopProperty`, should return `true` for any stack and any value (where the `s1 == s2` compares the two stacks for equality of contents).

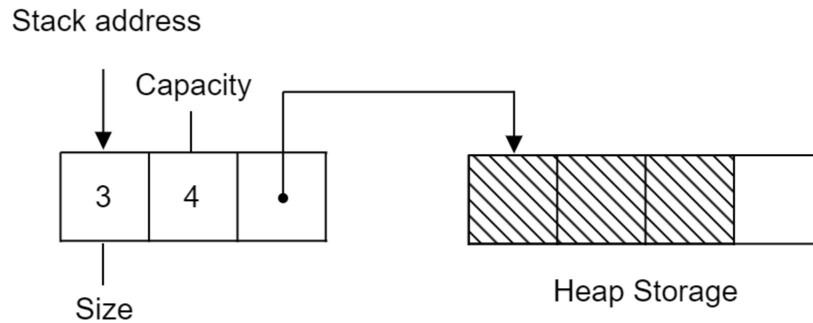

Figure 1.2: An idealized stack

```
bool pushPopProperty<T>(Stack<T> s1, T x){
    Stack<T> s2 = s1.clone();
    push(x,s1);
    T y = pop(s1);
    return (s1 == s2);
}
```

Assuming an implementation for which this holds, consider what would happen if we accessed the memory location of the stack, offset it to access the capacity and the size, and overwrote them with larger values than they originally contained, as shown below.

```
bool pushPopProperty<T>(Stack<T> s1, T x){
    Stack<T> s2 = s1.clone();
    push(x1,s1);
    int* s1_addr = &s1;
    s1_addr[1] = 30000;
    s1_addr[2] = 30000;
```

```
    T y = pop(s);
    return (s1 == s2);
}
```

It seems clear that, in addition to being a security vulnerability (this could allow one to access memory past the end of the stack) [PLO21], this could cause the property mentioned previously to fail, as the pushed value would likely not be at precisely the index we set as the top of the stack. But consider a less deterministic procedure, one that repeatedly writes a random integer into a thousand random memory locations between pushing and popping the stack, as shown below. It should be clear that this property does not always hold in this context, even though this function may return `true` the vast majority of the time.

```
void sideEffect(){
    int* adr;
    for( int i = 0; i <1000; i++) {
        adr  = randAddress();
        *adr = randInt();
    }
}
bool non-DeterministicpushPopProperty<T>(Stack<T> s1, T x){
    Stack<T> s2 = s1.clone();
    push(x,s);
    sideEffect();
    T y = pop(s);
    return (s1 == s2);
```

```
}
```

Although these specific examples seem contrived, analogous things often occur unnoticed. Off by one errors, for example, can write past allocated blocks of memory, causing data corruption [21c]. Multiple references to the same memory location can result in unpredictable behavior, where values can be invisibly changed in unnoticed function calls [ACD+15]. Simply put, this means that, even if one's data structure seemingly conforms perfectly to some specification, aliasing and arbitrary memory access renders the abstraction transparent, and in doing so converts assurances to suggestions.

There have been attempts to limit features of languages, to render their abstractions opaque. For example, Haskell is a pure language [Mar10], and as such, all side effects must be explicit and in the context of a specific IO monad[2] In safe Rust, there is the ability to mark variables as immutable, arbitrary memory access is disallowed [KN19], and the type system enforces a memory ownership mechanism preventing variables from being modified and then accessed in the original context [JJK+17]. The reason our example stack-related programs above let us pierce the veil of the abstraction, so to speak, is that they did not give us strong language guarantees against accessing the memory underlying our data structures or limit the casting of pointers. Another way of thinking about this, which will become important later, is that some programming languages allow us to reason explicitly about concepts like ownership, mutability, and side effects by raising them into the realm of the type system.

[2] Except for certain explicitly unsafe functions in the foreign function interface [Mar10], as well as in the GHC implementation and standard library, such as the dreaded `unsafePerformIO` which allows one to extract a "pure" value from an IO value [GHC01b].

Eventually, regardless of their source language, many programs run on a physical computer[3]. Memory locations are written and re-written. Data is loaded into the processor, mutated, and shuffled back into memory. The computation is inherently stateful, and the assembly language native to that architecture (with some exceptions currently in the research space [XH01]) is generally not typed. How is the gap bridged from these guarantees in the top-level language to a free for all? In the formal study of type systems for lambda calculus we speak of *type erasure* [Pie02, p.110]. That is, an expression in the lambda calculus is type checked to make sure that certain properties of it hold, specifically that it will evaluate to a value of a specific type at runtime without a type error. The types are then erased, and the untyped lambda calculus expression is then evaluated. We know that the untyped expression will evaluate properly because the type checker guaranteed that the original program will do so and because it can be proved that erasing the types will not change the runtime behavior of the expression [Pie02]. In the same way, we have type level structural guarantees in a safe language like Haskell or Rust that still hold when accurately compiled into a language without these safety guarantees, such as x86 assembly.

1.4 Pipelines and transformations

It seems obvious, in a strictly deductive sense, that compiling a program written in a safe language to an unsafe language should maintain the guarantees of the safe language, if the compiler accurately translates the semantics of the source program to the target program. This roughly corresponds to the statement that, if program P_a satisfies a set of run-time properties S, and a program P_b (a program created by

[3]Many others, for example are meant to run exclusively on our minds.

compiling P_a), has the same run-time properties as P_a, then program P_b satisfies the run-time properties S, where P_a represents the safe language program, P_b represents the unsafe language program, and S represents the language safety properties. What is less obvious is what happens when a compiler step materially changes the run time behavior from P_a to language P_b.

For example, consider tail call optimization. Using Haskell as an example, no mutability means no naive loops (excluding the use of the IO, State, or ST monads), and thus all of our code must operate by recursion over immutable data structures. This presents many practical and other efficiency issues. First, unoptimized recursion pushes a new frame on the stack for each recursive function call. This necessarily limits how many calls deep a recursive function can go before one runs out of stack space. Second, even if one does not run out of stack space, one is simply filling the memory with redundant immutable data. Third, these additional calls take longer than a loop takes to compare and then jump to the next iteration. Tail call optimization is a method by which, under very specific circumstances, a recursive function can be transformed to an imperative function where the recursive calls are transformed to a structure corresponding to a loop [LS99].

Something to note here is that these mutable data structures are not representable in Haskell. To optimize in this way, we must translate the source program to a language that is high-level enough that the context that allows such an optimization is still present, but low level enough that we lose some of the guarantees that Haskell gives us about mutability. This is reflected in compiler design, where, typically, some source language passes through many intermediate language steps. This, in and of itself, is not any more of a problem than doing a single compilation step in theory when it comes to correctness. If we have P_a, and a transformation $T_{a \to b}$ such that P_a and P_b

have the same run-time semantics, then it follows by the transitive property that, if we then apply a second semantics preserving function $T_{b \to c}$, then the resulting program P_c has the same run-time semantics as well, and so on.

But what does it mean for two programs to have the same runtime semantics? Does this require that the programs evaluate to exactly the same value given the same input, and that the state of the stack and the heap are exactly the same between the two executions at all times? If so, that would necessarily exclude almost all optimizations, as these (hopefully) change how efficient the program is in some way. In the next chapter, we will make this more concrete by describing a particular compiler, with a particular intermediate language and an associated set of optimizations.

1.5 Contributions and overview

Before going any further, let us prepare to read the rest of the thesis by giving an overview of our theoretical contributions and a summary of the rest of the thesis, chapter-by-chapter.

1.5.1 Contributions

The main contributions of this thesis are as follows. We:

- Designed and implemented a series of six intrinsically-typed language definitions and corresponding intrinsically-typed definitional interpreters of increasingly sophisticated languages, culminating with an intrinsically-typed formulation of a significant subset of Monadic Intermediate Language.
- Formalized a collection of program optimizations, including examples of constant folding, use of algebraic identities, known-constructor conditional elimination,

and tail call introduction, and proved their correctness by developing approximately sixty formal proofs encoded in the proof assistant Agda.

- Explored a new paradigm of compiler and optimization correctness where intrinsically-typed language definitions are used to narrow the domain of the optimizations to only type-correct and well-scoped source programs. This approach contrasts with other established methods, such as those used in CompCert, whose optimizations are defined over all source programs but whose correctness is only proven for valid source programs [Ler09].

1.5.2 Overview

We begin this development in Chapter 2, in which we describe the MIL language, its position and use in the Habit compiler, and example classes of optimizations defined in the MIL paper. We then describe the relation between optimization and evaluation that defines what makes an optimization semantic-preserving.

In Chapter 3, we define a simple language and corresponding typechecker and evaluator in Haskell. We then illustrate a weaknesses in this approach: that it allows ill-typed expressions to be representable. We then update the language definition to be intrinsically-typed, which makes only well-typed expressions representable in the language by embedding the types in the data type definitions of the expressions. We end this chapter by introducing the ability to use variables, and how they break this correct-by-construction language design.

Next, in Chapter 4, we explore the Curry-Howard correspondence and how it draws parallels between types and propositions and programs and proofs. We then go on to define a simple type-level data type and function, and prove properties over it. Finally,

we show a path to simpler proofs over our programs by using dependent types, which allow types to depend on values.

Chapter 5 introduces the Agda proof assistant and shows the advantage of a natively dependently-typed language by re-proving the propositions we had proved earlier in the last chapter. We then introduce the concept of a monad, implement examples of monads, and prove that our implementation of these monads abides by the monad laws with explicit proofs using dependent types.

In Chapter 6, we re-implement the final interpreter from Chapter 3 in Agda, with the power of dependent types allowing us to maintain the intrinsically-typed nature of the interpreter even with variables added. We go on to define two constant-folding optimizations of increasing complexity and prove that they maintain semantics as defined in Chapter 2. We then prove the simpler of the two optimizations correct on an untyped version of the language, implemented in Agda, and draw comparisons between the untyped and intrinsically-typed proofs of that proposition.

Later, in Chapter 7, we implement a minimal subset of MIL consisting only of Tails, MIL's basic unit of computation. We then implement a constant folding optimization as in Chapter 6, as well as a representation transformation where boolean values and the conjunction primitive operation are transformed to integers and the multiplication operation, respectively. We then prove both of these optimizations correct.

In Chapter 8, we update the language defined in Chapter 7 to include code sequences, the construction related to a basic block in other intermediate languages. We then define a constant-folding optimization similar to the one featured in the last chapter. We expand on this by defining an optimization that applies the constant folding optimization to all the tails in a code sequence, and prove it correct. We then introduce the concept of a monoid, a binary operation with a unit element, and

encode this as a record in Agda. We then implement an instance of this record and implement an optimization that allows optimization of any tail that is a monoid, with one of the operands being the unit element. We prove this correct and point out that any proof mapping a tail-optimization across a code sequence would look the same other than the specific optimization function being applied. We then write higher-level optimizations that allow us to map any tail optimization across a code sequence and prove that this map is correct as long as the optimization is correct. Finally, we prove that any two tail optimizations that are proven correct can be composed while maintaining correctness.

Chapter 9 extends the language implemented in the last chapter with an output operation and a monadic evaluator effect: the aggregation of a log, meant to be a standard out analog. We then implement the right monad law optimization and prove it correct using the right monad law proof implemented in Chapter 5. We then finish the chapter with an implementation of substitution, and implement an optimization based on the left-monad law.

In Chapter 10, we extend the language with block calls, the ability to branch, and the ability for tails to return lists of values. The introduction of block calls necessitates an exploration of the totality of Agda, how allowing non-termination can introduce logical errors, and how we can embed a language that allows non-termination into one that does not. We then illustrate the capabilities of our language by writing a realistic program that uses memoization and mutual recursion, and show that executing it results in the correct output and side-effects. We then implement a function that re-writes branches with known conditionals as block calls to the appropriate block, and prove it correct.

Finally, in Chapter 11, we review the techniques used in this thesis and discuss

other type-system relevant advances, as well as the place of strong types in software engineering. We then discuss the limitations of this style of program verification and end on a discussion of the importance of software correctness in a society increasingly integrated with computers.

Chapter 2

Habit, MIL, and optimizations

2.1 Habit

Habit is a high-level functional language with ML-inspired semantics and Haskell-like syntax [21a]. The problems that arise with attempting safe low-level programming are numerous, for example: preventing buffer overflows, null pointer dereferencing [21a], and safely manipulating data stored in bit-fields [Dia07]. The designers of Habit aim to solve some of the problems with safe low-level programming by using an algebraic data type approach to specifying value and memory layouts on a bit level.

The current Habit compiler is split into two sections: a front end parser and type-checker called Alb[21b], and a back end called mil-tools[21d]. The full compiler involves compilation through multiple intermediate languages before finally being compiled to an executable binary. Specifically, the front end translates from Habit to LambdaCase (LC), an intermediate language corresponding to the lambda calculus extended with case constructs; the back end translates from LC to Monadic Intermediate Language (MIL) and then from MIL to LLVM. As a final step, the generated LLVM can be translated into assembly, and then to executable object code [JBC18].

Although functional programming has a reputation for inefficiency, modern func-

tional programming languages tend to run only slightly slower than their imperative counterparts [Pau96, p.9]. We can narrow this gap and significantly increase the ability to use functional programming for computationally intensive tasks by using compiler optimizations on functional programs. The purpose of the MIL step in the compiler is to exist as an optimization platform so that otherwise inefficient Habit code can be rewritten into more efficient but equivalent code.

2.2 MIL

The grammar of the MIL language, shown in Figure 2.1 is relatively simple. A MIL program is a list of definitions. The most important definitions are those that define a block, a closure, a top level definition, or a data type definition. Each code block consists of a code sequence, which is a sequence of monadic binds of tails, and assertions that a variable is a specific constructor, and which terminate either with a conditional statement, a case construct, or a tail call. The tail can consist of a return statement, a block call in which a code block is executed with some given parameters, a primitive operation such as the addition of two words, a data value allocation, selecting a component out of a constructed value, allocating a closure, or entering a closure.

Although MIL syntax reads a little like Haskell, it is better thought of as a higher level assembly code. A revealing design choice in this language is that case constructs and if statements do not map to additional code sequences in the same block, but rather terminate the code sequence with a block call. These code sequences are effectively a series of computations in a row, where changes in control flow are either calls to other functions or blocks in tails, or branches at the end of a code sequence.

```
-- program
prog ::= def1 ... defn

-- definition
def ::= b[v1 ,... ,vn] = c                    -- block definition
      | k{v1 ,... ,vn} [u1 ,... ,um] = c      -- closure definition
      | [v1 ,... ,vn] <- t                    -- top-level definition
      | ...

-- code sequence
c   ::= [v1..vn] <- t; c                      -- monadic bind
      | assert a C ; c                        -- data assertion
      | t                                     -- tail call
      | if v then bc1 else bc2                -- conditional
      | case v of alts                        -- case construct

alts ::= {alt1 ; ... ; altn}                  -- alternatives

alt ::= C -> bc    -- match against constructor C
      | _ -> bc    -- default branch

bc ::= b[a1 ,.. ,an]         --block call with list of parameters

-- tail
t ::= return [a1... ,an]              -- monadic return
    | b[a1 ,... , an]                 -- block call
    | p((a1 ,... ,a2))                -- primitive operation call
    | C(a1 ,... ,a2)                  -- allocate data value
    | C i a                           -- select component
    | k{a1 ,... , an}                 -- allocate closure
    | f @ [a1 ,... ,an]               -- enter closure

-- the atom datatype
a    ::= i -- a constant
       | v -- a variable name
```

Figure 2.1: An annotated grammar for MIL

This reveals the design of blocks to be closer to that of basic blocks than function definitions in a high level functional language. Like many other intermediate languages, such as LLVM [22], rather than code blocks or features just being a list of operations in a row, they have additional type information — as MIL is statically typed — that gives greater structure and safety to the code. In addition, every block and function call can return multiple arguments, much like Core-Erlang [CGJ+04], the intermediate language in the Erlang compiler.

To get a taste of what MIL programs actually look like, let us consider a simple program in the MIL paper [JBC18]: one that defines `List`, and implements a `length` function that gets the length of the list supplied as a parameter. Defining the list data type looks much like how we could define a list in Haskell.

```
data List a = Nil | Cons a (List a)
```

For the `length` function itself, where in Haskell we could pattern match on the list constructor and then evaluate to some expression, in MIL case constructs terminate the code sequence. This means that in the `loop` block, the `Nil` and `Cons` cases must be handled in separate blocks, `done` and `step` respectively. `step` does the work of getting the tail of the list, incrementing the length counter, and calling `loop` on the rest of the list, while `done` acts as an identity. We can glean from this the basic feeling and syntax of MIL: each pattern match or branch that would be handled as nested case constructs or recursive expressions in a higher-level functional language is broken out into separate blocks or flattened to tails, respectively.

```
length :: forall (a::type) [List a] >>= [Word]
```

```
length[list] = loop[0,list]

loop :: forall (a::type) [Word, List a] >>= [Word]
loop[n,list] =
  case list of
    Nil  -> done[n]
    Cons -> step[n,list]

done :: forall (a::type) [a] >>= [a]
done[n] = n

step :: forall (a::type) [Word, List a] >>= [Word]
step[n,list] =
  assert list Cons
  tail <- Cons 1 list
  m    <- add((n,1))
  loop[m,tail]
```

2.3 Optimizations and transformations

As much of this thesis revolves around optimizations described in the MIL paper [JBC18], it makes sense that first we should know what those optimizations are! The rest of this chapter will focus on the optimizations that are mentioned in the following chapters, so we can worry about correctness then, and the essence of the optimizations now.

2.3.1 Monad laws

MIL code sequences correspond to programs executed in a monad, and hence can be rewritten for the purposes of optimization using the standard monad laws. For example, in the case that the result of a tail evaluation is bound to some variable which is returned on the next line, the right monad law specifies that $(x \leftarrow t \,;\, return\; x\,) = t$, and so the bind followed by the return can be rewritten as just the tail. For example, consider the following code sequence, where a `Fahrenheit` value is converted to a `Celsius` value. First, we define the types of `Fahrenheit` and `Celsius` with a `newtype` declaration, meaning that there is a single constructor wrapping some underlying type, in this case a `Word`.

```
newtype Fahrenheit = Fahrenheit Word
newtype Celsius    = Celsius Word
```

Once we have our data types defined, we can define a function `toC` to do our temperature conversion. We can see that this block definition extracts the underlying `Word` from the `Fahrenheit` constructor, subtracts it by 32, multiplies it by 5/9, and wraps the resulting `Word` in a `Celsius` constructor. In the original definition of this, the `Celsius` value is bound to a variable, c. This creates the bind-then-return pattern described above, so the code is then optimized to end with the `Celsius(cdeg)` tail, rather than a `return`.

```
toC :: [Fahrenheit] >>= [Celsius]
toC [f] =
  fdeg  <- Fahrenheit 0 f
  scon  <- return 32
  m     <- sub((deg,scon))
  dcon  <- div((5,9))
  cdeg  <- mul((m,dcon))
  c     <- Celsius(cdeg)
  return c
```

```
toC :: [Fahrenheit] >>= [Celsius]
toC [f] =
  fdeg  <- Fahrenheit 0 f
  scon  <- return 32
  m     <- sub((deg,scon))
  dcon  <- div((5,9))
  cdeg  <- mul((m,dcon))
  Celsius(cdeg)
```

The left monad law deals with the case that some variable is returned in a tail. One can substitute the returned value for the bound variable in the rest of the code sequence. This can be written as $x \leftarrow \textit{return}\ a;\ c = [a/x]c$. Interestingly, this is a more general form of a traditional optimization called copy propagation [Sco16], as instead of making a copy of a in x, the code sequence c can be re-written to use a directly instead of x. This is a very powerful optimization, allowing known values to be propagated forward through the code sequence. In the `toC` function described above, we had defined the adjustment subtracted from the `Fahrenheit` value as a constant `scon`, to avoid having 32 as a magic value. Using the left monad law, we can delete the bind to `scon`, and substitute `scon` for 32 in `sub((deg,scon))`.

```
toC :: [Fahrenheit] >>= [Celsius]
toC [f] =
  fdeg  <- Fahrenheit 0 f
  scon  <- return 32
  m     <- sub((deg,scon))
  dcon  <- div((5,9))
  cdeg  <- mul((m,dcon))
  Celsius(cdeg)
```

```
toC :: [Fahrenheit] >>= [Celsius]
toC [f] =
  fdeg  <- Fahrenheit 0 f
  m     <- sub((deg,32))
  dcon  <- div((5,9))
  cdeg  <- mul((m,dcon))
  Celsius(cdeg)
```

2.3.2 Constant folding

mil-tools also supports constant folding, that is pre-computing structures with known values at compile time. For example, `add((5,3))` can be rewritten as `return 8`, since $3+5=8$. In the last optimization description, we optimized away the declaration of `scon`. Using this constant folding, we can reduce our `dcon` to the same form. We can pre-compute the value of `div((5,9))` to `0.55556`, which can then be substituted for `dcon` in a left monad law application.

```
toC :: [Fahrenheit] >>= [Celsius]
toC [f] =
  fdeg  <- Fahrenheit 0 f
  m     <- sub((deg,32))
  dcon  <- div((5,9))
  cdeg  <- mul((m,dcon))
  Celsius(cdeg)
```

```
toC :: [Fahrenheit] >>= [Celsius]
toC [f] =
  fdeg  <- Fahrenheit 0 f
  m     <- sub((deg,32))
  dcon  <- return 0.55556
  cdeg  <- mul((m,dcon))
  Celsius(cdeg)
```

```
toC :: [Fahrenheit] >>= [Celsius]
toC [f] =
  fdeg  <- Fahrenheit 0 f
  m     <- sub((deg,32))
  cdeg  <- mul((m,0.55556))
  Celsius(cdeg)
```

2.3.3 Newtype elimination

Looking at our most recent iteration of the `toC` function, it might be useful to point out an additional point of inefficiency. We had added a level of type safety by using `Celsius` and `Fahrenheit` data types, preventing functions that expect one from being supplied the other, even though they have the same underlying data type. However, it is important to notice that half of the lines in our most recent iteration of `toC` are extracting a value from a `Fahrenheit` value, and constructing a `Celsius` value. This overhead, particularly costly in the constructor case, is unnecessary. We can re-write our `Fahrenheit` and `Celsius` to their underlying data type and eliminate the constructor and component selection lines.

```
toC :: [Fahrenheit] >>= [Celsius]
toC [f] =
  fdeg  <- Fahrenheit 0 f
  m     <- sub((deg,32))
  cdeg  <- mul((m,0.55556))
  Celsius(cdeg)
```

```
toC :: [Word] >>= [Word]
toC [f] =
  m     <- sub((deg,32))
  cdeg  <- mul((m,0.55556))
  return cdeg
```

2.3.4 Using algebraic identities

As shown above, there are sometimes situations where a computation is entirely known at compile time, but we are not always so lucky. There are certain algebraic identities, however, that can be resolved at compile time and used to simplify tails. These allow certain operations partially known at compile time to be resolved to a known value. For example, if one of the inputs to a Boolean conjunction function is known to be false at compile time, the entire expression is known to be false, as there is a known identity $\forall x : \mathbb{B}, (x \land \text{false} \equiv \text{false})$. Thus a tail `and((false, x))` can be rewritten to `return false`, eliminating the need to call the `and` primitive operation and allowing for further compile-time optimizations. There are many such identities, for example multiplying by zero or dividing by one. To illustrate, consider a common algebra problem in introductory mathematics: squaring a binomial. Expanding $(a + b)^2$ into a sum of three expressions, $1 * a^2 + 2ab + 1 * b^2$ is a is a specific case of a general pattern, which can be derived by copying the coefficients of each expression in order from Pascal's triangle [Kli72, p.272]. In our function `binomialSquare` below, we implement this function without omitting the constant multiple of 1 on the squared terms, as one would if they directly copied the coefficients off of Pascal's triangle. The multiplications by one can be rewritten to `return`s, using the identity $\forall x : \mathbb{Z}, (x * 1) = x$.

```
binomialSquare :: [Word,Word] >>= [Word]
binomialSquare [x,y] =
  x2        <- mul((x,x))
  y2        <- mul((y,y))
  xTimesY   <- mul((x,y))
```

```
  2xTimesY  <- mul((xTimesY,2))
  x2c        <- mul((x2,1))
  y2c        <- mul((y2,1))
  ones      <- add((x2c, y2c))
  add((2xTimesY,ones))

binomialSquare :: [Word,Word] >>= [Word]
binomialSquare [x,y] =
  x2        <- mul((x,x))
  y2        <- mul((y,y))
  xTimesY   <- mul((x,y))
  2xTimesY  <- mul((xTimesY,2))
  x2c       <- return x2
  y2c       <- return y2
  ones      <- add((x2c, y2c))
  add((2xTimesY,ones))
```

2.3.5 Known constructors

The mil-tools optimizer has a way to optimize away known constructors in `if` and `case` statements: if the constructor of a value or the value of a conditional is known at compile time, the `case` and `if` statements involving it can be simply rewritten as block calls to the block pointed to by the known constructor. To illustrate this, let us consider a modified version of the `length` function described earlier in this chapter. In our new version, the `done` block ends with an unnecessary case construct. We do not need to pattern match on `list`, because we already asserted that `list` is a `Nil`.

This means that `list` has a known constructor at compile time, and therefore the `case` construct can be turned into the block call pointed to by the `Nil` constructor.

```
length[list] = loop[0,list]

loop[n,list] = case list of
  Nil  -> done[n, list]
  Cons -> step[n,list]

done[n, list] =
  assert list Nil
  case list of
     Cons -> loop[n,list]
     Nil  -> id[n]

id[n] = n
```

```
length[list] = loop[0,list]

loop[n,list] = case list of
  Nil  -> done[n, list]
  Cons -> step[n,list]

done[n, list] =
  assert list Nil
  id[n]

id[n] = n
```

2.4 Pipelines of optimizations

In the previous section, we dealt with a relatively small set of optimizations. For some of them, understanding how they maintain program semantics may seem obvious, for example in the case of constant folding: one is simply doing the same computation that would normally be done at run time at compile time. Even the use of identities to optimize code has a similarly compelling argument: although the tails are not literally the same expression, they can be shown to always evaluate to the same value. Arguments this like could be made for some of the listed optimizations, but note that this argument does not strictly hold for other optimizations described

in the MIL paper.

Consider what happens when a single constructor type is eliminated. Most of the time, this should not change what is returned from a computation. But, what if the original program returned a constructed value that, in the optimized program, is now returned as a primitive value that the single constructor wrapped? This definitionally changes the output of the program. It is also clear, at least on the face of it, that this does not change the behavior of the program in a way that invalidates the optimization. If we ask ourselves why this is, a more holistic view of program equivalence comes into view. It is too restrictive to say that the behavior of an optimized program must exactly match the unoptimized program, but rather that there is a known correspondence between the two. This can be written as a commutative diagram as shown in Figure 2.2, showing that the relationship between optimization, some matching function f on the evaluation of the un-optimized function, and evaluation, should commute[1]

$$\begin{array}{ccc} Program_a & \xrightarrow{optimize} & Program_b \\ \downarrow{\scriptstyle Eval} & & \downarrow{\scriptstyle Eval} \\ Value_a & \xrightarrow{f} & Value_b \end{array}$$

Figure 2.2: A commutative diagram illustrating the relation between optimization and evaluation

Although each of the optimizations discussed in this chapter seems simple, the power of these operations comes from their working in concert together. Identities can be used to aggregate known values, which are then computed at compile time.

[1] In this diagram, $Value$s are monadic values which encapsulate the side-effects of evaluation.

These known returns can be substituted through the rest of the code block, which may admit further optimizations. The strategy of optimizing tails to `return`s and then propagating these forward with the left monad law is a powerful one. This stacking of optimizations can result in vastly more efficient programs that differ in length and content from the originals. In fact, the MIL paper mentions an example where a program that initially consisted of 910 lines of MIL code was optimized down to 140 through repeated optimization passes. This stacking of optimizations can be represented as an extension of the commutative diagram featured in Fig. 2.2, and is shown in Fig. 2.3

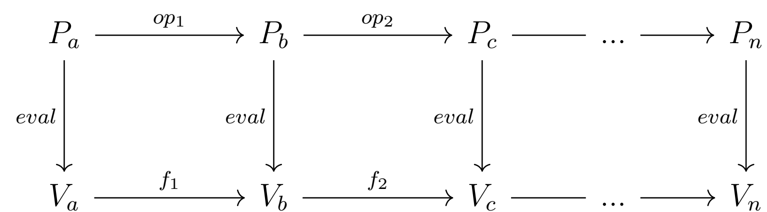

Figure 2.3: A pipeline of optimizations

Simple optimizations working alone seem intuitively correct. But many things that may seem intuitively correct are wrong. In addition, combinations of optimizations can radically transform programs in ways that make the correspondence between source and optimized programs much less obvious. How can we gain confidence that these potentially radical changes will not meaningfully change the semantics of our programs, with a confidence that would result in us comfortably running these transformed programs on critical systems? In the next chapter, we will discuss what it means to really know what our programs are doing, and, in turn, work towards knowing that these optimizations maintain the program semantics we expect.

Chapter 3

A path to knowing

In our previous chapter, we emphasized that there are nontrivial optimizations that we must know –with absolute certainty– will not change the semantics of the programs they are applied to if we are to have confidence in the integrity of our optimized programs. To answer the question of how we really know that semantics are maintained through these transformations, let us begin by constructing an interpreter for a language, and assuring certain things with the type system. That way we can begin to work towards assurances in our program evaluation.

3.1 A scattered semantics

Consider a simple language operating over integers and booleans, whose grammar, typing rules, and big step semantics are described in Figures 3.1, 3.2, and 3.3 respectively. We can implement this language in Haskell and attempt to match the formal semantics with an implementation, as shown below. In this paradigm, the context free grammar corresponds to an algebraic data type `Expression` describing expressions; the typing rules correspond to the typechecking function, `typeCheck`; and the evaluation corresponds to the behavior of the evaluator function, `evaluate`.

$\langle Expression \rangle ::=$ Num $\langle Integer \rangle$
| True
| False
| Plus $\langle Expression \rangle$ $\langle Expression \rangle$

$\langle Type \rangle ::=$ Int
| Bool

$\langle Value \rangle ::=$ ValNum $\langle Integer \rangle$
| ValTrue
| ValFalse

Figure 3.1: The grammar of our language

$$\frac{}{\text{True} : \text{Bool}}\ \text{T-True} \qquad \frac{}{\text{False} : \text{Bool}}\ \text{T-False} \qquad \frac{}{\text{Num}\ x : \text{Int}}\ \text{T-Num}$$

$$\frac{e_1 : \text{Int} \qquad e_2 : \text{Int}}{\text{Plus}\ e_1\ e_2 : \text{Int}}\ \text{T-Plus}$$

Figure 3.2: Language typing rules

$$\frac{}{\text{True} \Downarrow \text{ValTrue}}\ \text{E-True} \qquad \frac{}{\text{False} \Downarrow \text{ValFalse}}\ \text{E-False} \qquad \frac{}{\text{Num}\ x \Downarrow \text{ValNum}\ x}\ \text{E-Num}$$

$$\frac{e_1 \Downarrow \text{ValNum}\ x \qquad e_2 \Downarrow \text{ValNum}\ y}{\text{Plus}\ e_1\ e_2\ \Downarrow\ \text{ValNum}\ (x + y)}\ \text{E-Plus}$$

Figure 3.3: Big step semantics

```
data Expression = ExpNum Int
                | ExpTrue
                | ExpFalse
                | ExpPlus Expression Expression

data Ty = TInt | TBool

data Value = ValTrue | ValFalse | ValNum Int
```

```
evaluate :: Expression -> Maybe Value
evaluate (ExpNum i)      = Just (ValNum i)
evaluate  ExpTrue        = Just ValTrue
evaluate  ExpFalse       = Just ValFalse
evaluate (ExpPlus e1 e2) = do
  v1 <- evaluate e1
  v2 <- evaluate e2
  case (v1 , v2) of
    (ValNum a , ValNum b) -> Just (ValNum (a + b))
    (_ , _) -> Nothing
```

However, there is a critical problem with the data types defining the abstract syntax for the expressions. Consider the expression `ExpPlus ExpTrue (ExpNum 3)`. Because `ExpTrue` and `(ExpNum 3)` are both valid expressions, `ExpPlus ExpTrue (ExpNum 3)` is also valid expression. It is also clear that this is not a valid expression from the perspective of having useful run-time semantics. This means that there exist expressions whose representations typecheck in the host language, but which fail to evaluate to a value[1] at runtime, as shown below.

```
*Main> let e = (ExpPlus ExpTrue (ExpNum 3))
*Main> evaluate e
Nothing
*Main>
```

The possibility for an expression to fail to evaluate to a value is expressed in the return type of the `evaluate` function. The `Maybe` wrapping the `Value` type represents that either the evaluation will succeed and evaluate to a `Just` value, or fail and evaluate to a `Nothing`.

[1]Note that this behavior may be useful when writing interpreters, as it allows one to give detailed errors rather than just throwing a type error in the host language.

In addition, the `evaluate` function evaluates an expression to a value, which can either be a `TInt`, or a `TBool`. This results in an additional problem when one attempts to use the result of an evaluation in a further calculation, as there is no assurance as to what kind of value an expression will evaluate to. As such one has to account for the possibility it will evaluate to a value of a type that is not expected by whatever operation the resulting value is fed into. Whether through missing cases of the evaluator, or a catch-all error when such a type mismatch happens, this creates the possibility of failing to evaluate to a value at runtime.

We can attempt to catch these errors by using a typechecker or type inference function to determine if these types of errors will happen at runtime, and ideally describe any type errors present. Such a typechecker can be seen below. By accepting an `Expression` as input, type checking it, and then evaluating the expression if and only if it type checks, we can prevent ill-typed programs from making it to the evaluator and failing at run time.

```
typeCheck :: Expression -> Maybe Ty
typeCheck (ExpNum i)      = Just TInt
typeCheck ExpTrue         = Just TBool
typeCheck ExpFalse        = Just TBool
typeCheck (ExpPlus e1 e2) = do
  t1 <- typeCheck e1
  t2 <- typeCheck e2
  case (t1 , t2) of
    (TInt , TInt) -> Just TInt
    _             -> Nothing
```

The separation between the AST and the type checking rules means that expressions that would fail at runtime are expressible in this language. The typechecker is an active safeguard to prevent these invalid expressions from reaching the evaluator, but only if we ensure that it is used. Recall that similar engineering safeguards were

implemented incorrectly on the Therac-25 as discussed in Chapter 1.

We are simply trusting, or should attempt to prove, that all of the typing rules are encoded in the typechecker, and that the typechecker itself has sufficient coverage to exclude all ill-typed expressions. In much the same way as the checked or inferred types are generally erased at runtime, we trust that the evaluator is properly constructed so that it does not break the typing rules that we expect it to follow at runtime.

For example, if this hypothetical buggy evaluator evaluated `ExpTrue` to `ValNum 1`, this would not be caught by the type of the evaluator, as `ValNum 1` is a valid value. The level of typing used in our data definitions does not allow us to relate the type of the input expression and the output value, which is the power needed to assure that errors like this cannot be present in the evaluator. This intertwining of relations, and the inability to prevent the mentioned errors before runtime raises a question of whether there might be more elegant way to prevent the evaluation of ill-typed expressions.

3.2 Wrapping types and structure together

The previous subsection raises the question of whether it is possible to embed the types in the `Expression` and `Value` types in a way that would avoid much of the distributed responsibility for preventing runtime failures. For example, if we could parameterize the `Expression` data definition with a type, then we could enforce that subexpressions have the proper type. This would eliminate the need for an explicit runtime typechecker, as this effectively lifts type checking into the host language implementation's typechecker.

Conveniently, generalized algebraic data types (GADT), and data kinds [Mag18]

in a language like Haskell, allow us to do just this. Consider an updated definition of the expression type:

```
data Expression a where
  ExpTrue  :: Expression TBool
  ExpFalse :: Expression TBool
  ExpNum   :: Int             -> Expression TInt
  ExpPlus  :: Expression TInt -> Expression TInt -> Expression TInt
```

This change in the data definition has two primary effects; first, it makes the constructor types explicit; and second, it allows us to parameterize the `Expression` type with another type. It even allows us to specify what type each sub-expression must be parameterized by in the constructor, for example specifying that the two subexpressions for `ExpPlus` must be parameterized by `TInt`, and that the expression itself has that same type. For example, consider the previous expression `ExpPlus ExpTrue (ExpNum 3)` which both fails to evaluate and failed to type check in the previous interpreter. We could construct that expression in the GHCi REPL without any errors using the old definition. If we try that with this new definition a Haskell type error is thrown, as the `ExpPlus` constructor requires that both sub-expressions be parameterized by `TInt`, whereas `ExpTrue` is parameterized by `TBool`.

```
*Main> let e = (ExpPlus ExpTrue (ExpNum 3))
<interactive>:1:18: error:
    * Couldn't match type 'TBool with 'TInt
    ...
*Main>
```

This is an example of an *intrinsically-typed* interpreter, where the types of the expressions and values are embedded in the data definitions themselves [BRT+17]. Type embedding in data definitions has made the sorts of invalid expressions described above unrepresentable in the new `Expression` type. In doing this, we no longer need a typechecker to keep out invalid inputs to the evaluator because we have embedded the typing rules in the abstract data types themselves. This has taken care of the correspondence between expressions and types, however the correspondence between expressions and values that the interpreter captures has not been addressed by this alone. If we parameterize the `Value` data definition in the same way, we can then change the interpreter type so that the input expression, `Expression a`, and the returned value type, `Value a` are parameterized by the same type, which captures this correspondence.

```
data Value a where
    ValTrue  :: Value TBool
    ValFalse :: Value TBool
    ValNum   :: Int -> Value TInt

evaluate :: Expression a -> Value a
evaluate ExpTrue          = ValTrue
evaluate ExpFalse         = ValTrue
evaluate (ExpNum i)       = ValNum i
evaluate (ExpPlus e1 e2) =
  case (evaluate e1 , evaluate e2) of
      (ValNum x , ValNum y) -> ValNum (x + y)
```

Whereas before we needed two cases when we evaluated the sub-expressions of

`ExpPlus`, one for the case in which they both evaluate to `ValNum` and the other for the case in which they evaluate to something else and therefore fail to evaluate, now there is only one case. This is possible because the updated constructor for `ExpPlus` specifies that the type of the sub expressions must be `Expression TInt`, and the type of the evaluator specifies that the value returned from evaluating an expression of that type must be a `ValNum`, so we only need to check that one case. Whereas earlier the return value had to be wrapped in a `Maybe` type in case the evaluation failed, we have no such requirement for this evaluation function. Rather than having to do any manual proofs, we have encoded these properties in the type system, so we get them for free.

3.3 Correct-by-construction programming and intrinsically safe design

One might ask why it is so important that we raise type information into data type definitions. After all, while it has so far been easy in our `Expression` example, this is due to the simplicity of the language, and as we are about to see it gets much more difficult as the complexity of the language increases. By doing this, we are making ill-typed expressions, for example `ExpPlus (ExpNum 1) ExpTrue`, unrepresentable in the host language. The previous alternative, has been to allow illegal statements to be constructed, and to have runtime or pre-runtime checks assure that only correct programs get executed.

There are limits to this however. For example, moving away from language design and implementation, consider a function that fails catastrophically when a certain well-typed input is evaluated. The classic example of this is a division function which takes an integer type, but for which a divide by zero exception be thrown if

zero is supplied as the denominator —in spite of the fact that calling it with the zero denominator would typecheck.

A traditional way of dealing with this in functional programming is to use a safe division function [Hut18], which does not return a `Double` or an `Integer`, but rather returns a `Maybe Double` or `Maybe Integer`. That is to say that it builds in the possibility for failure in the type, rather than leaving the possibility that it fails implicit. While this is certainly better than the alternative, it simply kicks the can down the road for the programmer to deal with. It does not prevent someone calling that function with a zero denominator, it simply makes sure that the failure of that calculation must be handled. Unfortunately, there are cases where such recovery is not possible. Imagine a plane calculating real-time flight control information while landing, or a nuclear reactor calculating the rate at which the control rods are inserted, for example. These are not cases where throwing an error and re-attempting are acceptable.

At least on the positive side, a division by zero error is easy to check for. A function that has the possibility of initiating a side effect of critical importance or magnitude that cannot be undone — such as the secure deletion of a file, or errors that fail silently and cause incorrect data to be surreptitiously generated —are not so easy to simply correct. In these cases we can use special types to prevent these sort of errors at compile time. For example in the case of a safe division operator, a nonzero integer type —as used in Habit— for the denominator would prevent the possibility of getting a division by zero error at compile time. In the case of some more complicated schemes, we can use a type system to assure that certain operations are only possible in specific states.

Importantly, these are encoded in the type system, which assures that, if the

program type checks in the host language, then these properties are guaranteed to hold. Like the evaluator earlier in the chapter, contrast this with manual checks that we hope will adequately capture the requirements we need them to. We can think of these as grounding the logic of the program in a higher logic, that of the language we are working in. Are there analogs for this sort of safe-by-design construction?

In the related field of electric circuit design, there is a similar design approach, that of intrinsically safe design [Cro]. The safety of electrical equipment in terms of ability to generate heat and sparks in a combustive environment is obviously of high importance, given that one generally does not want unintentional combustion in such circumstances. One way to mitigate this risk is to use engineering controls, that is to assure that there is sufficient space and ventilation around the device that an overheating circuit is unlikely to cause significant damage in the event of a catastrophic failure. Obviously, this depends on the relevant engineering controls being correctly implemented, which, like the type checker above or the recovery mechanisms for bad inputs, may or may not be correctly implemented or sufficiently constraining to guarantee safe operation. The alternative, intrinsically safe design, attempts to minimize this risk by limiting the power running through the device in such a way that insufficient energy will be released to initiate an explosion [Cro]. Rather than amassing a large amount of energy and attempting to use engineering controls to minimize damage that will be done if energy is released, we have a design-level guarantee that the energy necessary to fail catastrophically in this way will not be amassed, further guaranteeing that these sorts of damaging failures cannot occur.

In the somewhat less related field of chemical engineering, there is a similar concept called Inherently Safer Design [Hen11]. While historically it was seen as acceptable to have large amounts and dangerous concentrations of hazardous chemicals

present at chemical manufacturing plants, the engineering controls that were used to prevent releases or hazardous reactions were not always adequate to prevent dangerous releases of toxic chemicals, sometimes with tragic results [Joh05]. Inherently safer design, however, attempts to minimize the hazardous condition, as opposed to using engineering controls to contain it. For example, this might mean using less concentrated reagents, handling them in smaller volumes, or substituting less harmful chemicals or processes for those traditionally used [Hen11]. It is certainly hard to accidentally leak chemicals one does not have.

We could continue going through the various fields that have similar concepts, however this would very quickly turn an already lengthy tangent into a miniature thesis of its own. The point is, there is a paradigm shift that has occurred in various fields over the past few decades that involves the acknowledgment that engineering controls restraining the potential for catastrophic malfunctions are insufficient. Correct-by-construction design may be the computer science recognition of this paradigm shift, and the ability to reflect much of the desired computation in types makes our systems ever safer. Although this is a major positive for stronger type systems, this is not the only advantage they confer to us, as we will soon see.

3.4 Adding variables

Considering that this language is significantly less powerful than a pocket calculator, it makes sense that we would want to add some features while maintaining the containment of type information to the data definitions. Let us consider a new rule, one that allows us to look up variables in an execution context at run time.

If we attempt to encode these properties in our data type, however, we run

$$\langle \mathit{Expression} \rangle ::= \ \ldots$$
$$\mid \quad \text{Var}\ \langle \mathit{String} \rangle$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash \text{Var}\ x\ :\ T}\ \text{T-Var}$$

Figure 3.4: Grammar and typing rules for `ExpVar`

into a problem: we need to know what type the variable maps to in context in order to have the `ExpVar` be well-typed. This means that our expression needs to carry along some additional context in which the types of the variables are stored. It seems reasonable that we could parameterize the expression with an additional variable, one that represented the typing context of the expression. But, in order to look up the variable in that context at a type level we would need to be able to execute functions on our types, and, as such, we need a more powerful type system.

In the standard Haskell type system, once we add the feature of variables, we are again reliant on some type checking mechanism to relate the contexts, expressions, evaluations, and values. In the next chapter, let us focus on the correspondence between logic and types, which will give us the background that we need to understand proofs of programs, and the type-level computation features that are required to deal with type contexts.

Chapter 4

Curry, Howard, and friends[1]

In the last chapter, we introduced a simple language, as well as its interpreter and type checker. We then showed how lifting certain properties into the data type definitions of the language obviates the need for explicit checks of these properties, and how we can lift the logic of our embedded language into the logic of the host language using more advanced type constructions such as GADTs. This, however, brings up an interesting question: does our host language actually have a logic in the mathematical sense? In this chapter, we will explore this question and the relationship between types, values, properties, and proofs.

4.1 Types as propositions, programs as proofs

The Curry-Howard correspondence, or Curry-Howard isomorphism as it is sometimes called [nLa21f], refers to a relation between types, programs, propositions, and proofs. Specifically, it asserts that types correspond to propositions in a certain logic, and that the values inhabiting each type correspond to proofs of the associated proposition in that logic. From this, we can draw parallels between logic and type

[1] These friends are presumably numerous, but are generally considered to include Professor Joachim Lambek [Bro20].

theory [How80].

For example, function types can be read as implications, so if we have a function of type `Int -> Int`, that means that given an `Int` in context, we can derive an `Int` as a result. This is simultaneously boring in that it is not a very interesting proof, as well as interesting to notice that there are many such functions that match this type signature. That is to say, there exist many proofs of that property. In fact, given the nature of this isomorphism, all type signatures can be mapped into the realm of logic, and, as such, our programs are actually littered with logical propositions and proofs of them. With simple types, these proofs are not particularly interesting. However, if we use more expressive types, for example adding polymorphism, we can start encoding the basics of propositional logic in our types.

For example, consider the classical inference rule of modus ponens, that, if p implies q, and p holds, then q holds. We can encode this in a type as `(p -> q) -> p -> q` [Bro20, p.4]. The proof of this can be gleaned from function application. If a function `f :: (p -> q)`, and `x :: p` are supplied to our function, by applying `f` to `x` we receive a value of type `q`, as shown below.

```
modusPonens :: (p -> q) -> p -> q
modusPonens f x = f x
```

Implications are not the only tools in this logic. The unit type, `()`, for example, consists of a single constructor that carries no information, which is isomorphic to the true value in logic. Much like one can always derive true in propositional logic, we can always create a unit value. This is represented by the function `trueFromAnything` shown below. As a function, it takes some value `x` of a polymorphic type `a`, and returns a unit value represented by `()`.

```
data () = ()

trueFromAnything :: a -> ()
trueFromAnything x = ()
```

Conversely, there is the uninhabited type, `Void`. This refers to the type for which there does not exist a value. Thinking again in logical terms, this is isomorphic to the canonical proposition for which no proof can exist: false. We can encode a classic property of logic, the principle of explosion[nLa21d]: given a false proposition, anything can be derived.

In order to show this we must use the lambda case extension [GHC20a], an optional extension which allows us to do anonymous case statements. These consist of a list of semicolon-separated constructors for the input type, with corresponding arrows to the resulting expressions, as in a normal case statement. For example, the `toInt` function below is a lambda-case statement which maps `True` to `1` and `False` to `0`.

```
toInt :: Bool -> Int
toInt = \case { True -> 1; False -> 0 }
```

We can use the empty case alternatives extension [GHC20b] —which allows case statements over types with no constructors —to prove the principle of explosion. This encoding makes explicit that, for each constructor of the `Void` type we must supply a proof that `a` is derivable. But as there are no constructors for `Void`, there are no proofs to write.

```
data Void


principleExplosion :: Void -> a
principleExplosion = \case{}
```

We can encode other logical connectives as well. For example, if we consider conjunction, a proposition that two sub-propositions are true, the pair type seems to be a natural analog of this. If we have some true propositions `p` and `q`, in order to derive the conjunction of them we simply create a pair `(p,q)` that contains both.

```
conjInt :: p -> q -> (p,q)
conjInt x y = (x,y)
```

Using this construction, we can prove propositions that we know hold in logic in our types. For example, consider the statements that, for all propositions P, $P \to P \wedge True$, and $P \wedge True \to P$. We can encode these in Haskell, and show that these types are inhabited, and thus proven. In fact the statement "for all propositions P, $P \to P \wedge True$, and $P \wedge True \to P$" can be encoded in the Haskell type system as well. We can also encode the elimination rules in the same way.

```
pImpPandTrue :: p -> (p, ())
pImpPandTrue x = (x,())


pandTrueImpP :: (p, ()) -> p
pandTrueImpP (x, ()) = x


combination :: (p -> (p, ()), (p, ()) -> p)
combination = ( \a -> (a, ()) , \(a,b) -> a)


conjElim1 :: (p,q) -> p
conjElim1 (x,y) = x
```

```
conjElim2 :: (p,q) -> q
conjElim2 (x,y) = y
```

Going beyond this, we can encode disjunctions using a sum type[2] rather than a product type. Sum types must be inhabited by one of the two parameterizing types, but not both. The correspondence between this and disjunction should be clear; both allow us to represent one out of two possible types of values. We can encode both of the introduction rules and their elimination rule in the type system.

```
data  Either a b  =  Left a | Right b


ptoOr :: p -> Either p q
ptoOr = Left


qtoOr :: q -> Either p q
qtoOr = Right


disjunctionElim :: Either p q -> (p -> r) -> (q -> r) -> r
disjunctionElim (Left p)  f g = f p
disjunctionElim (Right q) f g = g q
```

We can go further and use the representation of conjunction as a pair to encode the if-and-only-if construct, represented as a pair of arrow types. Specifically, this would be a pair of proofs that p implies q, and that q implies p.

[2]This implementation of the `Either` data type is taken from the Haskell Prelude [GHC01a].

```
type p <-> q = ( p -> q , q -> p )


iffIntro :: (p -> q) -> (q -> p) -> (p <-> q)
iffIntro pIq qIp = (pIq , qIp )


iffElim :: p <-> q -> (p -> q , q -> p)
iffElim (pIq, qIp) = (pIq, qIp)
```

The correspondence between the not operator and type-level logical operations is much less obvious. The constructive definition of a negation is that if `Not p`, then, given a `p` one can generate a `Void` value. Phrased more directly, `Not p` means that `p` implies `Void`. We can encode this in the type system, and use it to prove a DeMorgan's law. It should be clear at this point that the type system can be used to encode and prove propositions in propositional logic.

```
type Not a = a -> Void


demorgan :: Not (Either p q) <-> (Not p , Not q)
demorgan =
     (\ notporq -> (notporq . Left, notporq . Right) ,
    \(np , nq) ->
        \x ->  case x of
            (Left p) -> np p
            (Right q) -> nq q)
```

4.1.1 Proofs beyond propositional logic

Even given that we can prove properties of propositional logic in our type system, it may be surprising that we can encode arbitrary computations in Haskell's type system. In fact, GHC's type system is Turing complete, and there exist type-level encodings of the SK calculus [Doc06]. While that is somewhat beyond the scope of this chapter, we can, for example, do computations with type level natural numbers. Consider a type level number system based on the Peano axioms [18], in which the natural numbers are represented by either a zero or the successor of another natural number[3]. With this, we can see that we have to define types with no data members, `Z` and `Suc`, which in turn takes another type as a parameter. We can see that the data type `Nat` has type `*-> *`, where `*` indicates the type of simple types in Haskell, meaning that it is constructing types from input types. This is a way of allowing a value to carry a type-level `Nat`[4]. There exist two constructors for this, a `NatZ`, which can only be parameterized by a `Z`, representing a zero value. The second constructor takes some natural number `Nat n`, and wraps a successor around the `n`, resulting in `Nat (Suc n)`, effectively adding one to the input number. This form of unary numbering allows us to represent numbers with added successors onto zero, for example representing three as `Nat (Suc (Suc (Suc Z)))`.

```
data Z
data Suc :: * -> *


data Nat :: * -> * where
```

[3]This implementation of the natural numbers, natural number addition, and natural number equality is based on those used in a programming challenge[Far17], which involved proofs of type level natural numbers.

[4]This is an example of a singleton [EW12], a strategy of Haskell-based dependent typing.

```
  NatZ :: Nat Z
  NatS :: Nat n -> Nat (Suc n)
```

Now that we have defined type-level natural numbers, let us define addition of type level natural numbers. Type families [Zav21] can be thought of in this case as functions from types to types, with type instances as matching patterns. As our representation of natural numbers are types, we can use type families to do calculations on them and return other type-level natural numbers. We define a type family named `(:+:)`, which is parameterized by two types, `n` and `m`. This is much like how we parameterized our GADT `Expression` data type in the last chapter, except that the inputs here are typed explicitly. We can see the we have two instances — type-level analogues to function definitions with pattern matching— we need to account for, one in which `n` is a `Z`, in which case the result of `Z` plus `m` equals `m`, representing adding zero to another number. In the second such instance, where `Suc n` is added to some `m`. This works by recursively pulling `Suc` constructors off of the `Nat` on the left hand side of the `:+:`, and stacking them onto the result of the recursive `:+:` call.

```
type family (:+:) (n :: *) (m :: *) :: *
type instance Z     :+: m = m
type instance Suc n :+: m = Suc (n :+: m)
```

We can then use the `kind!` command in GHCi, which allows us to see type-level computations in the REPL, to show that this addition works as we expect, in this case that $3 + 2 = 5$.

```
*Main> :kind! Suc (Suc (Suc Z)) :+: (Suc (Suc Z))
Suc (Suc (Suc Z)) :+: (Suc (Suc Z)) :: *
= Suc (Suc (Suc (Suc (Suc Z))))
```

We are not limited to computing in the land of types: we can also extract values from the world of types into the world of values by using type classes, traditionally thought of as vehicles for principled function overloading. For example, we can extract a runtime number from a type-level number by creating a type class `ToValue`, which requires its instances to implement a function `toValue :: Natural`. We then create an instance for `Z` where `toValue Z = 0`, as well as an instance for `ToValue (Suc n)` that adds one to a recursive call on `n`[5]. The `@n` is a visible type application [GHC20c], where we are explicitly supplying the `n` from the instance declaration to the recursive `toValue` call.

```
class ToValue n where
  toValue :: Natural

instance ToValue Z where
  toValue = 0

instance ToValue n => ToValue (Suc n) where
  toValue = 1 + toValue @n
```

As we can clearly see, this allows us to take type-level natural numbers and extract them into corresponding value-level natural numbers. This use of type checking as evaluation allows us to calculate values from types at compile time, or in a REPL as shown below.

```
*Main> toValue @(Suc (Suc (Suc Z)) :+: (Suc (Suc Z)))
5
```

[5]This implementation is based on an article by Alexis King on typeclass metaprogramming [Kin21].

We can also represent relationships, for example natural number equality, as a data type `:~:`.We do this with two axioms: The first axiom is that zero is equal to zero, represented by the `ZeqZ` constructor. Note that the constructor enforces that both the parameters must be `Z`. The second constructor, `CongSuc`, a name choice that will be clear in the next chapter, asserts that, if two natural numbers `n` and `m` are equal, then `Suc n` is equal to `Suc m`.

```
data (:~:) :: * -> * -> * where
    ZeqZ    :: Z :~: Z
    CongSuc :: (n :~: m) -> (Suc n :~: Suc m)
```

Given that, as established earlier in this chapter, we can write proofs in our types, and we have an embedding of the natural numbers and their equality relations in our types, let us prove something about our natural numbers[6]. Specifically, it is known that natural number addition follows the commutative property[Lan66, p.6], that is $\forall a, b : \mathbb{N}, (a + b) \equiv (b + a)$. This is a basic enough property that we take it for granted, but it may be useful to reflect that this is not true for subtraction or exponentiation[7], so this is not a trivial property. Given that we claim our representation of `Nat` corresponds to the natural numbers, this is a property that definitely should hold.

We must begin with some lemmas. The first of these is that of *reflexivity*, that all natural numbers are equal to themselves. This can be represented by the `refl` function.

```
refl :: Nat n -> n :~: n
```

[6]These proofs are based on my solutions to a programming challenge[Far17], which involved proofs of type level natural number addition.

[7]As proof, consider that $2 - 4 = -2 \neq 2 = 4 - 2$, and $3^2 = 9 \neq 16 = 2^3$.

```
refl  NatZ     = ZeqZ
refl (NatS n) = CongSuc (refl n)
```

The type signature can be read as: "For all natural numbers n, n is equal to n". Note that the `Nat` value is used as a carrier of the type-level information to the `:~:`. In the case that `n` is equal to `NatZ`, we need a way to show that `Z :~: Z`. Conveniently we have such a way: our first axiom of equality, `ZeqZ`. In the other case, where the supplied `Nat` is a `NatS n`, we need some way to show that `Suc n :~: Suc n`. Our other axiom of equality could prove this, but it requires a proof that `n :~: n` as input. Calling `refl` on the natural number `n` return such a proof, and so by combining our `CongSuc` axiom and a recursive call to `refl` our function type checks, and thus reflexivity is proven over our implementation of the natural numbers.

Consider for a moment what we have done here: we have a base case when `n` is a `NatZ`, that we can prove without assuming anything. The other case, however, only works because we are able to assume that this property holds for `n`, and from that prove that it holds for `Suc n`. If this sounds like an inductive proof, that's because it is! In much the same way that `()` is analogous to true, `Void` is analogous to false and `->` is analogous to implication, recursion is analogous to induction.

Another proof, that `Z` is the right identity over addition [8] can be proved in much the same way. To prove the base case, all we need to show is that `(Z :+: Z) :~: Z`, which can be generated directly generated by our axiom `ZeqZ`, as `(Z :+: Z)` reduces to `Z`. In our inductive case, we need to show that `((S n) :+: Z) :~: (S n)`. A recursive call of `plusIdentityR n : (n :+: Z) :~: n` , so we can again use `CongSuc` to wrap each side in a `S`, completing our inductive case.

[8] A zero being a right identity over addition means that $\forall n : \mathbb{N}, n + 0 = n$

```
plusIdentityR :: Nat n -> (n :+: Z) :~: n
plusIdentityR  NatZ     = ZeqZ
plusIdentityR (NatS n) = CongSuc (plusIdentityR n)
```

Proving that a `Suc` can be taken from the right side and pulled out the outside of the remaining addition is slightly trickier. In the base case we have to prove that `Z :+: (Suc m) :~: Suc (Z :+: m)`. We can see that `Z :+: (Suc m)` reduces to `Suc m` and `Suc (Z :+: m)` reduces to `Suc m`. The proposition we actually need to prove for the base case is `Suc m :~: Suc m`, which we can prove by using `refl` over m, and then applying `CongSuc`. Using `refl m` we can create a proof that `m :~: m`, and from this we can use our `CongSuc` constructor to construct a proof that `Suc m :~: Suc m`. The inductive case follows simply by induction as before.

```
plusSuc :: Nat n -> Nat m -> (n :+: Suc m) :~: Suc (n :+: m)
plusSuc  NatZ     m = CongSuc (refl m)
plusSuc (NatS n) m = CongSuc (plusSuc n m)
```

We are not limited to induction on numbers, though! We can also use induction on the equality relations themselves, as they are defined recursively. For example, we must use this approach to prove the symmetry of equality: if $a = b$, then $b = a$. The base case requires that we prove `Z :~: Z`, and thus can be directly solved by our axiom `ZeqZ`. The inductive case, where we must prove `Suc b :~: Suc a` given `Suc a :~: Suc b`, can be solved by getting the inductive hypothesis from a recursive call of type `b :~: a`, and then adding a `Suc` to both sides with `CongSuc`.

```
symm :: a :~: b -> b :~: a
symm  ZeqZ         = ZeqZ
symm (CongSuc n) = CongSuc (symm n)
```

We can also prove the transitive property, if $a = b$ and $b = c$, then $a = c$, in basically the same way. The base case is actually solved identically, and the inductive case, which requires that we prove that `Suc a :~: Suc c` given that `Suc a :~: Suc b` and `Suc b :~: Suc c`, follows by straightforward induction. The recursive call `(trans aEqb bEqc)` (recursing on both variables) has type `a :~: c`, which becomes `Suc a :~: Suc c` when `CongSuc` is applied. This property is proven in the following definition:

```
trans :: a :~: b -> b :~: c -> a :~: c
trans  ZeqZ          ZeqZ                 = ZeqZ
trans (CongSuc aEqb) (CongSuc bEqc) = CongSuc (trans aEqb bEqc)
```

With all of these lemmas, we can now begin to prove the property that we had intended to prove initially: the commutativity of natural number addition. Using our `plusIdentityR` lemma, we can prove the base case, that `(n :+: Z) :~: n`. Our inductive case requires CongSuc applied to a recursive call
`CongSuc (+-comm n m) : Suc (n :+: m) :~: Suc (m :+: n)`, and
`right :: (n :+: Suc m) :~: Suc (n :+: m)`, in order to prove the required
`(n :+: Suc m) :~: Suc (m :+: n)`. Using the transitive property to stitch these two proofs together, where `a = (n :+: Suc m)`, `b = Suc (m :+: n)`, and
`c =  Suc (m :+: n)`, `trans right ih` produces such a stitching, completing our proof.

```
plusComm :: Nat n -> Nat m -> (n :+: m) :~: (m :+: n)
plusComm n  NatZ     = plusIdentityR n
plusComm n  (NatS m) =
```

```
let
  ih    = CongSuc (plusComm n m)
  right = plusSuc n m
in
  trans right ih
```

To summarize, we have shown that computations can be done at a type level, that we can prove properties of type-level computations (in fact, the type checker of some dependently-typed languages are based on proof engines [Bra13]), and that we can extract values from types. This implies that we could have verified computation occur on the type level at compilation time and then extract the resulting value at runtime. Although this is an interesting property, the difficulties of using a type-level language to implement an interpreter should be obvious, as one presumably wants to run on data encoded in files and not be required to be translate them into type-level constructs. Dependent types solve this conundrum by allowing us to construct types based on values directly, completing this circuit. The interaction between these concepts is shown in Figure 4.1.

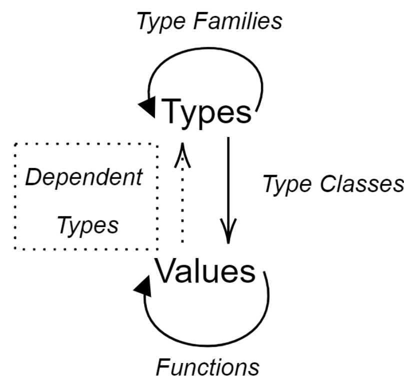

Figure 4.1: Relations between type levels

This is intended to show that, much like functions can operate over values and return values, type families can operate over types and return types. We can extract values from types by using type classes, however, in order to lift values into the type domain, we must use dependent types, which we have not meaningfully introduced yet. Without these dependent types, we would be relegated to writing type-level interpreters if we wanted to prove properties of them using the type system, however if we had these dependent types we could use dependent typing to lift properties of a value-level interpreter into the types, and use them prove that these properties hold.

4.2 A map forward

Of course, in keeping with the theme of this thesis so far, we really need to know what our programs are doing. This includes our compilers and optimization pipelines, especially given that they have the potential to break any programs that pass through them if the optimizations are not known to be correct. Although that is intuitively satisfying, what does it mean to be “known to be correct”? This chapter has focused on the relation between logic and languages, and, in particular, on how one can use types to reason about the logic of their programs. Given the existence of dependent types — where types can depend on values — we can use these dependent types to prove the properties of our value-level interpreters correct. Importantly, this means we can also prove properties about the modifications of programs, and in doing so prove that our optimizations do not change the semantics of programs when evaluated. In the next chapter, we will work with a more expressive dependently-typed language to explore this further.

Chapter 5

A short introduction to Agda

In the last chapter, we worked through examples of type-level programming and proofs about these programs. We then introduced the idea of dependent types, where types can depend on values, and thus we can prove properties about values in our types. In this chapter, let us learn to use dependent types to prove properties about our programs. Although there are ways to use dependent types in Haskell, and although they are powerful and used in industry, there are also practical difficulties that come with their use [CDD+19]. Instead of gradually enabling more and more exotic Haskell extensions to allow us, for example, to execute context lookups at a type level, we will now move to a natively dependently typed language, Agda [Nor07][1]. Here, for example, is a simple Agda program that defines a type ℕ, of natural numbers and a corresponding addition operator:

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
```

[1] In this thesis we are using Agda version 2.6.1.3.

$\mathsf{zero} + n = n$

$(\mathsf{suc}\ m) + n = \mathsf{suc}\ (m + n)$

At first glance Agda's syntax looks quite similar to Haskell's, with the obvious differences being unicode symbols: → and ℕ being used in place of `->` and `Nat` respectively, as well as the single colon being used as the "has type" operator. The natural number definition above looks much like how we defined data types in our GADT style in Chapter 3, and the design mirrors the type-level natural numbers in the last chapter. The addition operator defined above is very similar to how we defined addition on our type-level natural numbers in the last chapter, except done as a typical function rather than a type family. This is both because these are somewhat canonical implementations in this space, and because we want to illustrate the power that Agda gives us, by contrasting an Agda program with a similar program in type-level Haskell.

To show that two things are equal, we need a way to express equality in Agda. Rather than a specific set of equality axioms for a single data type, as we had for our type-level natural numbers, Agda supplies an equality type family, ≡. Simply, there is one way to show that things are equal, and that is to show that they are literally the same. The definition of the equality data type is different enough from standard Haskell that it warrants a detailed explanation. As in our GADT in Haskell, we define a data type between the data and where keywords. The space between these two keywords is taken up by three sections from left to right: the name, the parameters, and the indices. The name of this data type is an operator ≡ surrounded by underscores, which indicate that it is an infix operator. After that, we have variables that parameterize the data type: {a}, {A : Set a}, and (x : A). These will be explained in the next paragraphs.

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  instance refl : x ≡ x
```

In the last chapter we alluded to the fact that each value has a type, and that there is a type of types represented in Haskell by $*$. It seems reasonable to ask what type $*$ has, as we are rapidly entering dangerous ground when talking in terms of self reference in this way[2]. A check into GHCi shows that the kind of $*$ is $*$. Through this we've encountered a way to introduce a logical inconsistency via Girard's paradox [Hur95], as defining a type in terms of itself allows one to prove false. Agda avoids this possibility by creating an infinite hierarchy of types, known as *universes*, where Set a indicates a type of universe level a. For example, ℕ : Set 0, Set 0 : Set 1, and so on [AAC+21e]. This means that functions which are polymorphic over any data type may be polymorphic over universe levels, a feature known as universe polymorphism.

With this in mind, we can go over the parameters of our equality data type, from right to left. This data type is parameterized by some value x of type A — much like the type parameterization of our `(:+:)` data type in the last chapter — where the previous parameter indicates that A is some Set a, where a is inferred to be a universe level. The first two parameters — the ones surrounded by curly braces — only exist to provide a sufficient definition for A. The curly braces indicate that they are implicit arguments, meaning that they can be omitted in the cases when the type checker can figure out what values they should be when creating an instance of this data type. After the :, the A → Set a indicates that the data type must be indexed by a value

[2]If one wants to delve deeper into the dangers of self-reference, a wonderful book that explores self-reference in a mathematical context, among other things, is "Gödel, Escher, Bach: an Eternal Golden Braid" by Douglas Hofstadter.

of type A, which in turn has type Set a. This means that the universe level of the equality data type is the same as that of the type over which equality is being shown.

On the next line is the single constructor refl, which indexes that data type with the input value [3]. This illustrates that, like the `refl` function in the last chapter, in order for two things to be considered equivalent in this type theory, they must be identical.

What may be obvious from that definition is that every e ≡ e type is inhabited. For example, zero ≡ zero is inhabited. We can illustrate this as a declaration with type annotations specifying the equality we are showing, and with the value refl as the proof, as in the last chapter. Note that an underscore of a definition refers to the next implementation with an underscore, and a function definition with an underscore as a name refers to the previous type declaration. An interesting detail with Agda's type checking engine is that types are automatically normalized, so refl successfully type checks as having type $1 + 2 \equiv 3$, because they both normalize to the same value, suc (suc (suc zero)).

```
_ : zero ≡ zero
_ = refl

_ : (suc zero + suc (suc zero)) ≡ suc (suc (suc zero))
_ = refl
```

In our previous chapter, we needed an axiom to derive suc m ≡ suc n given m ≡ n. Normalization is not powerful enough to figure this out automatically, as it depends on an equality between m and n. It turns out that this is a specific case of a more

[3]The instance keyword instructs Agda it to use a different kind of constraint solving algorithm, but does not otherwise change the meaning of the constructor.

general principle, that of *congruence*, which means that if m ≡ n, then f m ≡ f n for all functions f. This property can be captured in Agda using the following definition from the Agda standard library [AAC+21a]:

```
cong : ∀ (f : A → B) {x y} → x ≡ y → f x ≡ f y
cong f refl = refl
```

cong and the trans function encoding the transitive property are conveniently supplied in the standard library, which allows us to re-prove the propositions we proved in the last chapter, but this time in a much more convenient manner and with more general tools.

```
+-identityʳ : ∀ (m : ℕ) → m + zero ≡ m
+-identityʳ zero = refl
+-identityʳ (suc m) = cong suc (+-identityʳ m)

+-suc : ∀ (m n : ℕ) → m + suc n ≡ suc (m + n)
+-suc zero n = refl
+-suc (suc m) n = cong suc (+-suc m n)

+-comm : ∀ (m n : ℕ) → m + n ≡ n + m
+-comm m zero = +-identityʳ m
+-comm m (suc n) = trans (+-suc m n) (cong suc (+-comm m n))
```

We can see above that we can prove the same properties as we did in the last chapter, but in a much clearer manner. The strict division between values and types has been obliterated, and, as such, rather than using type families to create types from types and type classes to extract values from types, we can just write functions

from types to types, types to values, values to types, and values to values. It is easy to default to a pre-type-level programming perspective given this power, where, paradoxically, the ability to transition between types and values causes one to treat this more like value-level programming than type-level programming.

A dimension this brings into programming that is not usually thought of is that of evidence. For example, let us say that we are trying to write a function that takes a list as an argument and then gives us information about whether or not some value is in the list. A straightforward way to do this in a typical programming language would be to recurse through the list, and if a matching value is found, return true, and if it hits the end of the list, return false. In one sense, this does exactly what it should: if the value is in the list, the function returns true, otherwise it returns false. What it has not provided, however, is evidence of anything. We have no indication of what that boolean really represents, as one can create a boolean simply by returning "true" or "false". For the same reason that ≡ does not return a boolean to indicate equality, but rather a Set whose value contains the evidence of that equality, our list membership function must return a Set if we want it to prove anything.

The Any data type can be used to provide such evidence of list membership. An Any type is used to show that there exists some member in a list for which a given proposition holds. Given some predicate P, an Any data type has two constructors: the recursively defined there constructor, where the predicate holds over some member of the tail of the list, and here, where the predicate holds over the current head of the list. This effectively creates a list bundled with the predicate, and an index of a list member for which the predicate holds. This works by providing a path back to the value, as an index, and a proof that the predicate holds over that data member.

For example, consider using ≡ as part of the predicate supplied to the Any type.

If the predicate that we apply to our list is the equality value parameterized with the desired number x, if this type checks, we know that x is in the list. This becomes clearer when illustrated on a concrete list of numbers. Suppose that we have a list of numbers, (2 :: 7 :: 0 :: 2 :: []), and that we want evidence that zero is in the list. The way we would express this proposition with our Any data type is Any (0 ≡) (2 :: 7 :: 0 :: 2 :: []). What proof do we have that this is true? there (there (here refl))) inhabits the above type, by indicating that the member with index 2, 0, is equal to 0. This shows that the supplied predicate (that the number equals zero) matches the third member of the list and supplies a proof to that effect, refl.

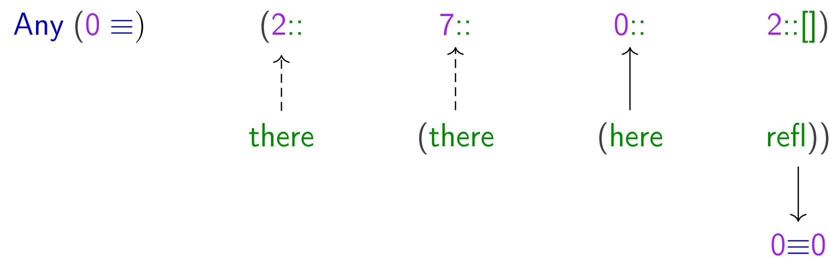

Figure 5.1: An illustration of the Any type using our example list.

5.1 Proofs, monads, and more

The previous section showed that we can prove properties of mathematical functions correct, much as we had done in Chapter 4. These properties of the natural numbers are ones that are simple to understand and often discussed in introductory classes on discrete mathematics, and thus made them useful candidates for us to introduce earlier in this chapter. Extending these tools to properties of less simple constructions than the natural numbers, such as simple monads, will help us along on the path towards proving properties about interpreters.

In order to proceed, let us consider what monads are, and what is required to implement them faithfully. In functional programming, a monad is a Set that has two associated functions which fulfill three laws. The two associated functions are return [4] and bind. return has type a → M a, and can be thought of as a way to inject a value into a monad. bind, often written as ≫= as an infix operator, on the other hand, has type M a → (a → M b) → M b where M is the relevant monad and a and b are types. This allows updates to a value in the context of a monad. The monad laws, as shown in Figure 5.2, constrain how ≫= and return must be implemented in order to be true monadic operations.

$$\frac{\Gamma \vdash m : Monad}{\Gamma \vdash m \gg= return \equiv m} \text{ Right monad law}$$

$$\frac{\Gamma \vdash f : a \rightarrow Monad\, b\, , x : a}{\Gamma \vdash return\, a \gg= f \equiv f a} \text{ Left monad law}$$

$$\frac{}{\Gamma \vdash (m \gg= g) \gg= h \equiv m \gg= (\lambda\, x \rightarrow g\, x \gg= h)} \text{ Associativity}$$

Figure 5.2: Monad laws

Rather than trying to give nontechnical explanations of what a monad is, let us go through an example.

[4] If one is reading Haskell programs, they may see `pure` used in place of `return`. `pure`, from the applicative typeclass, is equivalent to `return` from the monad typeclass. There is currently a proposal to eliminate `return` from the monad typeclass and use `pure` instead, as the monad typeclass now requires a monad to be an applicative [GHC21].

5.1.1 Writer

A classic example of a monad is the Writer monad [Wad92][5], which consists of a pair of some value to be returned and a list of values as a log. We could imagine this being implemented as a debugging log, where important debug information is aggregated as a program executes.

```
Writer : (a : Set) → Set
Writer a = a × (List String)
```

For example, we can imagine an addition function, which would normally have type $\mathbb{Z} \to \mathbb{Z}$. We can modify this to return a pair of both the result of the addition and a description of the function and inputs used. We can see that if we run add' 5 8, the result is a (+ 13)[6] as the value and a singleton list of "added 5 to 8" as the log.

```
add' : ℤ → ℤ → Writer ℤ
add' x y = (x + y) , ("added " ++s show x ++s " to " ++s show y) :: []

_ : add' (+ 5) (+ 8) ≡ ((+ 13) , ("added 5 to 8" :: []))
_ = refl
```

Let us say we want to chain two of these operations together and append the log of the second function to the result of the first. We can accomplish this by defining an implementation of ≫= to compose these functions, that updates the log automatically without additional programmer input. If we then take the same addition expression as described above and bind it to another addition function that adds 7 to it, we can see that the resulting log contains both of the addition steps.

---

[5] Sometimes the Writer monad is referred to as the Output monad.

[6] The + is a constructor for integers in Agda that constructs a positive integer from a supplied natural number.

```
_ : (add' (+ 5) (+ 8))
      >>= (λ x → add' x (+ 7))
      ≡ ((+ 20) , ("added 5 to 8" :: "added 13 to 7" :: []))
_ = refl
```

How does this work? Well, we can see that we are given some (a, vs_1) : Writer a and a function of type a → Writer b. We can use the with construct to pattern match on the Writer resulting from applying f to a, and construct the Writer b by returning the result of evaluating f to a, b, and appending the log vs_1 to vs_2. Note that the with construct allows us to case split on the result of some computation much like a case expression, except it automatically normalizes in ways that case does not, which can be useful for leveraging Agda's unification abilities to ease proving desired properties. The with construct additionally allows us to case over many different expressions simultaneously, a feature that we will use a lot in the coming chapters.

```
_>>=_ : ∀ {a b} → Writer a → (a → Writer b) → Writer b
_>>=_ (a , vs₁) f with f a
...| (b , vs₂) = b , vs₁ ++ vs₂
```

Finally, we can define our return function, which is much simpler. It injects the value a into the Writer, in this case by pairing it with an empty log.

```
return : ∀ {a} → a → Writer a
return = λ a → a , []


_ : return (+ 5) ≡ (+ 5 , [])
_ = refl
```

We have implemented functions that are claimed to be legitimate monadic binds and returns, but we mentioned earlier that, in order for these to be true monadic operations, bind and return must be related to each other through the monad laws. How can we prove that our implementation respects these laws?

Proving the left monad law is easy, as Agda's unification process is advanced enough to normalize the two expressions involved and determine that they are reflexively equal.

```
lML : ∀ {A B C : Set}
  → (a : A)
  → (f : A → Writer B)
  → (return a) >>= f ≡ (f a)
lML a f = refl
```

The right monad law requires that bind and return be related in such a way that binding some value x to a return evaluates to x. We can take the first step by pattern matching on x into (x₁ , vs), and simply showing that x >>= return is reflexively equal to x with the empty list appended to the vs component. We can then use the right identity over lists, that is vs ++ [] ≡ vs, which is a member of Agda's standard library. This works specifically by generating a proof (++-identity^r vs) : vs ++ [] ≡ vs, and using the congruence property over (λ y → x₁ , y), resulting in a proof that has type x₁ ,vs ++ [] ≡ x₁ , vs.

```
rML : ∀ {t} → (x : Writer (DataVal t)) → (x >>= return ) ≡ x
rML x =
  let ( x₁ , vs) = x
  in
```

```
begin
(x >>= return )
≡⟨ refl ⟩
x₁ , vs ++ []
≡⟨ cong (λ y → x₁ , y) (++-identityʳ vs) ⟩
  x ■
```

If we read through the proof of the right monad law, we can see that after the let expression, there is an expression consisting of a begin, a series of expressions separated by ≡⟨ ...⟩, and which ends with a ■. This is an approach to proofs called *equational reasoning*, where in trying to prove a ≡ c for some a and c, we set up a series of transitive property proofs such that we can build a chain of reasoning with intermediate steps, with ≡⟨...⟩ requiring a proof of equality between the brackets. For example, begin a ≡⟨ aEqb ⟩ b ≡⟨ bEqc ⟩ c ■, where aEqb : a ≡ b and bEqc : b ≡ c, would have type a ≡ c. With this background, we can walk through the right monad law proof and see that we start with x >>= return, the left hand side of the equality we are trying to prove. We then show that it is reflexively equal to x_1 , vs ++ [], and then use the congruence property and a standard library proof of the identity to show equality to x, the right-hand side of the equality we are trying to prove.

This approach is very readable, although somewhat verbose. In fact, cong (λ y → x_1 , y) (++-identity^r vs) has the same type as what we're trying to prove, which makes sense given that the only other step in it is reflexive. We also do not need a let expression to deconstruct the Writer: we can pattern match on it in the function definition. This more concise approach can be seen below as another proof of the same property. It is undoubtedly more concise, but is it more understandable? In this case

it is arguable, but as the proofs get longer and more complex, equational reasoning is much clearer to read and thus will be used as much as possible in the rest of this document.

```
rML' : ∀ {t} → (x : Writer (DataVal t)) → (x >>= return ) ≡ x
rML' (x₁ , vs) = cong (λ y → x₁ , y) (++-identityʳ vs)
```

In order to prove the associative property for bind, the third monad law, we must show that binding some (m_1 , m_2) : Writer a to a function g : a → Writer b, and binding the result to a function h : a → Writer c results in the same value as binding g to h in a lambda, which is bound to the initial (m_1 , m_2) value.

```
writer-assoc : ∀ {a b c}
  → (m : Writer a)
  → (g : (a → Writer b) )
  → (h : (b → Writer c) )
  → (m >>= g) >>= h ≡ m >>= (λ x → g x >>= h)
writer-assoc (m₁ , m₂) g h =
  let
    (gm₁ , gm₂) = g m₁
    (hgm₁ , hgm₂) = h gm₁
  in
  begin
  ((((m₁ , m₂) >>= g) >>= h))
  ≡⟨ refl ⟩
  hgm₁ , (m₂ ++ gm₂) ++ hgm₂
  ≡⟨ cong (λ x → hgm₁ , x) ( ++-assoc m₂ gm₂ hgm₂) ⟩
```

$$hgm_1 \;,\; m_2 \mathbin{+\!+} (gm_2 \mathbin{+\!+} hgm_2)$$
$$\equiv\langle \text{ refl } \rangle$$
$$(((m_1 \;,\; m_2) \gg= (\lambda\ x \to g\ x \gg= h))) \blacksquare$$

If we apply g to m_1, extracting the result as a pair of the value gm_1 and the log gm_2, and then apply h to gm_1 and extract the result as a pair (hgm_1 , hgm_2), we have exposed the components of the composition. We can show that the Writer resulting from the bind composition consists of a value equal to hgm_1, and a log equal to the concatenation of the log sections of the writers, (m_2 ++ gm_2) ++ hgm_2. We can then use the standard library proof that the ++ operator is associative over lists, showing that (m_2 ++ gm_2) ++ hgm_2 ≡ m_2 ++ (gm_2 ++ hgm_2). We can use our cong function to turn that into a proof that (hgm_1 , (m_2 ++ gm_2) ++ hgm_2) ≡ (hgm_1 , m_2 ++ (gm_2 ++ hgm_2)). Agda is able to normalize this to (m_1 , m_2) $\gg=$ (λ x → g x $\gg=$ h), which is the equality we were trying to prove, proving that our implementation holds for the third law.

5.1.2 Monads in monads in monads in...

We actually already introduced a monad in Chapter 3! Our first evaluator, the one that was not implemented with with a GADT and therefore had to account for the case that there was a type error, returned a `Maybe Value`. The `<-` arrow is syntactic sugar for constructing a series of >>= compositions. There are many such cases where there exists at least one input where the function cannot evaluate to a correct value. For example, an integer division function fails if a zero is passed into the denominator. Rather than just throwing a run-time error, this possibility for failure can be encoded in the type system, and rather than returning an ℤ, it can return Maybe ℤ, as our

evaluator did. The Maybe a type has two constructors, just a , and nothing. The monadic bind action, rather than aggregating a list, propagates failure forward so that (nothing ≫= f) ≡ nothing, but ((just x) ≫= f)≡ (just f x). The return injects the value into Maybe a by wrapping in in a just constructor. With these, we can define a chain of operations that propagate the failure forward if any link in the chain fails. We can define these operations below.

```
return : ∀{A} → {a : Set A} → (m : a) → Maybe a
return a = just a

_≫=_ : ∀{A B}
  → {b : Set B}
  → {a : Set A}
  → (m : Maybe a)
  → (a → Maybe b)
  → Maybe b
nothing ≫= f = nothing
just x ≫= f = f x
```

As evidence that this is faithfully applied, consider the proofs below that these follow the monad laws. The proof of the left monad law is reflexive as before, however the right monad law required us to case over the input Maybe value. The proof of associativity is also relatively simple, where when casing over the input and the result of the application of the supplied function f to m, every branch of the proof ending in a nothing value is reflexive, because they all have the type nothing ≡ nothing, and the one branch ending in a just value has type just y ≡ just y, and is thus reflexive as well.

```
lml : ∀{A B C : Set}
  → (a : A)
  → ( f : A → Maybe B )
  → (return a) >>= f ≡ (f a)
lml a f = refl

rml : ∀ {A} → {a : Set A } → (x : Maybe a) → (x >>= return) ≡ x
rml nothing = refl
rml (just x) = refl

maybe-assoc : ∀ {A B C} → {a : Set A} → {b : Set B} → {c : Set C}
  → (m : Maybe a)
  → (g : (a → Maybe b) )
  → (h : (b → Maybe c) )
  → (m >>= g) >>= h ≡ m >>= (λ x → g x >>= h)
maybe-assoc nothing g h = refl
maybe-assoc (just x) g h with g x
... | nothing = refl
... | just x1 = refl
```

This is a relatively simple monad though, so proving things about it directly was not that interesting. Let us take a step forward and nest our monads, so we have a Maybe, where the parameterizing type is a Writer. This results in a computation that either returns nothing, or a value with a log[7].

[7] There are better ways of dealing with combined effects than simply nesting them: two examples are monad transformers [Jon95] and algebraic effects [XL20]. Re-implementing the monad transformer library would be a thesis on its own, so instead, here they are nested, and the relevant functions manually implemented.

Looking at the the bind function, we can see that if we bind a nothing value to some function, the bind evaluates to a nothing. The interesting case is when a just value is being bound, in which case the result of the bind depends on the application of the function f to the value in the Writer a. This is accomplished by using a with construct. This captures both the ability for the calculation to fail and return nothing, and the log aggregating capability of the Writer monad.

```
return : ∀ {t} (a : t) → Maybe (Writer t)
return a = just (a , [])

_»='_ : ∀ {a b}
  → Maybe (Writer a )
  → (a → Maybe (Writer b))
  → Maybe (Writer b)
nothing »=' f = nothing
just (a , vs₁) »=' f with f a
...| nothing = nothing
...| just (b , vs₂) = just (b , vs₁ ++ vs₂)
```

For example, we can define a division function that takes two natural numbers, and if the denominator is non-zero, it returns a writer with the result of division as the value and a description of the call in the log. In all other cases, the division function returns a nothing value.

```
div : ℕ → ℕ → Maybe (Writer ℕ)
div n zero = nothing
div n (suc m) = just
```

```
    (n / (suc m) ,
    [ "divided " ++s shown n ++s " by " ++s shown (suc m) ]//)
```

For an illustration of a single operation in this monad, if we divide 2 by 4 we get a just value of 2 as our returned value, and a description of the operation in the log. If we divide that by zero however, we get a nothing value, as division by zero is not defined.

```
_ : div 4 2 ≡ just (2 , [ "divided 4 by 2" ]//)
_ = refl

_ : div 4 0 ≡ nothing
_ = refl
```

We can see that if we try and bind the result of this division function to another division, the failure propagates forward, whereas if we bind together two successful division operations, we have the correct resulting value and a log of both of the division steps.

```
_ : div 4 0 >>= (λ x → div 32 x) ≡ nothing
_ = refl

_ : div 4 2 >>= (λ x → div 32 x)
      ≡ just (16 , [ "divided 4 by 2" ,// "divided 32 by 2" ]// )
_ = refl
```

In order to prove the right monad law for the MaybeWriter monad, we have to do a proof by cases over the constructors. In the case that x is nothing, then the equality

that we are trying to prove is reflexive. In the case that it is a just value, then the same proof as in the right monad law proof of the Writer can be used if we wrap the Writer value in a just constructor.

```
rML : ∀ {a} → (x : Maybe (Writer a )) → (x »= return ) ≡ x
rML nothing = refl
rML (just (x₁ , x₂)) = cong (λ y → just (x₁ , y)) ( ++-identityʳ x₂)
```

The bind operator in this case has a difference in behavior depending on the result of the application of the function to the value in the Writer. We can encapsulate this change in behavior via another proof by cases, but over the result of the function application of f to a. Whether f a evaluates to a nothing value or a just value, the proofs are reflexive.

```
lML : ∀ {A B C : Set}
  → (a : A)
  → (f : A → Maybe (Writer C ))
  → ((return a) »= f ) ≡ (f a)
lML a f with f a
... | nothing = refl
... | just x = refl
```

In order to prove the final monad law, that of associativity, we first handle the case where the MaybeWriter is a nothing value, in which case the proof is reflexive. In the case that it is a just value, we can use a with construct to examine the cases when the first bound function is applied to the first value, x_1. As before, the nothing case is reflexive, but in the just case we can use a nested with expression to consider the

cases of the second bound function, h, applied to the result of the previous function application, y_1. Again the nothing case is reflexive, and in the just case we use the same proof as in the Writer associativity proof, with a just constructor wrapping the Writer in the cong function.

```
maybe-writer-assoc : ∀ {a b c}
  → (f : Maybe (Writer a))
  → (g : (a → Maybe (Writer b) ))
  → (h : (b → (Maybe (Writer c) )))
  → (f »= g) »= h ≡ f »= (λ x → g x »= h)
maybe-writer-assoc nothing g h = refl
maybe-writer-assoc (just (x₁ , x₂)) g h with g x₁
... | nothing = refl
... | just (y₁ , y₂) with h y₁
... | nothing = refl
... | just (z₁ , z₂) = cong (λ a → (just (z₁ , a ))) (++-assoc x₂ y₂ z₂)
```

Hopefully this has given the reader enough of a background in Agda to understand the following chapters enough to follow along. We have shown how Agda's notation is similar to that of Haskell, how dependent types allow one to seamlessly work with types as data and to prove properties of our programs. What we have not shown is how we can use these techniques to prove properties of languages, which we will do next chapter.

Chapter 6

SimpleLang

In the last chapter, we used dependent types to prove properties about data types and functions in our natively dependently typed language, Agda. Recall that our original motivation for entering the world of more advanced typing was to assure that variable lookups were well-typed in our Haskell interpreter in Chapter 3. This required function application on a type level to maintain the well-typed nature of the interpreter, as we needed to know what type variables had in context, and thus could not be completed with a non-dependently typed language. Now that we've had at least an introduction to such a language, we return to the previous question.

In this chapter, we explore a simple language embedded in Agda, that has been (extremely creatively) called SimpleLang[1], and it is effectively an Agda re-implementation of the language from Chapter 3. It is also an example of an *intrinsically-typed definitional interpreter*. The data types are *intrinsically-typed*, meaning that the type system of this language is included in the definition of the abstract data types that define the language. The interpreter that operates over these data types is a *definitional interpreter*, meaning that the semantics of the object language are defined in terms of a well-known host language, in this case Agda.

[1]This language is based on the introductory interpreter in the paper "Intrinsically-Typed Definitional Interpreters for Imperative Languages" [BRT+17]

6.1 Abstract syntax and interpreter

As we can see below, the basic structure of the expression, type, and value data types has remained the same when compared to our language defined in Chapter 3. The Haskell GADT `Expression` and Agda Expr have corresponding data types: `ExpNum` maps to num, `ExpPlus` maps to plus, and `ExpTrue` and `ExpFalse` map to the bool constructor. The `Value` and Val constructors also have a similar correspondence, of `ValNum` mapping to numV and `ValTrue` and `ValFalse` mapping to boolV. The only real differences across both types in the Agda translation are that the data types' parameter and index types are explicit. For example, instead of the value definition being parameterized by some typeless a, we say that Val is indexed by a Ty and returns a Set, an alias for Set_0. The Expr type is also a Set indexed by a Ty, but this time it is parameterized by a typing context, Γ. This means that each expression has its own context in which it can look up the types of variables, as shown in the var case.

```
data Ty : Set where
  int  : Ty
  bool : Ty

Ctx = List Ty

data Val : Ty → Set where
  numV : ℤ → Val int
  boolV : Bool → Val bool

data Expr (Γ : Ctx) : Ty → Set where
  num : ℤ      → Expr Γ int
  bool : Bool  → Expr Γ bool
```

```
var  : ∀ {t} → t ∈ Γ → Expr Γ t
plus : Expr Γ int → Expr Γ int → Expr Γ int
```

The variables in this interpreter do not hold strings representing a variable names, but rather are Any types, as introduced in the last chapter. This consists of the type of the variable, t, as well as a proof that t is present in the context Γ, which entails specifying the index of it. This in turn can be used to look up the value referenced by the index in the evaluation context at runtime. The $\in$ function is a way to create an Any from a list and a specified member. This ensures that variable lookup is well-typed because the type that parameterizes the var is guaranteed to be in the typing context Γ, and has a matching type to the var which is parameterized by the $\in$. This is enforced by the Expr being indexed by t, the same type as in the $\in$ expression.

The evaluation rules are similar to the language in Chapter 3, except that we now define an operator $+^v$, to correspond with the plus constructor, and implement the evaluation of plus in terms of that. An interesting point to notice is that the expression and the environment refer to the same type context. That means that the type level lookups in the abstract data type correspond to variable lookups at evaluation time. In this small language there is no way to introduce variables, other than passing in an initial context, so we only need to worry about looking up variables, not extending the context.

```
infixl 5 _+ᵛ_
_+ᵛ_ : Val int → Val int → Val int
numV v₁ +ᵛ numV v₂ = numV (v₁ + v₂)

eval : ∀ {Γ t} → Expr Γ t → Env Γ → Val t
```

```
eval (num x) env = numV x
eval (bool x) env = boolV x
eval (var x)  env = lookup env x
eval (plus e1 e2) env = eval e1 env +ᵛ eval e2 env
```

With great types come great responsibility, and we have unintentionally run across a complication with GADTs generally by leveraging our types in this manner. Our typing context is represented by a list of types, and our evaluation environment should be represented by a list of values, which we index by length from the front. Recall however that our Val types are indexed by a type, so a list of Vals would actually require a list of Val t, where the t varies depending on the expression. This kind of heterogeneous data structure is not representable with a normal list without some modification.

There is another means by which we can apply a predicate to members of a list, and that is the All data type. While the Any data type allows us to collect evidence that some member of a list satisfies a predicate, an All data type allows us to collect a list of elements that all satisfy a predicate. Recalling that a predicate in Agda is some function that returns a Set, we can notice that Val is already a predicate: it has type Ty → Set. We can use this insight to define Env as a function which takes a typing context Γ as a variable and generates an All type, allowing us to have a list of these heterogeneously typed Val items where the parameterizing types of Vals are supplied by Γ.

```
Env : Ctx → Set
Env Γ = All Val Γ
```

6.2 Proofs of programs

Consider a function called simplifyPlus that, given an expression, recursively simplifies plus expressions if both sides are known at compile time. Effectively, this would be pre-computing the additions at compile time, so we only perform those operations once rather than each time the program is executed, a form of constant folding.

In the case that the input to simplifyPlus is a plus with two num sub-expressions, simplifyPlus simply adds the sub-expressions and wraps the result in a num constructor. In the case that the plus constructor does not have two num sub-expressions, it recursively calls simplifyPlus on the sub-expressions. In every other case it acts as an identity.

$$\begin{aligned}
&\mathsf{simplifyPlus} : \forall \{\Gamma\ t\} \to \mathsf{Expr}\ \Gamma\ t \to \mathsf{Expr}\ \Gamma\ t\\
&\mathsf{simplifyPlus}\ (\mathsf{plus}\ (\mathsf{num}\ x)\ (\mathsf{num}\ y)) = (\mathsf{num}\ (x + y))\\
&\mathsf{simplifyPlus}\ (\mathsf{plus}\ e_1\ e_2\ ) = \mathsf{plus}\ (\mathsf{simplifyPlus}\ e_1)\ (\mathsf{simplifyPlus}\ e_2)\\
&\mathsf{simplifyPlus}\ (\mathsf{num}\ x) \qquad = \mathsf{num}\ x\\
&\mathsf{simplifyPlus}\ (\mathsf{var}\ x) \qquad = \mathsf{var}\ x\\
&\mathsf{simplifyPlus}\ (\mathsf{bool}\ x) \qquad = \mathsf{bool}\ x
\end{aligned}$$

How do we know that this does not change the semantics of the execution? It seems intuitive that adding these at compile time should correspond to adding them at runtime, but many things are both intuitive and incorrect. In essence, we want to show that, although simplifyPlus e is not necessarily the same as the expression e, they always evaluate to the same value if they are in the same context. As dependent types allow us to run arbitrary functions at a type level, we can encode this relationship as a type, as shown below. We again proceed with a proof by cases, where each case corresponds to a constructor of the Expr type. The proofs that this property holds for

non-recursive cases are reflexive, meaning they are trivially equivalent.

```
simplPreserves : ∀ {Γ} → {t : Ty} → ( e : Expr Γ t) → (env : Env Γ)
  → eval e env ≡ (eval (simplifyPlus e) env)
simplPreserves (num x) env = refl
simplPreserves (var x)   env = refl
simplPreserves (bool x) env = refl
```

...

This is not true when it comes to evaluating the plus case because, in certain cases, it does modify the expression if both sides are numbers at compile time. To prove this, we need a lemma showing that the simplifyPlus function distributes over plus under evaluation. This turns out to be extremely simple to prove, as shown below, because if one splits it down to the requisite cases, Agda will then correctly deduce that each case is reflexive.

```
simplifyPlusDistributes : ∀ {Γ} → ( e₁ e₂ : Expr Γ int) → (env : Env Γ)
  → eval (plus (simplifyPlus e₁) (simplifyPlus e₂)) env
    ≡ eval (simplifyPlus (plus e₁ e₂)) env
simplifyPlusDistributes (num x) (num x₁) env = refl
simplifyPlusDistributes (num x) (var x₁)   env = refl
simplifyPlusDistributes (num x) (plus b b₁) env = refl
```

...

With this lemma out of the way we can begin proving the last case of simplePreserves, that of the plus constructor. To prove this, we need the lemmas which are defined in

the let block of the function described below. Let us go through them before moving to the meat of the proof. Because this is a recursive data structure, we have access to the inductive hypotheses as recursive calls, so we can assume that simplPreserves holds for the subexpressions of plus, e_1 and e_2, and thus that eval e_1 env $\equiv$ eval (simplifyPlus e_1 env) and eval e_2 env $\equiv$ eval (simplifyPlus e_2 env). These are assigned to ihL and ihR respectively. Using congruence, we can show that given ihL, (eval e_1 env) $+^v$ eval e_2 env is equal to eval (simplifyPlus e_1 env) $+^v$ eval e_2 env. We can also show that eval (simplifyPlus e_1 env) $+^v$ eval e_2 env is equal to eval (simplifyPlus e_1 env) $+^v$ eval (simplifyPlus e_2 env), by using ihR.

```
simplPreserves (plus e₁ e₂) env =
  let
    ihL : eval e₁ env ≡ eval (simplifyPlus e₁) env
    ihL = simplPreserves e₁ env
    ihR : eval e₂ env ≡ eval (simplifyPlus e₂) env
    ihR = simplPreserves e₂ env
    addLPlus : (eval e₁ env ) +ᵛ (eval e₂ env)
               ≡ (eval (simplifyPlus e₁) env ) +ᵛ (eval   e₂ env)
    addLPlus =  cong (λ y → y +ᵛ (eval e₂ env) ) ihL
    addHPlus : (eval (simplifyPlus e₁) env ) +ᵛ (eval e₂ env)
               ≡ (eval (simplifyPlus e₁) env ) +ᵛ (eval (simplifyPlus e₂) env)
    addHPlus = cong (λ y → (eval (simplifyPlus e₁) env ) +ᵛ y) ihR
```

...

We can begin our main proof now by starting at the left-hand side of the equality that we are trying to prove, eval (plus e_1 e_2) env, and by using a sequence of lemmas

and reflexive equalities, deriving the corresponding right side: eval (simplifyPlus (plus e1 e2)) env .

```
in
  begin
  eval (plus e1 e2) env
  ≡⟨⟩
    eval e1 env +^v eval e2 env
  ≡⟨ addLPlus ⟩
  (eval (simplifyPlus e1) env ) +^v (eval e2 env)
  ≡⟨ addHPlus ⟩
  (eval (simplifyPlus e1) env ) +^v (eval (simplifyPlus e2) env)
  ≡⟨⟩
  eval (plus (simplifyPlus e1) (simplifyPlus e2)) env
  ≡⟨ simplifyPlusDistributes e1 e2 env ⟩
  eval (simplifyPlus (plus e1 e2)) env ■
```

Although proving properties in proof assistants is sometimes considered overly labor-intensive and pedantic, this proof closely tracked a paper and pencil proof in terms of steps and difficulty. Many of the things one would have to assume in a paper and pencil proof —that, for example, all terms are well typed or that all look-ups succeed —are assured by the type system, so we get them for free. In my opinion this is the primary benefit of encapsulating as much of the semantics of the language in its type system as possible, because many properties that we care about are trivially true if the program is well-typed. In addition, the type-checker gives feedback about what types things are, what they normalize to, and the goal of the current step, that

simply are not available with paper and pencil proofs.

The above optimization is relatively trivial, but it is a good model for some other proofs later in the thesis and was a useful case study in introducing proving properties of intrinsically-typed interpreters. However, it is not powerful enough to maximally simplify a series of added numbers known at compile time. Consider the expression below, and how the simplifyPlus function optimizes it.

```
_ : ∀ {Γ} → simplifyPlus {Γ}
        (plus
          (plus (num (+ 3)) (num (+ 2)))
          (plus (num (+ 11)) (num (+ 1))))
    ≡ plus (num (+ 5)) (num (+ 12))
_ = refl
```

It is obvious that one could add the num 5 and num 12 at compile time, but this function was not powerful enough to do it. In order to optimize compile-time addition fully, we need a more complicated optimization function, one that is able to recursively add from the bottom of the expression abstract syntax tree up.

6.3 A less simple simplifier

The above simplifyPlus function is limited in terms of being able to simplify nested plus constructors, only simplifying the most nested ones. Optimizing this maximally would mean calling the optimization function on the two sub-expression of the plus constructor, and then if both of the sub-expressions simplify to num, evaluating to their sum wrapped in a num constructor. This algorithm is captured in the below

definition, and we can also see that it fully simplifies the example expression that our last algorithm failed to above.

```
simplifyPlusR : ∀ {Γ t} → Expr Γ t → Expr Γ t
simplifyPlusR (num x) = num x
simplifyPlusR (var x)  = var x
simplifyPlusR (bool x) = bool x
simplifyPlusR (plus e₁ e₂ ) =
  case (simplifyPlusR e₁) , (simplifyPlusR e₂) of λ where
    (num x , num y) → num (x + y)
    (e₁′ , e₂′) → plus e₁′ e₂′

_ : ∀ {Γ} → simplifyPlusR {Γ}
        (plus
          (plus (num (+ 3)) (num (+ 2)))
          (plus (num (+ 11)) (num (+ 1))))
    ≡ num (+ 17)
_ = refl
```

Although this seems like a relatively trivial modification, the effort required to prove its correctness is significantly increased over the simplifyPlus optimization[2], particularly given the case splitting on the result of a recursive call. Importantly, the property that we were able to use above, that simplifyPlus distributes over the plus constructor, is not true in the general case, but only if the recursive calls do not both simplify to numbers.

[2]Proof repair, the process of automatically fixing proofs broken by changes of implementation, is a current area of research [Rin21].

In fact, this is the first lemma that we need to prove, that if either of the sub-expressions of a plus expression simplify to something other than a num, that simplifyPlusR distributes over the plus. Before we do this, however, we have to tackle two new concepts: ⊎ and ∃. ⊎ is the Agda encoding of a sum type, of which the Haskell `Either` type is an implementation. As such it can hold a value of one of two types. Haskell's `Left` and `Right` constructors are implemented in Agda as inj_1 and inj_2.

Up until now, all proofs we have constructed have been of the "for all" variety. These are referred to as *dependent product types*, or Π types [nLa21b]. The means by which one constructs a Π type is by supplying a proposition that one wants to show is true for all types T, P. Π allows one to lift some value x into the rest of the type, and defines a dependent type that for all $x : T$, $P(x)$. For example, if we wanted to prove that for all pairs of natural numbers, commutativity of addition holds, we could encode that in a type as $\Pi_{(m,n):\mathbb{N}\times\mathbb{N}}(m + n \equiv n + m)$. Conveniently Agda manages the Π types automatically, so we do not have to manually construct them[3].

The dual of dependent product types is *dependent sum types*, or Σ types [nLa21c]. As with Π types, Σ types take as parameters some $x : T$, and a predicate $P : T \to Set$. Unlike in Π types, Σ types assert that there exists some value x for which P holds. So, if one wanted to prove that 57 is a composite number[4], they could phrase that as "there exists a pair of natural numbers such that when you multiply them, they equal 57". The encoding of this type would simply be $\Sigma_{(m,n):\mathbb{N}\times\mathbb{N}}(m * n \equiv 57)$. To inhabit such a type, one constructs a pair where the first member is the input value that makes the proposition true, and the second is the proof, as shown in Figure 6.1. Σ types can be introduced in Agda programs with the ∃ data type.

[3]The book "The Little Typer" by Daniel P. Friedman and David Thrane Christiansen explores using dependent types where nothing is really automated in this way.

[4]This number was chosen because it looks like a prime number at first glance, but is not.

$$\frac{v : T \qquad proof : P(v)}{(v, proof) : \Sigma_{x:T}(P(x))}$$

Figure 6.1: Type introduction for Σ types

Now that we understand how to encode existentials and sum-types, we can move onto our proof of distributivity. We want to prove that if either of the subexpressions of a plus evaluate to a non-num term, that simplification distributes over the plus constructor. Phrased more exactly, we can take as input expressions e_1 and e_2, as well as a proof that either e_1 does not simplify to a num, or e_2 does not simplify to a num. Recalling that a `Not a` type in our Haskell logical encoding is simply a way of writing `a -> Void`, we can see that Agda's ¬ works in much the same way in the case where e_1 is pattern matched as a num value. We have a function `e1¬≡n` passed as in input that if given a proof that there does exist some num v_1 that equals e_1, generates a ⊥ value. Conveniently we have a value that makes this predicate hold true: x_1. We can then use ⊥-elim, Agda's version of `principleExplosion`, to finish the proof in this case. It may be useful to notice that the only reason we were able to generate ⊥ is because num was not really a valid constructor given a proof that was passed in. Thus ⊥-elim can be read as a sort of "nothing should be able to get here" indicator. In fact the rest of the proof (with many reflexive cases omitted) consists of cases that are either reflexive or use the ⊥-elim technique.

```
simplRNonNumDistrib : ∀ {Γ} (e₁ e₂ : Expr Γ int) →
  (¬ (∃[ v₁ ](simplifyPlusR e₁ ≡ num v₁)))
    ⊎
    (¬ (∃[ v₂ ](simplifyPlusR e₂ ≡ num v₂))) →
  (simplifyPlusR (plus e₁ e₂))
```

```
    ≡ (plus (simplifyPlusR e₁ ) (simplifyPlusR e₂ ))
simplRNonNumDistrib (num x₁) e₂ (inj₁ e₁¬≡n) = ⊥-elim (e₁¬≡n (x₁ , refl))
simplRNonNumDistrib (var x₁) e₂ (inj₁ x) = refl
simplRNonNumDistrib {Γ} (plus e₁ e₃) e (inj₁ ¬se₁≡num)
  with simplifyPlusR (plus e₁ e₃)
... | num x      = ⊥-elim (¬se₁≡num (x , refl))
... | var _      = refl
... | plus _ _   = refl
```

...

Now, onto proving our main proof of simplRPreserves's semantic maintenance. As before, this property holds reflexively for the non-recursive cases, but the recursive case requires quite a bit more effort. Importantly, unlike the simplPreserves proof where we only had to case over the constructors, now we also have to case over the results of simplifying the recursive expressions in the plus case. This helps capture that simplifying the plus constructor can result in two different behaviors depending on the results of the simplifications.

```
simplRPreserves : ∀ {Γ} → {t : Ty} → ( e : Expr Γ t) → (env : Env Γ)
  → eval e env ≡ (eval (simplifyPlusR e) env)
simplRPreserves (num x) env = refl
simplRPreserves (bool x) env = refl
simplRPreserves (var x)  env = refl
```

...

In order to prove these, we need a few additional lemmas, first of which are that the num and numV constructors are *injective*. In general, a function f is said to be injective if f x ≡ f y implies that x ≡ y. Although this may seem obvious, not all functions are injective (for example, the absolute value function is not), and so we need to prove it for the particular functions that we are interested in here. In addition, we need to prove that no var can equal a num, and that no plus can equal a num value, both of which can be proven with an absurd pattern, which indicates that none of the constructors are valid in this context [AAC+21c].

```
num-injective : ∀ {x y Γ} → num {Γ} x ≡ num y → x ≡ y
num-injective refl = refl

numV-injective : ∀ {x y} → numV x ≡ numV y → x ≡ y
numV-injective refl = refl


var≢num : ∀ {Γ x y} → ¬ (var {Γ} x ≡ num y)
var≢num = λ ()

plus≢num : ∀ {Γ x x₁ y} → ¬ (plus {Γ} x x₁ ≡ num y)
plus≢num = λ ()
```

Thinking back to the simplification function simplifyPlusR, given a plus constructor there are two basic behaviors the simplify function can result in. The first of these is when both of the subexpressions are able to be simplified at compile time to numbers. In this case, simplifying the plus expression results in a sum of those returned numbers. We can prove this in a lemma plusSimplifies', which simply says that if simplifyPlusR of e_1 and e_2 both equal some num values, then simplifyPlusR (plus e_1 e_2) can be simplified

to a num wrapping the sum of the resulting num values. We can prove this by using the with construct to case over the result of simplifying e_1 and e_2, which only results in one case as we have proofs that both are num values. We can then use the rewrite construct, which automatically substitutes the left hand side of the equality for the right on the right hand side of the of the =, to rewrite values e_1 and e_2 as num _ using the proofs accepted as parameters, after which the proof becomes reflexive.

```
plusSimplifies' :
  ∀ {Γ v₁ v₂}
  (e₁ e₂ : Expr Γ int)
  → simplifyPlusR e₁ ≡ (num v₁)
  → simplifyPlusR e₂ ≡ (num v₂)
  → simplifyPlusR (plus e₁ e₂) ≡ (num (v₁ + v₂))
plusSimplifies' e₁ e₂ sPRe₁≡v₁ sPRe₂≡v₂
  with (simplifyPlusR e₁) | (simplifyPlusR e₂)
... | num _ | num _ rewrite num-injective sPRe₁≡v₁ | num-injective sPRe₂≡v₂
   = refl
```

To understand the next section, it is important understand the inspect function [AAC+21g]. Essentially one can apply inspect' to a function application, for example f x, and it evaluates to both the result of the function application and an explicit proof that the function application equals the returned variable, as shown in Figure 6.2. So if f a ≡ x, then inspect' (f a) ≡ x with≡ proof, where proof : (f a) ≡ x.

If we look at the case in our main proof where we are simplifying a plus constructor with two subexpressions that simplify to numbers, we can use the with construct with the inspect' function to get a proof that they simplify to num values, and use a nested

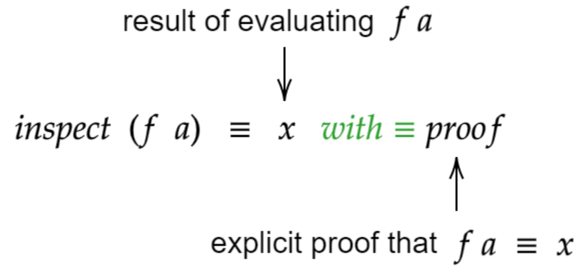

Figure 6.2: An intuition for how inspect works

with to show that those evaluate to numV values. Once we have these, we can set up a list of lemmas in our let block that we need in order to prove that the simplifyPlusR function does not change what the expression evaluates to. The first of these lemmas simply calls the plusSimplifies' function defined above. We then have a series of proofs relating the inductive hypotheses for both of the subexpressions: first that evaluating the subexpression is equal to evaluating the simplified version of the subexpression in the same environment (ih_1 and ih_2), that evaluating the simplified expression equals the numV value in the with block above ($\mathsf{intstep}_1$ and $\mathsf{intstep}_2$), and using the transitive property that evaluating each subexpression is equal to that numV (ih_{1t} and ih_{2t}).

```
simplRPreserves (plus e₁ e₂) env
  with inspect' (simplifyPlusR e₁) | inspect' (simplifyPlusR e₂)
... | num simpPRe₁ with≡ simpPRe₁≡ | num simpPRe₂ with≡ simpPRe₂≡
  with inspect' (eval (num simpPRe₁) env) | inspect'
                                          (eval (num simpPRe₂) env)
... | numV evSimpRe₁ with≡ evSimpRe₁≡ | numV evSimpRe₂
                                            with≡ evSimpRe₂≡
  =
```

```
let
  simplDistr : simplifyPlusR (plus e₁ e₂) ≡ num (simpPRe₁ + simpPRe₂)
  simplDistr = plusSimplifies' e₁ e₂ simpPRe₁≡ simpPRe₂≡
  ih₁ : eval e₁ env ≡ eval (simplifyPlusR e₁) env
  ih₁ = simplRPreserves e₁ env
  intstep₁ : eval (simplifyPlusR e₁) env ≡ numV simpPRe₁
  intstep₁ = cong (λ x → eval x env) simpPRe₁≡
  intstep₂ : eval (simplifyPlusR e₂) env ≡ numV simpPRe₂
  intstep₂ = cong (λ x → eval x env) simpPRe₂≡
  ih₁t : eval e₁ env ≡ numV evSimpRe₁
  ih₁t = trans ih₁ (trans intstep₁ evSimpRe₁≡ )
  ih₂ : eval e₂ env ≡ eval (simplifyPlusR e₂) env
  ih₂ = simplRPreserves e₂ env
  ih₂t = trans ih₂ (trans intstep₂ evSimpRe₂≡ )
```

...

Once we have these lemmas at our disposal, we can begin the main proof. We begin with eval e_1 env + eval e_2 env. Using our lemmas above we can rewrite each evaluation as the numV that it is equal to. We can show that this distributes, and the proof is simply reflexive because of how $+^v$ is defined. We can then use the evSimpRe proofs along with the proof of numV's injectivity to substitute the evSimpRe values with the simpRE values. This is reflexively equal to eval (num simPRe_1 + simPRe_2) env because of how evaluation of plus is defined.

```
in
```

```
begin
eval e₁ env +ᵛ eval e₂ env
≡⟨ cong₂ (λ x y → x +ᵛ y) ih₁ₜ ih₂ₜ ⟩
numV evSimpRe₁ +ᵛ numV evSimpRe₂
≡⟨ refl ⟩
numV (evSimpRe₁ + evSimpRe₂)
≡⟨ cong₂
    (λ x y → numV (x + y))
    (numV-injective (sym evSimpRe₁≡))
    (numV-injective (sym evSimpRe₂≡)) ⟩
numV (simpPRe₁ + simpPRe₂)
≡⟨ refl ⟩
eval (num (simpPRe₁ + simpPRe₂)) env
≡⟨ cong (λ pl → eval pl env) (sym simplDistr) ⟩
eval (simplifyPlusR (plus e₁ e₂)) env ∎
```

The other basic behavior of the simplify function is in the case where either subexpression fails to simplify to a num. Really this results in many cases, but we only include a single example, as all of the cases wind up having the same structure. We can see that the proof of this case looks almost exactly like the proof of the original simplification function above, however we need to justify with a lemma that we can distribute across the plus. We also need to supply a proof that at least one simplified value is not a num, which we can do by encoding a “there does not exist” statement. Constructively, we represent this by creating a function in which we assume that there does exist a value for which this is true, and generating a bottom value if one

is supplied. We can generate such a bottom value using our lemma that no var can equal a num, var≢num, as well as a proof that simplifying the second sub expression results in a var, simpe₂≡var.

```
simplRPreserves {Γ} (plus e₁ e₂) env | _ | var _ with≡ simpe₂≡var =
  let
    simple₂≢num : ¬ ∃ (λ v₂ → simplifyPlusR e₂ ≡ num v₂)
    simple₂≢num = λ { (fst , snd) → var≢num (trans
                                              (sym simpe₂≡var)
                                              snd)}
    ih₁ = simplRPreserves e₁ env
    ih₂ = simplRPreserves e₂ env
  in
  begin
  eval e₁ env +ᵛ eval e₂ env
  ≡⟨ cong₂ (λ a b → a +ᵛ b) ih₁ ih₂ ⟩
  eval (simplifyPlusR e₁) env +ᵛ eval (simplifyPlusR e₂) env
  ≡⟨ refl ⟩
  eval (plus (simplifyPlusR e₁) (simplifyPlusR e₂)) env
  ≡⟨ cong
       (λ a → eval a env )
       (sym (simplRNonNumDistrib e₁ e₂ (inj₂ simple₂≢num))) ⟩
  eval (simplifyPlusR (plus e₁ e₂)) env ∎
```

This is enough to finish our proof! Now we can pre-compute these additions in peace, knowing that applying the optimization will not cause something to go horribly

wrong in the program.

6.4 Approaches to optimization verification

We have already discussed at length the importance of proving that optimizations do not change the semantics of the programs we apply them to. In the case of well-formed programs, it is clear that this should be the case. However, what happens when we run an optimization on an ill-formed program? By using an intrinsically-typed interpreter, we have re-defined the domain of optimizations to that of valid programs, excluding ill-typed or ill-scoped expressions by definition. CompCert [Ler09], a well-known verified optimizing C compiler written in OCaml, takes a different approach. Rather than defining a correct-by-construction representation of an intermediate language, they construct an untyped representation and prove that if the source program is valid, then the optimized program is valid and maintains the semantics of the source program. To understand this approach, we can implement an untyped language definition, an interpreter that may fail, an optimization, and a proof using the optimization correctness framing used by CompCert. We can begin by defining our data types, which look much like they do in the intrinsically-typed definition, but with the context and indexed types removed from the definitions.

```
data Ty : Set where
  int  : Ty
  bool : Ty

data Val : Set where
  numV : ℤ → Val
  boolV : Bool → Val
```

```
data Expr : Set where
  num  : ℤ      → Expr
  bool : Bool   → Expr
  var  : String → Expr
  plus : Expr → Expr → Expr
```

Rather than using an All data type for our variables, we are using a String, which we look up in a keywordList data structure, as defined below.

```
keywordList : Set → Set → Set
keywordList a b = List (a × b)

lookup_a : keywordList String Val → String → Maybe Val
lookup_a [] _ = nothing
lookup_a ((key2 , val) :: ctx) key1 with key1 ≈? key2
... | false because _ = lookup_a ctx key1
... | true because _ = just val

Env : Set
Env = keywordList String Val
```

The evaluator is quite similar to the intrinsically-typed version, with the differences being the lookup function working over the keywordList, and a helper function plusEvalHelper — used to handle the possibility of type errors and failed variable lookups — in the plus case. This is operating in a Maybe monad, like the first Haskell evaluator in Chapter 3.

```
plusEvalHelper : Maybe Val → Maybe Val → Maybe Val
plusEvalHelper (just (numV x)) (just (numV y)) = just (numV (x + y))
```

```
plusEvalHelper _ _ = nothing

eval : Expr → Env → Maybe Val
eval (num x) env = just (numV x)
eval (bool x) env = just (boolV x)
eval (var x)  env = lookup_a env x
eval (plus e1 e2) env = plusEvalHelper (eval e1 env) (eval e2 env)
```

The simplifyPlus and simplifyPlusDistributes functions are identical to the ones implemented earlier in the chapter — with the exception being the type context and indexed types are removed from the Expr types — and thus will not be repeated here. We need two simple lemmas before we proceed further: one proving that just and nothing values cannot be equal, and another showing that plusEvalHelper evaluates to nothing when its second argument is nothing.

```
just≠nothing : {v : Val} → ¬ (just v ≡ nothing)
just≠nothing ()

plusEvalNothing : ∀ {e} → plusEvalHelper e nothing ≡ nothing
plusEvalNothing {nothing} = refl
plusEvalNothing {just (numV x)} = refl
plusEvalNothing {just (boolV x)} = refl
```

Through a simple set of equational reasoning steps, we can prove our next lemma: that if an expression e_2 evaluates to nothing that the evaluation of a plus with e_2 as its second parameter will also evaluate to nothing.

```
evalnothing :
  (e₁ e₂ : Expr)
```

```
    → (env : Env)
    → (eval e₂ env ≡ nothing)
    → (eval (plus e₁ e₂) env ≡ nothing)
  evalnothing e₁ e₂ env evale₂ =
    begin
    eval (plus e₁ e₂) env
    ≡⟨ refl ⟩
    plusEvalHelper (eval e₁ env) (eval e₂ env)
    ≡⟨ cong (λ x → plusEvalHelper (eval e₁ env) x) evale₂ ⟩
    plusEvalHelper (eval e₁ env) nothing
    ≡⟨ plusEvalNothing ⟩
    nothing ∎
```

Our last separate lemma proves that if a plus expression evaluates to a just value, its sub-expressions both evaluate to just values. This can be proven by casing over the results of evaluating the expressions, using ⊥-elim to fulfill the impossible case — where e_2 evaluates to nothing —, and creating a straightforward pair in the other case.

```
  evalSubEvals : ∀ {v}
    → (e₁ e₂ : Expr)
    → (env : Env)
    → (eval (plus e₁ e₂) env ≡ just v)
    → ((∃[ v₁ ](eval e₁ env ≡ just v₁)) × (∃[ v₂ ](eval e₂ env ≡ just v₂)))
  evalSubEvals e₁ e₂ env eq with eval e₁ env | inspect' (eval e₂ env)
  ... | just x | nothing with≡ x₁ rewrite x₁
    = ⊥-elim (just≠nothing (trans (sym eq) plusEvalNothing))
```

```
... | just x | just y with≡ x₁ = (x , refl) , (y , x₁)
```

Finally, we can move on to proving the optimization correct. We can see that this proof strategy is reflected in the type, where rather than proving the equality of the evaluation of optimized and unoptimized expressions, we prove that if some expression evaluates to a just value, its optimized form will evaluate to that same value. We can see that the num and bool cases are both reflexive, as in the intrinsically-typed proof. The variable case is different because of the lookup function's ability to fail. Because a proof that the source expression evaluates to a value is in scope, we can case over the result of evaluating the var, and use ⊥-elim to deal with the case that the lookup fails.

```
simplPreserves : ∀ {v}
  → (e : Expr )
  → (env : Env)
  → (eval e env ≡ just v)
  → ((eval (simplifyPlus e) env) ≡ just v)
simplPreserves (num x) env refl = refl
simplPreserves (bool x) env refl = refl
simplPreserves {v} (var x) env eq with eq | inspect' (eval (var x) env)
... | a | nothing with≡ x₁ rewrite x₁ = ⊥-elim (just≢nothing (sym a ))
... | a | just x₂ with≡ x₁ = a
```

...

In our last case, that of the plus constructor, we must define a set of lemmas as we did in the analogous intrinsically-typed proof. First, we can derive proofs that e_1 and e_2 evaluate to just values from the proof that plus e_1 e_2 evaluates to a just value

with our evalSubEvals function. We can then take these proofs and generate our right and left inductive hypotheses via recursive calls to simplPreserves. Finally, we can prove that the result of adding the values which the sub-expressions evaluate to with plusEvalHelper is equal to evaluating plus e_1 e_2.

```
simplPreserves {v} (plus e₁ e₂) env eq =
  let
    ((v₁ , evale₁≡v₁) , (v₂ , evale₂≡v₂)) = evalSubEvals e₁ e₂ env eq
    ihₗ : ((eval (simplifyPlus e₁) env) ≡ just v₁)
    ihₗ = simplPreserves e₁ env evale₁≡v₁
    ihᵣ : ((eval (simplifyPlus e₂) env) ≡ just v₂)
    ihᵣ = simplPreserves e₂ env evale₂≡v₂
    plusHelpEq : eval (plus e₁ e₂) env ≡ plusEvalHelper (just v₁) (just v₂)
    plusHelpEq =
      begin
      eval (plus e₁ e₂) env
      ≡⟨ refl ⟩
      plusEvalHelper (eval e₁ env) (eval e₂ env)
      ≡⟨ cong₂ (λ x y → plusEvalHelper x y) evale₁≡v₁ evale₂≡v₂ ⟩
      plusEvalHelper (just v₁) (just v₂) ∎
```

...

With these lemmas, we can begin our proof of the plus case. Through a sequence of equational reasoning steps, we can prove that the optimized plus expression evaluates to the same just value as the unoptimized version does, proving this optimization maintains the semantics of a validly constructed source expression.

```
in
begin
eval (simplifyPlus (plus e₁ e₂)) env
≡⟨ simplifyPlusDistributes e₁ e₂ env ⟩
eval (plus (simplifyPlus e₁) (simplifyPlus e₂)) env
≡⟨ refl ⟩
plusEvalHelper (eval (simplifyPlus e₁) env) (eval (simplifyPlus e₂) env)
≡⟨ cong (λ x → plusEvalHelper x (eval (simplifyPlus e₂) env)) ihₗ ⟩
plusEvalHelper (just v₁) (eval (simplifyPlus e₂) env)
≡⟨ cong (λ x → plusEvalHelper (just v₁) x) ihᵣ ⟩
plusEvalHelper (just v₁) (just v₂)
≡⟨ sym plusHelpEq ⟩
eval (plus e₁ e₂) env
≡⟨ eq ⟩
just v ∎
```

We can see that the simplifyPlus optimization can be proven correct in the intrinsically-typed and untyped styles. Much of the proof structure is the same in these two proofs: they both require us to use the inductive hypotheses to re-write the parameters of the respective addition functions, a distributivity of optimization lemma, and the num and bool cases are reflexive. The differences mainly arise from a difference in the base of truth in the proof: intrinsically-typed interpreters give us assurances by data type design, whereas truths of untyped interpreters must be derived from their evaluator behavior. For example, in the intrinsically-typed version of the proof, the fact that variable lookups always succeed is guaranteed by the type

of the `var` constructor, whereas in the untyped version, we have to derive successful lookups from the successful evaluation of the source expression. In addition, in the untyped case, we needed to derive that the evaluation of sub-expressions succeeds if the evaluation of the parent expression succeeds manually, which is a required input of recursive calls.

In the case of this specific proposition, both proofs seemed approximately equivalent in terms of effort required. Although the untyped proof is much longer and requires more lemmas, the data type design was trivial, whereas the design of the intrinsically-typed data types requires considerably more time, especially as the language complexity increases. The untyped approach is well-trodden ground, having been used in major compiler and optimizer verification efforts [Ler09], so for the rest of this thesis, our proofs will be in the intrinsically-typed style.

Hopefully this chapter was a sufficient introduction to understand what intrinsic typing means, how it is useful in verifying properties of languages, and the basics of using dependent types to verify properties of programs. Sadly, in this simple of a language, we have exhausted the useful optimizations available for it, and it is not clear how these relate to MIL's optimizations, as these two languages differ substantially in terms of basic structure. In the next chapter, let us work with something that looks a little more like MIL.

Chapter 7

Featherweight (M)IL

Now we can see the path the rest of this thesis will take: building progressively more complicated interpreters to work from what is effectively the simplest language available to something much more like the full functionality of MIL, and implementing and proving optimizations based on the new features correct along the way. The hope is that this work can eventually be extended to a full MIL implementation.

In this chapter, we introduce a language called *Featherweight MIL*, a very small subset of the full MIL. This language effectively consists of the the functionality of SimpleLang in the previous chapter, but without the ability to nest expressions. The basic computational unit in MIL is a Tail, as shown below. We have types called Word and Flag, which are analogous to int and bool respectively. The hope is that, by reasoning about these with a type that corresponds to the implementation but is slightly simplified, we can effectively smooth out the difficulty curve for those attempting to understand what the MIL language is really doing, as opposed to jumping directly to a full implementation.

7.1 Abstract syntax and interpreter

The data types of this language look very much like SimpleLang in MIL's clothing. For the results of computation, akin to Vals in last chapter, we have primitive values that can either carry Integers ($\mathbb{Z}$) or Booleans (Bool), and which are parameterized by their type. For input types, Atoms can either be variables (Var) or literals (Val), and are also parameterized by a context Γ, and indexed by the type of the value they carry.

Tails are either Return expressions (which contain an Atom which they share an indexed type with), or a PrimCall (which require a primitive binary operation, PrimOp, and two Atoms). Something to note here is that the PrimCall type definition enforces that the parameter types of the supplied PrimOp match the types that index both the input Atoms, and the result of the calculation. In this way, we assure that PrimCalls are guaranteed to work and generate a value of the proper type at run time. Note that although the input and output types are the same for the listed PrimOps, this design can support differing input and output types, and can be extended to support PrimOps of arbitrary arity. The data definitions of this language are described below:

```
data Ty : Set where
  Word : Ty
  Flag : Ty

Ctx = List Ty

data DataVal : Ty → Set where
  W : ℤ → DataVal Word
  F : Bool → DataVal Flag
```

```
data PrimVal : Ty → Set where
  I : ℤ  → PrimVal Word
  B : Bool → PrimVal Flag

data Atom (Γ : Ctx) : Ty → Set where
  Var : ∀ {t} → t ∈ Γ → Atom Γ t
  Val : ∀ {t} → PrimVal t → Atom Γ t

data PrimOp : Ty → Ty → Ty → Set where
  Mmul : PrimOp Word Word Word
  Mand : PrimOp Flag Flag Flag

data Tail (Γ : Ctx) : Ty → Set where
  Return : ∀ {t} → Atom Γ t → Tail Γ t
  PrimCall : ∀ {t₁ t₂ t₃} → PrimOp t₁ t₂ t₃
             → Atom Γ t₁ → Atom Γ t₂ → Tail Γ t₃
```

The evaluation strategy for this language differs slightly from SimpleLang. Instead of the language consisting of one expression type which is evaluated to values, Featherweight MIL splits programs into combinations of Tails, PrimOps, and Atoms, each of which requires its own small interpreter. Other than this structural difference, the basic approach to evaluation is the same.

As in last chapter, we define operators over DataVals as our grounding of the semantics of the evaluation in Agda's semantics. Specifically we implement multiplication ($*^d$) and conjunction ($\wedge^d$) as follows:

```
infixl 5 _*ᵈ_
_*ᵈ_ : DataVal Word → DataVal Word → DataVal Word
```

```
W a *^d W b = W (a Int.* b)

infixl 5 _∧^d_
_∧^d_ : DataVal Flag → DataVal Flag → DataVal Flag
F a ∧^d F b = F (a ∧ b)
```

Our evalAtom function is similar to the evaluation strategy of last chapter: a lookup is used in the case of a Var, and in the other cases the literal value is transferred from inside the Atom to the corresponding DataVal. Note that, as before, the type indexing the DataVal and the Atom is shared, to assure that our interpreter respects the typing correspondence between them.

```
evalAtom : ∀ {Γ t } → Env Γ → Atom Γ t → DataVal t
evalAtom env (Var x) = lookup env x
evalAtom env (Val (I x)) = W x
evalAtom env (Val (B x)) = F x
```

The evalPrimCall function works similarly to evaluating the Plus constructor, except that it determines which operation to apply to the two operands based on the PrimOp supplied. The type of this function assures that the operands and the DataVal resulting from the operation match the types indexing the PrimOp.

```
evalPrimCall :
  ∀ {Γ t₁ t₂ t₃} →
    Env Γ →
    PrimOp t₁ t₂ t₃ →
    Atom Γ t₁ →
```

```
    Atom Γ t₂ →
    DataVal t₃
evalPrimCall env Mmul a₁ a₂ = evalAtom env a₁ *ᵈ evalAtom env a₂
evalPrimCall env Mand a₁ a₂ = evalAtom env a₁ ∧ᵈ evalAtom env a₂
```

Finally, the eval function either evaluates a Return value by evaluating its Atom using evalAtom, or evaluates a PrimCall using the evalPrimCall function.

```
eval : ∀ {Γ t} → Tail Γ t → Env Γ → DataVal t
eval (Return x) env = evalAtom env x
eval (PrimCall primOp a₁ a₂) env = evalPrimCall env primOp a₁ a₂
```

With this we have our — very oversimplified — MIL interpreter! We can write a simplifyTimes function, as we did in our last chapter, that pre-computes the multiplication of a PrimCall Mmul if and only if both of the Atoms are known at compile time. Because of how simple the language design is at this time, the proof of each case is reflexive! Because we have embedded as much type information as we have in the data types themselves we get this proof effectively for free, so long as we case split down to all of the cases. We need to split this to cases because without a known constructor for the Tail we have no way of knowing what specifically it evaluates to, which is required to show that an optimization does not change the result of that evaluation.

```
simplifyTimes : ∀ {Γ t } → Tail Γ t → Tail Γ t
simplifyTimes (PrimCall Mmul (Val (I x)) (Val ( I y))) =
  Return (Val (I ( x * y )))
simplifyTimes e = e
```

```
simplifyTimesPreserves : ∀ {Γ t env} → (tail : Tail Γ t)
  → eval tail env ≡ eval (simplifyTimes tail) env
simplifyTimesPreserves (Return x) = refl
simplifyTimesPreserves (PrimCall Mand (Var x) (Var x₁)) = refl
simplifyTimesPreserves (PrimCall Mand (Var x) (Val x₁)) = refl
simplifyTimesPreserves (PrimCall Mand (Val x) x₂) = refl
simplifyTimesPreserves (PrimCall Mmul (Var x) (Var x₁)) = refl
simplifyTimesPreserves (PrimCall Mmul (Var x) (Val x₁)) = refl
simplifyTimesPreserves (PrimCall Mmul (Val (I x)) (Var x₁)) = refl
simplifyTimesPreserves (PrimCall Mmul (Val (I x)) (Val (I x₁))) = refl
```

7.2 A change of type

Although this is a very simple language, there are additional optimizations and properties that we can prove about those optimizations that are interesting and show how we might prove similar properties in more advanced interpreters. Consider that, even though we have Word and Flag types, eventually all of our variables will be represented as machine words when running on a physical machine. This implies that there is a way to convert booleans and boolean operations to word and word operations, while maintaining semantics.

If we convert the Boolean $\wedge^d$ conjunction operation into a $*^d$ multiplication operation, and map F true to W 1 and F false to W 0, we can see that this intuitively maintains a kind of semantics. By drawing out a truth table of the results of $\wedge^d$ and $*^d$ over all possible inputs, we can see a correspondence between these two operations, even though they are not literally equal. In the case that either input to $*^d$ is a W

0, the whole expression evaluates to a W 0, and in the case where both are W 1 the expression evaluates to W 1. In the same way, if either input to $\wedge^d$ is a F false the whole expression evaluates to an F false, and only in the case where both are F true does it evaluate to F true.

a	b	a $*^d$ b
W 0	W 0	W 0
W 1	W 0	W 0
W 0	W 1	W 0
W 1	W 1	W 1

(a) $*^d$ Truth table

a	b	a $\wedge^d$ b
false	false	false
true	false	false
false	true	false
true	true	true

(b) $\wedge^d$ Truth table

Table 7.1: Illustration of the $*^d$, $\wedge^d$ correspondence

So what happens we when try to write a function that converts our boolean operators to integer operators? Although this would be very simple to do in a non-intrinsically typed language, we will soon see that the use of this kind of language greatly complicates the matter, not just in the optimization function but also in the proof. If we were to copy the type from previous optimization, that is ∀ {Γ t env} → Tail Γ t → Tail Γ t, we would quickly run into a type error, as when t is a Flag, then the input and output t do not match! To implement this, we need to build it from the bottom up, implementing appropriate casts.

We first must write a function to process Atoms, so that Flags are converted to Words, but Words are untouched. This runs into the same typing problems as before because the output type may change depending on what t is. Conveniently we can write a function to selectively cast types with relative ease!

```
cBool : Ty → Ty
cBool Word = Word
cBool Flag = Word
```

With this, we can bring the output type into correspondence with the input type. The casts for literals are quite easy as well, we can simply use our castBool function below to cast Flags to Words.

$$\mathsf{castBool} : \mathsf{Bool} \to \mathbb{Z}$$
$$\mathsf{castBool}\ \mathsf{false} = (+\ 0)$$
$$\mathsf{castBool}\ \mathsf{true} = (+\ 1)$$

Once we begin to implement this cast for variables, however, things become more complicated. Recalling that our variables are implemented as a proof of membership in the Atom's typing context, it should become clear that we need to change the typing context to reflect this in the cast Atom. In order to do this, we need to write functions that cast everything in the typing context, and prove that the cast type is still present in the type-cast environment. The type environment cast cBoolTyEnv is simply accomplished by mapping our cBool function across the type environment as shown below.

$$\mathsf{cBoolTyEnv} : \mathsf{Ctx} \to \mathsf{Ctx}$$
$$\mathsf{cBoolTyEnv}\ ctx = \mathsf{map}\ \mathsf{cBool}\ ctx$$

If we consider the case that an Atom is a Var of type Word, in order for the proof of membership to match the updated type we need to show that given the proof Flag $\in \Gamma$, Word $\in$ cBoolTyEnv Γ, which follows by simple induction as shown in lookupConvF. A similar argument can be made in the case that the Atom is of type Word, which is proven in lookupConW, but is omitted here due to it being almost identical to lookupConF. Given these, we can finish writing our Atom casting function, castAtom.

$$\mathsf{lookupConvF} : \forall\ \{\Gamma : \mathsf{Ctx}\} \to \mathsf{Flag} \in \Gamma \to \mathsf{Word} \in \mathsf{cBoolTyEnv}\ \Gamma$$

```
lookupConvF (here refl) = here refl
lookupConvF (there xs) = there (lookupConvF xs)


castAtom : ∀ {t} → {Γ : Ctx} → Atom Γ t → Atom (cBoolTyEnv Γ) (cBool t)
castAtom {Word} {Γ} (Var x) = Var {cBoolTyEnv Γ} (lookupConvW x)
castAtom {Flag} {Γ} (Var x)  = Var {cBoolTyEnv Γ} (lookupConvF x)
castAtom (Val (I x)) = Val (I x)
castAtom (Val (B x)) = (Val (I (castBool x)))
```

Now that we have our castAtom function implemented, it is relatively trivial to implement a function to convert our Flags to Words, and convert PrimCall Mand to PrimCall Mmul in doing so.

```
collapseBool : ∀ {Γ t } → Tail Γ t → Tail (cBoolTyEnv Γ) (cBool t)
collapseBool (Return x) = Return (castAtom x)
collapseBool (PrimCall Mmul x y) = PrimCall Mmul (castAtom x) (castAtom y)
collapseBool (PrimCall Mand x y) = PrimCall Mmul (castAtom x) (castAtom y)
```

7.3 A proof of correspondence

As far as proving this optimization correct goes, it is essential to focus on what exactly we are trying to prove. In the previous optimization, correctness meant that applying some optimization function to an expression and then evaluating it was equal to the result of evaluating the original expression if the context was the same. That is not what we are saying here because — as we have already mentioned — this optimization maintains a correspondence between the input and output behavior, not

equality. For example, if we have a function that simply returns F true, this optimized version would return a W 1. To capture this correspondence, we need a function to convert the result of an evaluation of the optimized function to the type of the unoptimized function. We can write analogs to our cBool and cBoolTyEnv functions to operate over DataVals and evaluation contexts in order to finish the other half of the correspondence.

```
castDataVal : ∀ {t} → DataVal t → DataVal (cBool t)
castDataVal (F x) =  W (castBool x)
castDataVal (W x) = W x


simplifyEnv : ∀ {Γ} → Env Γ → Env (cBoolTyEnv Γ)
simplifyEnv [] = []
simplifyEnv (px :: env) = castDataVal px :: simplifyEnv env
```

With these two functions implemented, we can formalize this correspondence as saying that evaluating the Tail and then casting the result to a Word with castDataVal always results in the same output as running our optimization function, collapseBool, and then evaluating the Tail. This can be visualized by the commutative diagram shown in Figure 7.1, and represented by the type:

```
∀ Γ t env →
(tail : Tail Γ t) →
castDataVal (eval tail env) ≡ eval (collapseBool tail) (simplifyEnv env).
```

Rather than diving straight into the proof inhabiting that type, let us start with some lemmas which will make the proof substantially easier to understand. We can start by defining certain simple lemmas about how $\wedge^d$ and $*^d$ work. First, it will be

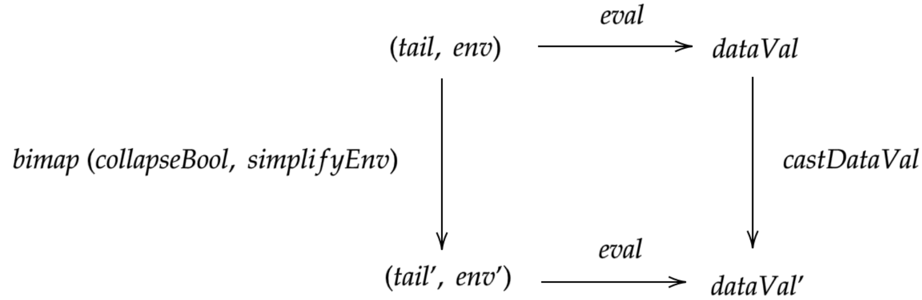

Figure 7.1: The correspondence between collapseBool, simplifyEnv, and castDataVal

useful later in this chapter to have proofs that $\wedge^d$ and $*^d$ are both commutative. It will also be necessary to show that F false $\wedge^d$ x always evaluates to F false no matter what x is. Next, we need a proof that W zero times anything equals W zero. Finally, we need a proof that no expression can equal both F true and F false. All of these lemmas are shown below:

$\wedge^d$-comm : $\forall\ \{a\ b\} \rightarrow a \wedge^d b \equiv b \wedge^d a$
$\wedge^d$-comm {F x} {F y} = cong F ($\wedge$-comm x y)

$*^d$-comm : $\forall\ \{x\ y\} \rightarrow x *^d y \equiv y *^d x$
$*^d$-comm {W x} {W y} = cong W ($*$-comm x y)

$\wedge^d$-false-left-inv : $\forall\ \{a\} \rightarrow$ F false $\wedge^d a \equiv$ F false
$\wedge^d$-false-left-inv {F false} = refl
$\wedge^d$-false-left-inv {F true} = refl

$*^d$-zeroVar^l : $\forall\ \{x\} \rightarrow$ W (+ 0) $*^d x \equiv$ W (+ 0)
$*^d$-zeroVar^l {W x} = refl

```
true≢false : ∀ {x} → x ≡ F false → x ≡ F true → ⊥
true≢false {F false} refl ()
```

We can leverage the falseLeftInv lemma in our multFalse proof below, to prove that this optimization of the $\wedge^d$ to $*^d$ holds true in the case that the first Atom evaluates to F false. We begin by approaching this via cases, casing over the result of inspecting the evaluation of x using the with construct. Normally we could just use the with construct to case over the evaluation of x itself, and Agda is intelligent enough to determine that x must evaluate to F false, given that a proof of this is in context. Later in this proof we do not want the proof of (evalAtom env x) ≡ F false being rewritten as F false ≡ F false, which is what happens when we attempt to case directly over the evaluation. Using the inspect' function only adds one extra case, where (evalAtom env x) ≡ F true, where we can simply use ⊥-elim to convert the ⊥ value generated by passing the proofs that (evalAtom env x) equals both F true and F false to true≢false.

```
multFalse :
  ∀ {Γ env}
  → (y : Atom Γ Flag)
  → (x : Atom Γ Flag)
  → (evalAtom env x) ≡ F false
  → castDataVal (evalAtom env x ∧ᵈ (evalAtom env y))
      ≡ evalAtom (simplifyEnv env) (castAtom x) *ᵈ
        evalAtom (simplifyEnv env) (castAtom y)
multFalse {Γ} {env} y x eq with inspect' (evalAtom env x)
... | F true with≡ evx≡True
  = ⊥-elim (true≢false {evalAtom env x} eq evx≡True )
```

Moving on to the case where x evaluates to F false, we begin with our initial expression on the left-hand side of the equality, castDataVal (F false $\wedge^d$ (evalAtom env y)). Using the lemmas defined earlier in this section, we can, through a series of equational reasoning steps, prove the equivalence to evalAtom {cBoolTyEnv Γ } simplifyEnv env) (castAtom x) $*^d$ evalAtom (simplifyEnv env) (castAtom y), finishing the proof.

```
... | F false with≡ evx≡False rewrite evx≡False =
  begin
    castDataVal (F false ∧ᵈ (evalAtom env y))
  ≡⟨ cong castDataVal ∧ᵈ-false-left-inv ⟩
    castDataVal (F false)
  ≡⟨ refl ⟩
    W (+ 0)
  ≡⟨ sym *ᵈ-zeroVarˡ ⟩
    W (+ 0) *ᵈ
        evalAtom (simplifyEnv env) (castAtom y)
  ≡⟨ refl ⟩
        castDataVal (F false) *ᵈ
        evalAtom (simplifyEnv env) (castAtom y)
  ≡⟨ cong
    (λ a → castDataVal a
        *ᵈ evalAtom (simplifyEnv env) (castAtom y))
    (sym evx≡False)⟩
    castDataVal (evalAtom env x) *ᵈ
```

```
      evalAtom (simplifyEnv env) (castAtom y)
    ≡⟨ cong
      (λ a → a
      *ᵈ evalAtom (simplifyEnv env) (castAtom y))
      (evalFAtom' {Γ} x )⟩
      evalAtom {cBoolTyEnv Γ} (simplifyEnv env) (castAtom x) *ᵈ
        evalAtom (simplifyEnv env) (castAtom y) ■
```

Now that this lemma has been proven, we can go on to write an additional lemma, which captures the case when the Tail to be optimized is a PrimCall Mand. In the case that the first Atom of the PrimCall evaluates to F false, we can directly call our multFalse function, supplying the proof returned by the inspect' function as input.

```
cast∧to* :
  ∀ {Γ}
  → {env : Env Γ}
  → (x : Atom Γ Flag)
  → (y : Atom Γ Flag)
  → castDataVal (eval (PrimCall Mand x y) env)
     ≡ eval (collapseBool (PrimCall Mand x y)) (simplifyEnv env)
cast∧to* {Γ} {env} x y
  with inspect' (evalAtom env x) | inspect' (evalAtom env y)
... | F false with≡ evx≡False | F _ with≡ _
  = multFalse y x evx≡False
```

The next case, where the second Atom in the PrimCall evaluates to F false, is a mirror of the first. For this case, we need to prove that :

castDataVal (evalAtom env x $\wedge^d$ evalAtom env y)

$\equiv$

evalAtom (simplifyEnv env) (castAtom x) $*^d$ evalAtom (simplifyEnv env) (castAtom y)

We can use the $\wedge^d$-comm lemma to rewrite the left-hand side of the equality to flip the Atoms, so that y is on the left and x is on the right, which refines the goal to castDataVal (evalAtom env y $\wedge^d$ evalAtom env x)

$\equiv$

evalAtom (simplifyEnv env) (castAtom x) $*^d$ evalAtom (simplifyEnv env) (castAtom y)

We can then rewrite the goal again with $*^d$-comm to further refine the goal to :

castDataVal (evalAtom env y $\wedge^d$ evalAtom env x)

$\equiv$

evalAtom (simplifyEnv env) (castAtom y) $*^d$ evalAtom (simplifyEnv env) (castAtom x)

Given that y is the variable we know to evaluate to F false this is the same goal as the first case, so we can use the same multFalse function but flip the input variables x and y's order.

```
... | F true with≡ _ | F false with≡ evy≡False
  rewrite (∧ᵈ-comm {evalAtom env x} {evalAtom env y})
        | *ᵈ-comm
          {(evalAtom (simplifyEnv env) (castAtom x))}
          {(evalAtom (simplifyEnv env) (castAtom y))}
```

```
                = multFalse x y evy≡False
```

In the final case, we must consider what happens when both of the PrimCall's Atoms evaluate to true. By using the rewrite construct with the proofs resulting from our inspect' calls, we can refine the beginning of our proof from castDataVal (evalAtom env x $\wedge^d$ evalAtom env y) to W (+ 1). Via a short sequence of equational reasoning steps as before, we can show this to be equal to evalAtom (simplifyEnv env) (castAtom x n), and in doing so finish the proof.

```
... | F true with≡ evx≡True | F true with≡ evy≡True
  rewrite evx≡True | evy≡True | castTrue {Γ} =
  begin
    W (+ 1)
  ≡⟨ refl ⟩
    castDataVal (F true) *ᵈ castDataVal (F true)
  ≡⟨ Eq.cong₂
    (λ a b → castDataVal a *ᵈ castDataVal b )
    (sym evx≡True) (sym evy≡True) ⟩
    castDataVal (evalAtom env x) *ᵈ castDataVal (evalAtom env y)
  ≡⟨ Eq.cong₂ (λ a b → a *ᵈ b) (evalFAtom' x) (evalFAtom' {Γ} y ) ⟩
    evalAtom (simplifyEnv env) (castAtom x) *ᵈ
      evalAtom (simplifyEnv env) (castAtom y) ∎
```

Finally, we can begin our main proof. The cases where Tail is a Return constructor are not particularly interesting. When tail just returns a Val, this optimization does not do anything, so the proof of the correspondence is reflexive. In the case where

a Var is being returned, this can be can be proved via evalFAtom and evalWAtom, depending on the type of the Var.

```
collapseBoolPreserves :
  ∀ {Γ t env}
  → (tail : Tail Γ t)
  → castDataVal (eval tail env)
    ≡ eval (collapseBool tail) (simplifyEnv env)
collapseBoolPreserves (Return (Val (I x))) = refl
collapseBoolPreserves (Return (Val (B x))) = refl

collapseBoolPreserves {Γ} {Flag} {env} (Return (Var x))
  = evalFAtom {Γ} {env}
collapseBoolPreserves {Γ} {Word} {env} (Return (Var x))
  = evalWAtom {Γ} {env}
```

The interesting cases are when the tail is a PrimCall, as these materially change the tails. In the case that the PrimCall is a Mand we can simply call our cast∧to* function, as proved earlier.

```
collapseBoolPreserves (PrimCall Mand x y) = cast∧to* x y
```

To prove the final case, when the Tail is a PrimCall Mmul, we can use the with construct to inspect the result of evaluating the first and second Atoms in the Tail, x_1 and x_2. We see this results in only one case as the type definitions for PrimCalls and the evalAtom are sufficiently constraining as to restrict the results of computation to the type indexing the PrimCall, in this case a Word.

```
collapseBoolPreserves {Γ} {Word} {env = env} (PrimCall Mmul x₁ x₂)
  with inspect' (evalAtom {Γ} env x₁) | inspect' (evalAtom env x₂)
... | W y₁ with≡ evx₁≡y₁ | W y₂ with≡ evx₂≡y₂ =
   begin
     castDataVal (evalAtom env x₁ *ᵈ evalAtom env x₂)
   ≡⟨ castDataVal-id (evalAtom env x₁ *ᵈ evalAtom env x₂) ⟩
     (evalAtom env x₁ *ᵈ evalAtom env x₂)
   ≡⟨ Eq.cong₂
       (λ a b → a *ᵈ b )
       (sym (castDataVal-id (evalAtom env x₁)))
       (sym (castDataVal-id (evalAtom env x₂))) ⟩
     castDataVal (evalAtom env x₁) *ᵈ
       castDataVal (evalAtom env x₂)
   ≡⟨ Eq.cong₂ (λ a b → a *ᵈ b) (evalWAtom' x₁) (evalWAtom' x₂) ⟩
     evalAtom (simplifyEnv env) (castAtom x₁) *ᵈ
       evalAtom (simplifyEnv env) (castAtom x₂) ∎
```

In the final case of this proof, a small reflexive lemma castDataVal-id, as defined below, simply shows that if a DataVal has type Word, castDataVal acts as an identity.

```
castDataVal-id :
  (x : DataVal Word)
  → castDataVal x ≡ x
castDataVal-id (W x) = refl
```

Beginning the proof, we can see that we start with a castDataVal (evalAtom env x_1 $*^d$ evalAtom env x_2). We can use the castDataVal-id to strip off the call to castDataVal, and

leave us with the multiplication alone. We can then use cong_2, and use castDataVal-id in the reverse order using the sym, to apply a castDataVal to each of the inputs to $*^d$. If we use another cong_2, we can apply our evalWAtom' lemma to convert each castDataVal (evalAtom env x_n) to evalAtom (simplifyEnv env) (castAtom x_n), completing the proof.

Even with this very simple interpreter, we can gain interesting perspectives on language optimizations. In this chapter we implemented an analogous optimization to the the simple addition optimization that we proved correct last chapter, and proved correct the conversion from Flags to Words, which changed the proposition we need to prove in interesting ways. In the next chapter, we will further increase the complexity of our interpreter by adding the ability to introduce variables, and through additional optimizations which depend on this feature, see how the increased complexity translates to increased proof difficulty.

Chapter 8

Pure (M)IL

In the last chapter, we dealt with a language that is so minimal it does not even let one introduce variables. This seems like a problem, given that programmers often want to introduce variables, so let us fix that by adding more advanced features to our language.

8.1 Abstract syntax and interpreter

The Ty and DataVal constructors are the same as in the last chapter, although the PrimVal data type has been changed. By taking a function that takes a Ty and returns a Set, we can have a single constructor for PrimVal that gives the flexibility to represent Words or Flags with a single Constant constructor, which greatly reduces the size of the related proofs. We also added a new PrimOp, that of addition.

```
constant : Ty → Set
constant Word = ℤ
constant Flag = Bool

data PrimVal : Ty → Set where
  Constant : (t : Ty) → constant t → PrimVal t
```

```
data PrimOp : Ty → Ty → Ty → Set where
  Madd : PrimOp Word Word Word
  Mand : PrimOp Flag Flag Flag
  Mmul : PrimOp Word Word Word
```

The most important addition to this language is that of a *code sequence* data type, Code, which allows us to compute a Tail and then introduce the resulting DataVal into the context in the nested code sequence. The code sequences are split up into two different constructors as shown below, a Ta constructor which simply wraps a Tail, and a Bind constructor.

```
data Code (Γ : Ctx) : Ty → Set where
  Ta : ∀ {t} → Tail Γ t → Code Γ t
  Bind : ∀ {t t1} → Tail Γ t →
    Code (t :: Γ) t1 → Code Γ t1
```

In keeping with the theme of naming things consistently and then working up to their full functionality, Bind is not really a monadic bind, as there is no monad the evaluator is working in (thus the M being parenthesized in the title of these chapters)[1]. It evaluates a Tail, and adds the result of the evaluation to the context of the next code sequence execution. Importantly, this means we have to add the type of the Tail, t, to the type context in the evaluation of the next code sequence, which is why if the context in the current Code sequence is Γ, the nested code sequence's context is t :: Γ.

Now that we can introduce variables, rather than only being able to look up variables in a supplied initial context as in the last two chapters, we can see that

[1]This can be thought of as working in the identity monad, but this is not a useful way of conceptualizing this interpreter.

rather than the All type referencing some static index n in the context — where two references to the same variable have the same All value — they count back to the variable n Binds before the Var referencing the variable. This method of variable binding based on distance from introduction is called a De Bruijn index [dBru72].

Note that the Bind is parameterized by the type of the Tail whose evaluation is to be added to the context (t), and is indexed by the type of the rest of the calculation (t_1), which winds up being the type of the entire code sequence. This means that we have embedded in the types the idea that the type of a code sequence is the type that it returns.

The only real difference between the evaluator of Featherweight MIL and this one — besides adding more functionality for the additional primitive operations and adapting to the changing representation of PrimVals — is that now we must add a codeEvaluator to evaluate the code sequences.

In the case of a Ta, this evaluator simply evaluates the Tail. In the case of a Bind it evaluates the Tail and then evaluates the next code sequence with the result of the Tail evaluation added to the evaluation context, as shown below.

```
infixl 5 _+ᵈ_
_+ᵈ_ : DataVal Word → DataVal Word → DataVal Word
W a +ᵈ W b = W (a + b)

evalAtom : ∀ {Γ t } → Env Γ → Atom Γ t → DataVal t
evalAtom env (Var x) = lookup env x
evalAtom env (Val (Constant Word x)) = W x
evalAtom env (Val (Constant Flag x)) = F x
```

```
evalPrimCall :
  ∀ {t₁ t₂ t₃} → PrimOp t₁ t₂ t₃
  → DataVal t₁ → DataVal t₂ → DataVal t₃
evalPrimCall Madd a₁ a₂ = a₁ +ᵈ a₂
evalPrimCall Mand a₁ a₂ = a₁ ∧ᵈ a₂
evalPrimCall Mmul a₁ a₂ = a₁ *ᵈ a₂

evalTail : ∀ {Γ t} → Tail Γ t → Env Γ → DataVal t
evalTail (Return a) env = evalAtom env a
evalTail (PrimCall primOp a₁ a₂) env =
    evalPrimCall primOp (evalAtom env a₁) (evalAtom env a₂)

evalCode : ∀ {Γ t} → Code Γ t → Env Γ → DataVal t
evalCode (Ta x) env = evalTail x env
evalCode (Bind t c) env =
  let
    tail = evalTail t env
  in
    evalCode c (tail :: env)
```

8.2 Another simplification

The optimization of addition of known constants at compile time in this language is similar to the last. The fact that this optimization only modifies the Tails and does not change the shape of code sequences outside of that means that this optimization is the same as that in last chapter, although this time over addition rather than

multiplication. We can see that the simplification on the Tail level is reflexive as before, in evalPreservesT.

```
simplifyPlusT : ∀ {Γ t } → Tail Γ t → Tail Γ t
simplifyPlusT (PrimCall Madd (Val (Constant Word x)) (Val (Constant Word y)))
  = Return (Val (Constant Word (x + y)))
simplifyPlusT t = t

evalPreservesT : ∀ {Γ t env} → (tail : Tail Γ t)
  → evalTail tail env ≡ evalTail (simplifyPlusT tail) env
evalPreservesT (Return x) = refl
evalPreservesT (PrimCall Madd (Var x) (Var x₁)) = refl
```

With the code sequence type, it is not enough to have an optimization that modifies the tails, we need a way of getting our optimization to those tails. We need one more function to map this optimization function across all the Tails present in a code sequence, which we call simplifyPlusC, where the C stands for Code. For a Ta, this function simply applies simplifyPlusT to the Tail it wraps. In the Bind case, simplifyPlusT is applied to the Tail, and simplifyPlusC is recursively called on the nested Code.

```
simplifyPlusC : ∀ {Γ t} → Code Γ t → Code Γ t
simplifyPlusC (Ta t) = Ta (simplifyPlusT t)
simplifyPlusC (Bind t c) = Bind (simplifyPlusT t) (simplifyPlusC c)
```

Proving that mapping this Tail optimization across a code sequence maintains its semantics is pretty straightforward. In the base case, where the code sequence

consists of a Ta, we can use our evalPreservesT function to prove that the simplification maintains semantics across Tail values. In the case of a Bind, we can use the inductive hypothesis, both from calling evalPreservesT on the tail and evalPreservesC recursively on the nested Code, which show that simplifyPlusT and simplifyPlusC do not change the result of evaluation, respectively. In our equational reasoning steps, we can use our Code inductive hypothesis, ih, followed by a congruence with our Tail inductive hypothesis, to complete this proof. This winds up being a pretty straightforward inductive proof, although we did have to reason about modified values being added to the context.

```
evalPreserves :
  ∀ {Γ t } → (env : Env Γ) → (code : Code Γ t)
  → evalCode code env ≡ evalCode (simplifyPlusC code) env
evalPreserves env (Ta x) = evalPreservesT x
evalPreserves env (Bind x c) =
  let
    ih : evalCode c ((evalTail x env) :: env)
        ≡ evalCode (simplifyPlusC c) ((evalTail x env) :: env)
    ih = evalPreserves ((evalTail x env) :: env) c
    ihT : evalTail x env ≡ evalTail (simplifyPlusT x) env
    ihT = evalPreservesT x
  in
  begin
  evalCode (Bind x c) env
  ≡⟨ refl ⟩
```

```
evalCode c (evalTail x env :: env)
≡⟨ ih ⟩
evalCode (simplifyPlusC c) ((evalTail x env) :: env)
≡⟨ cong (λ y → evalCode (simplifyPlusC c) (y :: env)) ihT ⟩
evalCode (simplifyPlusC c) ((evalTail (simplifyPlusT x) env) :: env)
≡⟨ refl ⟩
evalCode (simplifyPlusC (Bind x c)) env ■
```

8.3 Mountains of monoids

Our previous optimization had the simple effect of adding known constants together at compile time, rather than at runtime. Although this and optimizations like it are important in the optimization of programs, it is clear that there are many more optimizations that do not simply run parts of the program at compile time. For example, consider the existence of algebraic identities. If we have some integer x multiplied by 1, we know the resulting expression is equal to x. In a similar way, some boolean x when logically conjoined with $True$ is equal to x. In fact, integer addition also follows this pattern, as for all $x : Integer$, $x + 0 = x$. These combinations of types and operations are monoids [nLa21e][2], meaning that they follow the monoid laws as shown in Figure 8.1.

We can formalize the idea of a monoid in a record. We need to specify a type A, and a binary operation _<>_ that is closed, so it takes in two operands of type A, and evaluates to another value of type A. We then require an identity element,

[2]Technically for the upcoming optimization we only require them to be unital magmas [nLa21g], that is we do not need the associativity a monoid requires. Monoid is used here because the notation is more familiar to the average Haskell user, and is a commonly used functional programming concept.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash mempty\ <>\ a \equiv a}\ \text{Left identity} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash a\ <>\ mempty \equiv a}\ \text{Right identity}$$

$$\frac{\Gamma \vdash c : A}{\Gamma \vdash (a\ <>\ b)\ <> c\ \equiv a\ <> (\ b\ <> c\ )}\ \text{Associativity}$$

Figure 8.1: The monoid laws

mempty, and proofs that it acts as a left and right identity. We then require a proof of associativity, which completes the definition shown below.

```
record Monoid {l} (A : Set l) : Set l where
  field
    mempty : A
    _<>_  : A → A → A
    assoc    : ∀{a b c} → ((a <> b) <> c) ≡ (a <> (b <> c))
    identityₗ : ∀ {a : A} → (mempty <> a ≡ a)
    identityᵣ : ∀ {a : A} → (a <> mempty ≡ a)
open Monoid
```

For a simple example, we can prove that $+^d$ is a monoid by proving the relevant properties of it. Because $+^d$ is implemented in terms of integer addition, we can use cong W to wrap standard library proofs of the necessary properties in a W constructor, thus making them apply to the DataVal type.

```
+ᵈ-identityˡ : ∀ {a} → W (+ zero) +ᵈ a ≡ a
+ᵈ-identityˡ {W a} = cong W (+-identityˡ a)
```

```
+ᵈ-identityʳ : ∀ {a} → a +ᵈ W (+ zero) ≡ a
+ᵈ-identityʳ {W a} = cong W (+-identityʳ a)

+ᵈ-assoc : {a b c : DataVal Word} → a +ᵈ b +ᵈ c ≡ a +ᵈ (b +ᵈ c)
+ᵈ-assoc {W a} {W b} {W c} = cong W ( +-assoc a b c)
```

We can create a Monoid instance for DataVal Word by creating a record and supplying the relevant proofs, operation, and mempty value.

```
instance
  +ᵈ-monoid : Monoid (DataVal Word)
  +ᵈ-monoid = record
                { mempty = W (+ zero)
                ; _<>_   = _+ᵈ_
                ; assoc     = +ᵈ-assoc
                ; identityₗ = +ᵈ-identityˡ
                ; identityᵣ = +ᵈ-identityʳ
                }
```

These monoids all seem like candidates for the same sort of optimization: if either of the operands of an evalPrimCall function is the mempty value matching the PrimOp, we just convert the PrimCall to a Return of the other operand. As all of our PrimCalls are monoidal, we can optimize all of them in this way.

The optimization described above requires us to compare constants to the mempty value. This does bring up an interesting gap in what we have covered so far: how do we determine whether or not two values are equal? In a traditional programming

language, we could have some equality checking function that would return a boolean value. As we have covered earlier, this would not give evidence of the equality. Our $\equiv$ type does provide evidence of equality, but this does not admit the possibility of failure. A Dec or Decidable[3] data type is parameterized by a proposition P and has two constructors: yes, which contains a proof that P is true, and no, which carries a proof that it is false. For example, we can write a function $_ty\stackrel{?}{=}_$ which compares two types, and $_op\stackrel{?}{=}_$, which compares two PrimOps, as shown below.

$$
\begin{aligned}
&_ty\stackrel{?}{=}_ : (t_1\ t_2 : \mathsf{Ty}) \to \mathsf{Dec}\ (t_1 \equiv t_2)\\
&_ty\stackrel{?}{=}_\ \mathsf{Word}\ \mathsf{Word} = \mathsf{yes\ refl}\\
&_ty\stackrel{?}{=}_\ \mathsf{Word}\ \mathsf{Flag} = \mathsf{no}\ (\lambda\ ())\\
&_ty\stackrel{?}{=}_\ \mathsf{Flag}\ \mathsf{Word} = \mathsf{no}\ (\lambda\ ())\\
&_ty\stackrel{?}{=}_\ \mathsf{Flag}\ \mathsf{Flag} = \mathsf{yes\ refl}
\end{aligned}
$$

$$
\begin{aligned}
&_op\stackrel{?}{=}_ : \forall\ \{t_1\ t_2\ t_3\}\\
&\quad \to (p_1\ p_2 : \mathsf{PrimOp}\ t_1\ t_2\ t_3)\\
&\quad \to \mathsf{Dec}\ (p_1 \equiv p_2)\\
&_op\stackrel{?}{=}_\ \mathsf{Madd}\ \mathsf{Madd} = \mathsf{yes\ refl}\\
&_op\stackrel{?}{=}_\ \mathsf{Madd}\ \mathsf{Mmul} = \mathsf{no}\ (\lambda\ ())\\
&_op\stackrel{?}{=}_\ \mathsf{Mand}\ \mathsf{Mand} = \mathsf{yes\ refl}\\
&_op\stackrel{?}{=}_\ \mathsf{Mmul}\ \mathsf{Madd} = \mathsf{no}\ (\lambda\ ())\\
&_op\stackrel{?}{=}_\ \mathsf{Mmul}\ \mathsf{Mmul} = \mathsf{yes\ refl}
\end{aligned}
$$

Now that we have a type powerful enough to compare values for equality, we can move onto trying to formalize our mapping. There are two main complications with

[3]The reason this is called "decidable" is that it is also a proof that the supplied proposition is decidable, meaning that it can be determined to be either true or false for all input values. As not all propositions are decidable, this is not something we should take for granted.

this strategy. First, PrimCalls do not directly operate on DataVals, they operate on PrimVals, and as such we need some way to convert between these two data types and some proof that these conversions are correct. We also need to map from the PrimOp to the underlying monoidal operation we claim it is evaluated by. We can create another record to encapsulate this, called MapToOperation.

We can start by requiring cmp, an equality function required to compare the PrimVal to the mappend. Obviously the PrimOp, op, is required as well. We need wrap and unwrap functions to convert between constants and DataVals, as well as some proofs of their correctness, for example that unwrap is the left inverse of wrap[4]. We will also need a proof that wrapping an x is equivalent to evaluating it as a constant. Finally we need the function we're claiming this PrimOp is implemented by, fun, and a proof that the PrimCall is evaluated with that function, isEvaluatedWith.

```
record MapToOperation {a : Set} (t : Ty) : Set (lsuc lzero) where
  field
    cmp    : Decidable {A = (constant t)} _≡_
    op     : PrimOp t t t
    wrap   : (constant t)     → DataVal t
    wrapP  : ∀{x Γ env}       → wrap x ≡ evalAtom {Γ} env (Val (Constant t x))
    unwrap : DataVal t        → (constant t)
    wrapUnwrapInv : ∀{x y} → unwrap x ≡ y → x ≡ (wrap y)
    fun    : DataVal t        → DataVal t → DataVal t
    isEvaluatedWith : ∀ {l r : DataVal t} → fun l r ≡ (evalPrimCall op l r )
open MapToOperation
```

[4]Although this does not look like a left-inverse relation at first glance, if we substitute wrap y for x, we get (unwrap (wrap y ≡ y). The form of this is useful for a proof later in the chapter.

Implementing an instance for the Madd operation is relatively straightforward. The op is the Madd operation, cmp is the $_\overset{?}{=}_$ function, the wrap function is simply the W constructor, and the unwrap function just strips a W off. The proof that wrap over a constant is the same as evaluating it is reflexive, as is the proof that this PrimOp is evaluated with $_+^d_$. The proof of wrapUnwrapInv is simple, as unwrapx≡y : x ≡ y and the goal is W x ≡ W y, it can be solved with a simple congruence.

```
instance
  maddMapTo+ᵈ : MapToOperation {ℤ} (Word)
  maddMapTo+ᵈ =
    record
      { op = Madd
      ; cmp = _≟_
      ; wrap = W
      ; wrapP = refl
      ; unwrap = λ { (W x) → x}
      ; wrapUnwrapInv
        = λ { {W x} {n} unwrapx≡y → cong W unwrapx≡y }
      ; fun = (_+ᵈ_)
      ; isEvaluatedWith = refl
      }
```

Rather than a mapping we could have defined the MapToOperation record to have a Monoid record in it — with a proof that associates them — but the design we chose allows us to associate a PrimOp with various properties that may be useful for other

optimizations, for example associating Mmul with a instance of an absorption magma[5] to optimize any Mmul PrimOps with zero operands.

Before we can define our optimizations, it is useful to notice that this optimization should only change the Tail if it is operating over a Monoid. This first requires that the types of both of the operands and the return type are the same as the Tail type. We can write a helper function tysEq, that either returns a tuple of proofs that each of the types is equal to t, or a nothing value.

```
tysEq : (t ty₁ ty₂ ty₃ : Ty) → Maybe ((t ≡ ty₁ × t ≡ ty₂) × t ≡ ty₃)
tysEq t ty₁ ty₂ ty₃
  with t ty≟ ty₁ | t ty≟ ty₂ | t ty≟ ty₃
... | no _  | _      | _      = nothing
... | yes _ | no _   | _      = nothing
... | yes _ | yes _  | no _   = nothing
... | yes p | yes p₁ | yes p₂ = just ((p , p₁) , p₂)
```

Before rushing to write the optimization function, consider that one could optimize and then prove each of these monoid optimizations correct separately, but that does seem like a lot of repeated boilerplate code. It also requires one to write additional optimization functions and proofs whenever we added a primitive operation to the compiler. This is an unideal solution, particularly given that we've encoded the idea of a monoid in a record type. We can leverage this to define our function monoidSimplTail such that it takes some Monoid, a MapToOperation, and a proof that the _<>_ from the Monoid is the same as the fun associated with the operation in MapToOperation.

[5] An absorption magma is some set with a supplied closed binary operation that has a unique member of that set which, when combined with any other member of that set via the binary operation, evaluates to the unique member. For example, Integer multiplication is an absorption magma, because it satisfies the law $\forall x : \mathit{Integer}\ 0 * x \equiv 0 \land x * 0 \equiv 0$ [nLa21a].

This allows us to simplify a Tail with any Monoid in a single function, as shown in the type signature below.

```
monoidSimplTail : ∀ {t t₁ Γ}
  → (m : Monoid (DataVal t))
  → (mToOp : MapToOperation {constant t} t)
  → ((l r : DataVal t) → (fun mToOp l r ≡ (_<>_) m l r))
  → Tail Γ t₁
  → Tail Γ t₁
```

Although one has to optimize the cases when the left or right operand are mempty, these cases are symmetrical and therefore the same optimization, but flipped. The proof also follows this symmetry and duplication. Because of this symmetry, we will only show the left operand case. In the case that the Tail is a Return, the optimization should just evaluate to its input. If the input is a PrimCall, the first check that is required is if the types all match. If they do, we need to make sure that the PrimOp matches the op in the MapToOperation record. If it does, we can case split on the left operand, and in the case that it is a Val, we can compare it to mempty with the cmp function supplied by mToOp. If the left operand, l, is equal to mempty, we can replace this PrimCall with a Return of the right operand. An analogous procedure can be used to optimize for then l is not a Val.

```
monoidSimplTail monoid mToOp _ (Return x) = Return x
monoidSimplTail {t} monoid mToOp _ (PrimCall {pt₁} {pt₂} {pt₃} pcOp l r)
  with tysEq t pt₁ pt₂ pt₃
... | nothing = (PrimCall pcOp l r)
... | just ((p , p₁) , p₂) rewrite p | p₁ | p₂
```

```
        with pcOp op≟ (op mToOp)
... | no ¬op≡pcOp = (PrimCall pcOp l r)
... | yes op≡pcOp with l
... | Val (Constant _ l′) with cmp mToOp l′ (unwrap mToOp (mempty monoid))
... | no ¬x₃≡mempty = (PrimCall pcOp l r)
... | yes x₃≡mempty  = Return r
```

...

Rather than writing a bunch of specialized optimizations, our function is able to optimize away any operations with a mempty at compile time, as long as we've provided the proper data types!

In order to prove this optimization correct, we must in turn prove that this optimization maintains semantics over all Tails, Monoids, and MapToOperations. As in our earlier tail optimizations, we need to prove that $eval\ (f\ t) \equiv eval\ t$ where f is the optimization function, as shown in the type signature below.

```
monoidSimplTailPreserves : ∀ {t Γ env}
  → (tail : Tail Γ t)
  → (m : Monoid (DataVal t))
  → (mToOp : MapToOperation {constant t} t)
  → (mapping : (l r : DataVal t) → (fun mToOp l r ≡ (_<>_) m l r))
  → evalTail (monoidSimplTail m mToOp mapping tail) env ≡ evalTail tail env
```

As we can see below, the case when Tail is a Return is reflexive, as our optimization does not change the Tail in this case. In the case where the Tail is a PrimCall, we

can replicate the series of with statements to split the proofs into paths where the optimization is applied, and ones where the original Tail is simply returned. In the cases where the Tail is not modified, that is where the operands, the Tail, and the Primcall return type do not match, when the PrimOp does not match the op of the mToOp, or when the Val does not equal mempty, the proof is simply reflexive.

```
monoidSimplTailPreserves (Return x) m mToOp match = refl
monoidSimplTailPreserves
  {t} {Γ} {env} (PrimCall {pt₁} {pt₂} {pt₃} pcOp l r) m mToOp match
  with tysEq t pt₁ pt₂ pt₃
... | nothing = refl
... | just ((p , p₁) , p₂) rewrite p | p₁ | p₂ with pcOp op≟ (op mToOp)
... | no ¬p = refl
... | yes x≡op with l
... | Val (Constant _ l′) with cmp mToOp l′ (unwrap mToOp (mempty m))
... | no ¬x₂′≡mempty = refl
... | yes x₂′≡mempty =
```

In the cases where our optimization actually changes the PrimCall to a Return, we need to show that evaluating Return x is equivalent to evaluating the PrimCall. This reduces to proving that

```
    evalAtom env r
≡
evalPrimCall pcOp ( evalAtom env( Val ( Constant pt₂ l′)))( evalAtom env r)
```

We have introduced let bindings for mempty, _<>_, op, fun, and wrap so we do not have to add the Monoid or MapToOperation as a parameter to them, in order to make the proof easier to read. Working from the top to the bottom, we can use the identity_l to expand our initial evalAtom to be mappended to the mempty value. We can then use our match proof to convert this to an application of fun. We can use our proof x_2'≡mempty, along with our wrapUnwrapInv function to convert the mempty to (wrap l′), and then the wrapP proof to further convert that to l′ being evaluated as a constant. The isEvaluatedWith lemma allows us to convert the fun statement to the evaluation of a PrimCall with the op from our MapToOp, and finally we can complete the proof with an application of our x≡op lemma by converting op to pcOp.

```
begin
  evalAtom env r
≡⟨ sym (identityₗ m) ⟩
  (mempty <> evalAtom env r)
≡⟨ sym (match mempty (evalAtom env r)) ⟩
  fun mempty (evalAtom env r)
≡⟨ cong
      (λ a → fun a (evalAtom env r))
      (wrapUnwrapInv mToOp (sym x₂′≡mempty)) ⟩
  fun (wrap l′) (evalAtom env r)
≡⟨ cong
      (λ a → fun a (evalAtom env r))
      (wrapP mToOp {l′} {Γ} {env}) ⟩
  fun (evalAtom env (Val (Constant pt₂ l′))) (evalAtom env r)
```

```
  ≡⟨ isEvaluatedWith mToOp ⟩
    evalPrimCall
      op
      (evalAtom env (Val (Constant pt₂ l′)))
      (evalAtom env r)
  ≡⟨ cong
        (λ a → evalPrimCall a (evalAtom env (Val (Constant pt₂ l′)))
        (evalAtom env r)) (sym x≡op) ⟩
    evalPrimCall
      pcOp
      (evalAtom env (Val (Constant pt₂ l′)))
      (evalAtom env r) ∎
```

So now that we have an optimization over Tails and a proof of correctness, we can go on to write a function to optimize code sequences with this optimization. In order to do this, we need a way to map some optimization over all the Tails in a code sequence. This is relatively simple to accomplish, as in the Ta case one just applies the optimization to the Tail and wrap the result in a Ta. In the Bind case, we apply the tail optimization to the Tail, and then recursively call on the nested code.

```
monoidSimplCode : ∀ {t Γ}
  → (m : Monoid (DataVal t))
  → (mToOp : MapToOperation {constant t} t)
  → ((l r : DataVal t) → MapToOperation.fun mToOp l r ≡ (_<>_) m l r )
  → Code Γ t → Code Γ t
monoidSimplCode monoid mToOp match (Ta tail)
```

```
    = Ta (monoidSimplTail monoid mToOp match tail)
  monoidSimplCode monoid mToOp match (Bind tail code) =
    let
      tail' = monoidSimplTail monoid mToOp match tail
      code' = monoidSimplCode monoid mToOp match code
    in
      Bind tail' code'
```

If this seems like an overly-specific way to apply the optimization, that's because it is! Our simplifyPlusC function looks almost exactly like our monoidSimplCode function, but with the optimization function applied to each Tail differing. This is a hint that maybe there is a way to generalize this, so that we can apply any optimization function across a code sequence. In fact, this is an example of a functor[6], and so we can write a function that applies some function to every Tail in the code block. The function tailMap below is exactly such a generalization. We can see from the type annotation that this takes a function f as input, which takes in its own type, context, and corresponding Tail, and returns a Tail of a matching type. tailMap then takes a code sequence as input, and applies the function f to each Tail in the code sequence as our previous more specific code optimization functions did.

```
  tailMap :
    ∀ {t₁ Γ}
    → (f :
```

[6]The code sequence is isomorphic to a non-empty list of Tails, and so we can implement our map function almost identically to that of a list. If we had more information in our code block, for example a variable name each variable was bound to, this would be an example of a lens. This is because it would be applying some update function to a part of the nested data structure, rather than all the members of it.

$$\{t : \mathsf{Ty}\}$$
$$\to \{\Gamma' : \mathsf{Ctx}\}$$
$$\to \mathsf{Tail}\ \Gamma'\ t$$
$$\to \mathsf{Tail}\ \Gamma'\ t)$$
$$\to \mathsf{Code}\ \Gamma\ t_1$$
$$\to \mathsf{Code}\ \Gamma\ t_1$$
$$\mathsf{tailMap}\ f\ (\mathsf{Ta}\ tail) = \mathsf{Ta}\ (f\ tail)$$
$$\mathsf{tailMap}\ f\ (\mathsf{Bind}\ tail\ code) = \mathsf{Bind}\ (f\ tail)\ (\mathsf{tailMap}\ f\ code)$$

To prove this higher order optimization function correct, we need a higher order proof to go with it. The type signature gives insight into what this must accomplish. It is parameterized by a function f of the same type of f in tailMap. It takes an additional argument t≡, which takes a context, a type of the tail, an environment and a Tail parameterized by these, and evaluates to a proof that applying f to tail does not change the result of evaluation.

$$\mathsf{tailMap}{\equiv} :$$
$$\forall\ \{t_2\ \Gamma\}$$
$$\to \{env : \mathsf{Env}\ \Gamma\}$$
$$\to (c : \mathsf{Code}\ \Gamma\ t_2)$$
$$\to (f\quad :$$
$$\{t : \mathsf{Ty}\}$$
$$\to \{\Gamma' : \mathsf{Ctx}\}$$
$$\to \mathsf{Tail}\ \Gamma'\ t$$
$$\to \mathsf{Tail}\ \Gamma'\ t)$$
$$\to (t{\equiv} :$$

```
            ({Δ : Ctx}
            → {t₁ : Ty}
            → {env' : Env Δ}
            → (tail : Tail Δ t₁ )
            → (evalTail {Δ} tail env' ≡ evalTail {Δ} {t₁} (f tail) env')))
        → (evalCode c env ≡ evalCode {Γ} {t₂} (tailMap f c) env)
```

The Ta case is quite easy to prove, simply apply the proof passed in as a parameter to the tail. The Bind case is slightly more complicated. The proof that applying the optimization to the Tail can be created by applying the proof f≡ to the Tail value. The inductive hypothesis ih shows that evaluating the nested code sequence c is the same as the optimized code sequence tailMap f c. Given these lemmas, we can prove our proposition with two simple equational reasoning steps.

```
tailMap≡ (Ta x) f f≡ = f≡ x
tailMap≡ {t₂} {Γ} {env} (Bind {tₜ} {t_c} tail code) f f≡ =
  let
    Γ' = tₜ :: Γ
    tV = evalTail {Γ} tail env
    iht = f≡ tail
    ih  = tailMap≡ {t_c} {Γ'} {(tV :: env)} code f f≡
  in
    begin
      evalCode code (evalTail tail env :: env)
    ≡⟨ ih ⟩
    evalCode (tailMap f code) (evalTail tail env :: env)
```

```
  ≡⟨ cong (λ y → evalCode (tailMap f code) (y :: env)) iht ⟩
  evalCode (tailMap f code) (evalTail (f tail) env :: env) ■
```

We have shown that any optimization that can be applied to Tails and not change their semantics can be mapped across all the Tails in a Code sequence without changing the result of its evaluation. With our monoid optimization, we have many such possible optimizations. It is a relatively trivial matter to show that we can compose any two of these tail optimizations, and that the act of composing them does not introduce a semantic divergence.

We can define a function tailEqCompose, which takes a Tail, tail_1, and functions f g, the optimization functions. It is then parameterized by proofs that f and g do not change the semantics of the Tail evaluation. We can use these proofs with two equational reasoning steps to show that the composition of optimization functions f and g do not change the evaluation of the Tail they are applied to.

```
tailEqCompose :
  ∀ {t Γ env}
  {tail₁ : Tail Γ t}
  → (f g : Tail Γ t → Tail Γ t)
  → ((tail₂ : Tail Γ t) → evalTail tail₂ env ≡ evalTail (f tail₂) env )
  → ((tail₂ : Tail Γ t) → evalTail tail₂ env ≡ evalTail (g tail₂) env )
  → evalTail tail₁ env ≡ evalTail (f (g tail₁)) env
tailEqCompose {env = env} {tail₁ = tail} f g ef≡e eg≡e =
  begin
  evalTail tail env
  ≡⟨ eg≡e tail ⟩
```

evalTail $(g\ tail)\ env$

$\equiv\langle\ ef{\equiv}e\ (g\ tail)\ \rangle$

evalTail $(f\ (g\ tail))\ env$ ■

In this chapter we have added the ability to introduce variables, defined optimizations over monoids, and used higher level proofs to prove compositions of optimizations correct. In the next chapter we can increase the complexity of our interpreter once again, this time by adding an effect to our evaluator.

Chapter 9

Writer MIL

In the last chapter, we extended the language with the ability to introduce variables, which is certainly important. In this next language, we can finally remove the parenthesis from (M)IL, and introduce a monadic effect! It is important that we implement the interpreter in some monad, even a simple one, as two of the optimizations in MIL depend on the monadic structure of the language. The monad we are choosing to extend the language with is the Writer monad introduced in Chapter 5, as a standard out equivalent seems like an essential part of a program evaluator. The execution of our interpreter being in this monad means it does not just evaluate pure code, but amasses a list of outputs in addition to a return value.

9.1 Abstract syntax and interpreter

We have added a Unit type to our interpreter, as all Tails must have a type and return a value. Other than that, the types are the same as in the last interpreter.

```
data Ty : Set where
  Word : Ty
  Flag : Ty
```

```
  Unit : Ty
```

We have also added a U DataVal which is parameterized by the top value, tt. This the same as the Haskell () in Chapter 4.

```
data DataVal : Ty → Set where
  W : ℤ → DataVal Word
  F : Bool → DataVal Flag
  U : ⊤ → DataVal Unit

constant : Ty → Set
constant Unit = ⊤
constant Word = ℤ
constant Flag = Bool
```

One additional constructor has been added to Tail: Output. It is parameterized by an Atom, and is of type Unit. The Atom represents the value to be converted to a string and inserted into the output list. The Code data type has not changed since the last chapter, and so we will not be repeating it here.

```
data Tail (Γ : Ctx) : Ty → Set where
  Return : ∀ {t} → Atom Γ t → Tail Γ t
  PrimCall : ∀ {t1 t2 t3} → PrimOp t1 t2 t3
    → Atom Γ t1 → Atom Γ t2 → Tail Γ t3
  Output : ∀{t} → Atom Γ t → Tail Γ Unit
```

Now that we have finished defining the type of abstract syntax trees of the language, we can move on to implementing the evaluator. The evalAtom function has a case

added for the Unit type, but is identical otherwise. The evalPrimCall function is unchanged, as there is no way to evaluate a Unit value in this language.

```
evalAtom : ∀ {Γ t } → Env Γ → Atom Γ t → DataVal t
evalAtom env (Val (Constant Unit x)) = U x
evalAtom env (Var x) = lookup env x
evalAtom env (Val (Constant Word x)) = W x
evalAtom env (Val (Constant Flag x)) = F x
```

Given that this evaluator operates in a Writer monad, used as an analogue to standard out, we need a way to covert our DataVals to Strings before they can be supplied to this output list. We can implement this conversion in showdv, as shown below.

```
showdv : ∀{t} → DataVal t → String
showdv (W n)    = show n
showdv (F false) = "False"
showdv (F true) = "True"
showdv (U x)    = "()"
```

The evalTail function needs a case added to it for the Output constructor. In the Output case, we evaluate the supplied Atom to a DataVal, convert this to a String, and supply a singleton list of that string as the log. The return value of the Writer is U tt. In the other cases the result of evaluating the atom of a Return or a PrimCall, is injected into a Writer with return.

```
evalTail : ∀ {Γ t} → Tail Γ t → Env Γ → Writer (DataVal t)
evalTail (Return a) env = return (evalAtom env a)
```

```
evalTail (PrimCall primOp a₁ a₂) env =
  return (evalPrimCall primOp (evalAtom env a₁) (evalAtom env a₂))
evalTail (Output a) env with evalAtom env a
... | x = ( U tt ) ,′ showdv x :: []
```

The evalCode function is only altered from Featherweight MIL in that, in the Bind case, the result of evaluating the Tail is bound to the evaluation of the next code sequence by the ≫= function. This results in our Bind constructor being implemented as a legitimate monadic bind!

```
evalCode : ∀ {Γ t} → Code Γ t → Env Γ → Writer (DataVal t)
evalCode (Ta tail) env = evalTail tail env
evalCode (Bind tail code) env =
  (evalTail tail env) ≫= (λ a → evalCode code (a :: env))
```

Now that we have defined our data types and interpreter, we will move onto defining our monadic optimization functions in the next section.

9.2 Right monad law

We had mentioned in the beginning of this chapter that there are optimizations that depend on the evaluator being in some monad. One of these is the use of the right monad law to optimize a Return away in the case that it is wrapping the result of the previous ≫=. We proved that this holds for the Writer monad in Chapter 5, and as our evaluator is written in terms of return and ≫= we should be able to optimize code sequences in Writer MIL in the same way.

This is a simple optimization to write. In the case of a Ta, it should act as an

identity. In the case of a Bind it should only modify the code sequence in the case that the Tail is a Return of the most recently introduced variable in the context, and then it should change the Bind to a Ta of the most recently bound Tail. Finally, in the case of a Bind other than that special case, we leave the Tail untouched and recursively call the rightMonadLaw on the nested code sequence.

```
rightMonadLaw : ∀ {Γ t} → Code Γ t → Code Γ t
rightMonadLaw (Ta x) = Ta x
rightMonadLaw (Bind t (Ta (Return (Var (here refl))))) = Ta t
rightMonadLaw (Bind t c) = Bind t (rightMonadLaw c)
```

In order to prove that the rightMonadLaw function does not change the result of evaluation, we need to prove one straightforward lemma: that evaluating a variable in some environment the same as looking it up in that environment. A simple proof by cases is sufficient to show this to be true.

```
evalVal≡lookup : ∀ {t Γ}
  → {env : Env Γ}
  → {x : t ∈ Γ}
  → (evalAtom {Γ} env (Var x)) ≡ lookup env x
evalVal≡lookup {Word} {Γ} {env} {x} = refl
evalVal≡lookup {Flag} {Γ} {env} {x} = refl
evalVal≡lookup {Unit} {Γ} {env} {x} = refl
```

Moving onto the main proof, we start by case splitting over the Code parameter. In the rightMonadLaw optimization, the Code is only modified in two cases: the inductive case of Bind, where the nested code sequence is also a Bind, and the specific case

where the Bind has a nested code sequence that returns the last introduced variable: Ta (Return (Var (here refl))). All of the other cases are reflexive, as shown below.

```
rMLPreservesML : ∀ {Γ t env} → (code : Code Γ t)
  → evalCode code env ≡ evalCode (rightMonadLaw code) env
rMLPreservesML (Ta x) = refl
rMLPreservesML (Bind x (Ta (Return (Val x₁)))) = refl
rMLPreservesML (Bind x (Ta (PrimCall x₁ x₂ x₃))) = refl
rMLPreservesML (Bind x (Ta (Output x₁))) = refl
rMLPreservesML (Bind x (Ta (Return (Var (there _))))) = refl
```

...

For the non-reflexive cases, we can begin with the case where the nested code sequence returns the most recently introduced variable. If we begin with the left-hand side of the equality we are trying to prove, evalCode (Bind x Ta (Return (Var (here refl))))) env, we can step through the evaluation of this term in our equational reasoning block, until we reduce it to (evalTail x env ≫= return). This is exactly the form our right monad law proof, rML, took, and so we can use rML show that (evalTail x env ≫= return) is equal to evalTail x env. From this, we can show through two simple steps that it is equal to evalCode (rightMonadLaw Bind x Ta (Return (Var (here refl))))) env, completing our proof of this case.

```
rMLPreservesML {_} {t} {env} (Bind x (Ta (Return (Var (here refl))))) =
  begin
    evalCode (Bind x (Ta (Return (Var (here refl))))) env
  ≡⟨ refl ⟩
```

```
  evalTail x env
    >>= (λ a → evalCode (Ta (Return (Var (here refl)))) (a :: env))
≡⟨ refl ⟩
  evalTail x env
    >>= (λ a → evalTail (Return (Var (here refl))) (a :: env))
≡⟨ refl ⟩
  (evalTail x env)
    >>= (λ a → return (evalAtom (a :: env) (Var (here refl))))
≡⟨ cong (λ b → (evalTail x env)
              >>= (λ a → return b )) evalVal≡lookup ⟩
  (evalTail x env)
    >>= (λ a → return (lookup (a :: env) (here refl)) )
≡⟨ refl ⟩
  (evalTail x env) >>= (λ a → return  a)
≡⟨ refl ⟩
  ( evalTail x env >>= return)
≡⟨ rML (evalTail x env) ⟩
  evalTail x env
≡⟨ refl ⟩
  evalCode (Ta x) env
≡⟨ refl ⟩
  evalCode (rightMonadLaw (Bind x (Ta (Return (Var (here refl)))))) env ■
```

The final case we must consider is the inductive case, where the Code being optimized is a Bind and its nested code sequence is also a Bind. In this case, we can

simply step through the evaluation as above until we reach an explicit ≫= and then apply the inductive hypothesis, followed by reversing the evaluation steps as we did in the last proof.

```
rMLPreservesML {_} {_} {env} (Bind x (Bind x₁ c)) =
  let
    ih = rMLPreservesML (Bind x₁ c)
  in
    begin
      evalCode (Bind x (Bind x₁ c)) env
    ≡⟨ refl ⟩
      evalTail x env
        ≫= (λ a → evalCode (Bind x₁ c) (a :: env))
    ≡⟨ cong (λ z → (evalTail x env) ≫= (λ a → z) ) ih ⟩
      evalTail x env
        ≫= (λ a → evalCode (rightMonadLaw (Bind x₁ c)) (a :: env))
    ≡⟨ refl ⟩
       evalCode (rightMonadLaw (Bind x (Bind x₁ c))) env ■
```

With this, we have proven that our first monadic optimization correctly optimizes code sequences under very specific conditions. Now that we have this relatively simple proof out of the way, we can move onto the harder and more powerful of the two optimizations.

9.3 Left monad law

The other, more involved monad law optimization is that of the left monad law. In Chapter 5, we define the law in the form return a $\gg=$ c $\equiv$ c a. Intuitively, when a return injects a into some monad, and then $\gg=$ extracts it, these two operations undo each-others effects, and so the bind becomes simple function application. There is a problem here, however: MIL does not have pure function application! Everything is in the language of binds, as we are in the MIL execution monad. An alternative but equivalent representation (as shown in Chapter 2) is $x \leftarrow return\, a;\ c \equiv [a/x]c$, where the returned value is substituted for the bound variable in c. This implies that we need to write a substitution function, where we can substitute one variable for another in the rest of the code sequence.

This brings with it significant difficulties, as substitution is already notoriously difficult to get right. Our implementation of variable lookups as All values representing paths to the introduction of the variable introduces another problem: if we remove a Bind statement, we must change the lookup values for every variable below the removal, as one segment of the path is removed in every variable that is not to be substituted for the variable. Conveniently, similar representations of this have already been formalized in Agda [WKS20], which is what we will base our implementation of this method of substitution on.

First, we must consider the problem of contextual inconsistency between the substituted and non-substituted Atoms. By removing a Bind from a code sequence, we alter the context for everything below that removal. This requires us to implement *extension*: that given some mapping from variables in one context to another, we can extend both contexts with another variable and the mapping is maintained.

```
extend : ∀ {Γ Δ : Ctx}
  → (∀ {a } → (a ∈ Γ) → (a ∈ Δ))
  → (∀ {a b} → (a ∈ (b :: Γ) → a ∈ (b :: Δ)))
extend m (here px) = here px
extend m (there x) = there (m x)
```

Next, we must implement *renaming*, where given a mapping from variables in one context to another we can map our expressions from the first context to the second. Given the hierarchical nature of the MIL language design, this means implementing renaming for Atom, Tail, and Code data types. The basic strategy for such renaming is simple. In our renameAtom function, we apply the supplied mapping to Vars, and act as an identity for Vals. The renameTail function simply calls renameAtom with its supplied mapping. Finally, in renameCode we either call renameTail on the Ta case, or call the renameTail function on the Tail, and then call renameCode on the nested code sequence, with the mapping m extended with extend to account for the introduction of the Tail into the environment.

```
renameAtom : ∀{Γ Δ}
  → (∀ {a} → (a ∈ Γ) → (a ∈ Δ))
  → (∀ {a} → (val : Atom Γ a) → (Atom Δ a))
renameAtom m (Var x) = Var (m x)
renameAtom m (Val x) = Val x

renameTail : ∀{Γ Δ}
  → (∀ {a} → (a ∈ Γ) → (a ∈ Δ))
  → (∀ {a} → (val : Tail Γ a) → (Tail Δ a))
```

```
renameTail m (Return x)  = Return (renameAtom m x)
renameTail m (PrimCall op x₁ x₂) = PrimCall
                                      op
                                      (renameAtom m x₁)
                                      (renameAtom m x₂)
renameTail m (Output x) = Output (renameAtom m x)

renameCode : ∀{Γ Δ}
  → (∀ {a} → (a ∈ Γ) → (a ∈ Δ))
  → (∀ {a} → (val : Code Γ a) → (Code Δ a))
renameCode m (Ta t) = Ta (renameTail m t)
renameCode m (Bind t c) = Bind (renameTail m t) (renameCode (extend m) c)
```

Finally, we can begin to implement substitution itself. Our substitution functions, in keeping with our renaming functions, are split up into three different functions: substituteAtom, substituteTail, and substituteCode, all of the form that given a mapping from All values in one context to terms in another, we can map a value from the first context to the second. Our substituteAtom function resembles renameAtom: if the Atom is a Var it applies the mapping to the lookup in it, however, in the case of a Val, it acts as an identity.

```
substituteAtom : ∀ {Γ Δ}
  → (∀ {t} → (t ∈ Γ) → Atom Δ t)
  → (∀ {t} → Atom Γ t → Atom Δ t)
substituteAtom m (Var x) = m x
substituteAtom m (Val x) = Val x
```

Our substituteTail function acts like a lens, applying our substituteAtom function with the supplied mapping to each Atom in the Tail.

```
substituteTail : ∀ {Γ Δ}
  → (∀ {t} → (t ∈ Γ) → Atom Δ t)
  → (∀ {t} → Tail Γ t → Tail Δ t)
substituteTail m (Return x)
  = Return (substituteAtom m x)
substituteTail m (PrimCall op x₁ x₂)
  = PrimCall op (substituteAtom m x₁)(substituteAtom m x₂)
substituteTail m (Output x)
  = Output (substituteAtom m x)
```

Before we move onto the substituteCode function, recall that in our renameCode function, we needed a way to extend our mapping by increasing the index of the All value by one. We must develop an analogous extension function that extends a mapping from All values to Atoms, which can be implemented by using the renameAtom function to extend the context of the Atom.

```
extendSubst : ∀ {Γ Δ}
  → (∀ {t} → (t ∈ Γ) → Atom Δ t)
  → (∀ {t₁ t₂} → (t₁ ∈ (t₂ :: Γ)) → (Atom (t₂ :: Δ) t₁))
extendSubst m (here px) = Var (here px)
extendSubst m (there a) = renameAtom there (m a)
```

Up until this point, we have referred to a "supplied mapping", which would be applied to all the Atoms in some data structure. We can implement the mapping in

our function m'[1] by supplying an Atom we wish to substitute, as well as an All value, to indicate the index where the supplied Atom should be substituted out. In the case that the All is here, this means that m' substitutes the supplied Atom, otherwise the Atom representing a variable has one there removed from it, to compensate for the Bind removed in the optimization.

```
m' : ∀ {t t₁ Γ} → Atom Γ t₁ → t ∈ ( t₁ :: Γ) → (Atom Γ t)
m' a (here refl) = a
m' a (there i) = Var i
```

With these functions out of the way, we can finally implement our substituteCode function. This implementation is relatively straightforward: in the Ta case, we apply substituteTail to its Tail, whereas in the Bind case we take the additional step of a recursive call on c, extending the mapping with extendSubst.

```
substituteCode : ∀ {Γ Δ}
  → (∀ {t} → (t ∈ Γ) → Atom Δ t)
  → (∀ {t} → Code Γ t → Code Δ t)
substituteCode m (Ta t)
  = Ta (substituteTail m t)
substituteCode m (Bind t c)
  = Bind (substituteTail m t) (substituteCode (extendSubst m) c)
```

Now that we have our substitution functions implemented, we can implement our optimization, lmlSimplify. In the Ta case, it acts as an identity. In the (Bind (Return x) c) case, we want to substitute the returned Atom, x, through the rest of the code

[1]The prime is used to avoid shadowing of the m variable used to represent a mapping variable in the substitution and renaming functions.

sequence, c. In the remaining cases, we call lmlSimplify recursively on the nested code segment.

```
lmlSimplify : ∀ {t Γ} → Code Γ t → Code Γ t
lmlSimplify (Ta x) = Ta x
lmlSimplify (Bind (Return x) c)
  = substituteCode (m' x) c
lmlSimplify (Bind (PrimCall op x₁ x₂) c)
  = (Bind (PrimCall op x₁ x₂) (lmlSimplify c))
lmlSimplify (Bind (Output x) c)
  = (Bind (Output x) (lmlSimplify c))
```

To illustrate that this function works as intended, we apply it to a short code sequence that converts a DataVal Word representing a Celsius value to one representing a Kelvin value by adding 273 to it. Note that the optimized version both substitutes the returned constant for the variable referencing it in the Primcall and decreases the index of the other Var to compensate for the removal of the Bind.

```
_ : lmlSimplify
    (Bind (Return (Val (Constant Word (+ 273))))
    (Ta (PrimCall Madd (Var (there (here refl))) (Var (here refl)))))
      ≡
    Ta (PrimCall Madd (Var (here refl)) (Val (Constant Word (+ 273))))
_ = refl
```

In other chapters of this thesis, this is the point where we would prove this optimization correct. However, there are two important considerations that, when

combined, make this a misallocation of our space and attention. First, proving substitution correct is a famously difficult problem, and as such, the proofs involved would take up a huge proportion of the remaining thesis. Second, in order to account for additional features — such as imports — a full implementation of MIL will rely on a different representation of variables, and as such, the proof would not even generalize. Instead, let us end this chapter with an understanding of the monadic optimizations themselves, and spend our attention on our final, and more advanced, language.

Chapter 10

Block MIL

Although we've expanded the interpreter with a number of features throughout the previous chapters, it has previously been limited to executing a code sequence and returning a value at the end. One feature that has been conspicuously missing is block calls: the ability to call other code sequences with arguments. In turn, this has prevented us from implementing if statements, as in MIL they decide which of two possible blocks to call. Let us introduce Block MIL, a further extension of our subset of MIL, now with block calls and if statements added, among other features.

10.1 Abstract data types

The Ty and DataVal definitions have remained almost the same as in Writer MIL, but we have removed the Unit type and corresponding DataVal, for reasons that will become clear in the coming paragraphs. We have updated the PrimOp definition and added some additional data types to capture code blocks and their types.

Up until this point, we have been writing lists by using the :: constructor. This worked well, as generally we only used lists as contexts, in which case we were either adding or removing the head of the list, via using :: or pattern matching respectively. In this chapter we will often be constructing lists from scratch, and as such the ::

constructor gets somewhat unwieldy[1]. We can use pattern synonyms [AAC+21d] — a kind of syntactic sugar definition — to make construction of our lists more visually similar to Haskell's notation, as shown below[2].

```
pattern [_]// z = z :: []
pattern [_,//_]// y z = y :: z :: []
pattern [_,//_,//_]// x y z = x :: y :: z :: []
pattern [_,//_,//_,//_]// w x y z = w :: x :: y :: z :: []
pattern [_,//_,//_,//_,//_]// v w x y z = v :: w :: x :: y :: z :: []
pattern [_,//_,//_,//_,//_,//_]// u v w x y z = u :: v :: w :: x :: y :: z :: []
pattern [_,//_,//_,//_,//_,//_,//_]// t u v w x y z = t :: u :: v :: w :: x :: y :: z :: []
```

Up until now, PrimOps had two inputs and an output, all of the same type. We had claimed that there were ways to make this more expressive in Chapter 7, and so let us take this opportunity to make the implementation of our PrimOps expressive enough to capture any number of inputs, of varying types. We can see below that rather than having each PrimOp indexed by three types as we did before —representing the two input and one output types— it is instead indexed by a list of input types, and an output type. We have also implemented additional PrimOps, those of DataVal Word equality and subtraction.

```
data PrimOp : List Ty → Ty → Set where
  Madd : PrimOp [ Word ,// Word ]// Word
  Msub : PrimOp [ Word ,// Word ]// Word
```

[1]This chapter originally used :: notation, but it was difficult to parse, even for the author. For the sake of the readers, this was changed to a more easily visually parsed format.

[2]Our implementation of this pattern is based on a pattern used for the same purpose in Programming Language Foundations in Agda [WKS20]

```
Mand : PrimOp [ Flag ,// Flag ]// Flag
Mmul : PrimOp [ Word ,// Word ]// Word
MisZero : PrimOp [ Word ]// Flag
Mweq : PrimOp [ Word ,// Word ]// Flag
```

To define a block type, it is necessary to reflect for a moment on the limitations of our earlier languages, and the expressiveness of the actual MIL specification when it comes to code blocks. The actual specification allows multiple arguments to be passed into a code block, which is standard for most programming languages. Each code block, however, returns a list of values, possibly of different types. This change in return type is why we eliminated the unit type, as we can instead return an empty list.

As list types seem like a relatively important feature of MIL, let us extend our Block MIL interpreter with them! To do so, first we must define a type DefType, which consists of one constructor: BlockType. This is parameterized with two lists of types, representing the input and output types respectively. We also need to define a context to look code blocks up in, BCtx, much like our type context Ctx. BCtx is defined as a list of DefTypes.

```
data DefType : Set where
  BlockType : List Ty → List Ty → DefType

BCtx = List DefType
```

The Tail type has two differences from that in Writer MIL: first it is parameterized with a block context Δ, and it is indexed by a list of output types, rather than a single output type. To update the constructors present in Writer Mil, we update the Atom

Γ t, a single Atom, to an All expression representing a list of input Atoms. A param function is used as an alias for an All type where the type of the Atoms corresponds to the list of types supplied as parameters to other parameters of the constructor, or indexing the Tail as an output type.

```
params : Ctx → List Ty → Set
params Γ tys = All (Atom Γ) tys
```

In the case of Return, the t_{out} parameterizes the parameter list and indexes the Tail as its type. In the PrimCall case, t_{in} parameterizes PrimOp as its input types, as well as the list of arguments passed as inputs to it. The t_{out} is the type of the output of the PrimOp, and as such is the single member of the output type list for the Tail. The Output constructor simply requires us to convert the Atom to a list of Atoms and to construct a singleton list of the Unit type as the Tail type. Finally, the BlockCall constructor takes an Any value to index into the BCtx to retrieve a block of type BlockType t_{in} t_{out} in its context. It then takes a param that is parameterized by t_{in} as well, as the PrimCall case does, and then indexes the Tail with t_{out} as its type.

```
data Tail (Γ : Ctx ) ( Δ : BCtx) : List Ty → Set where
  Return    : ∀ {t_out} → params Γ t_out → Tail Γ Δ t_out
  PrimCall : ∀ {t_in t_out } → PrimOp t_in t_out
                                 → params Γ t_in
                                 → Tail Γ Δ ([ t_out ]′′)
  Output    : ∀ {t} → params Γ t → Tail Γ Δ []
  BlockCall : ∀ { t_in t_out } → (BlockType t_in t_out) ∈ Δ
                                   → params Γ t_in → Tail Γ Δ t_out
```

The Code type has the same updates from its Writer MIL definition as the Tail does: an added BCtx, and the type context additions are modified from single types to lists of types to account for the update to multiple return values. The Ta constructor has not actually changed syntactically, but the type ts is now a list of types, rather than a single type. The Bind constructor, as a result of the change to lists of types, requires the return types of the Tail to be concatenated to the front of, rather than consed onto the front of, the context that parameterizes the Code. Finally, we have a new language construct, the If constructor. This is parameterized by a single Atom, representing the conditional variable, and then two pairs of code blocks and matching parameters, with the same return types. Recall from Chapter 2 that MIL only allows if statements at the end of a block, and only allows them to call other blocks as resulting statements. This contrasts with a language like Haskell, where an if statement can directly evaluate to a value, for example `if True then 1 else 0`.

```
data Code (Γ : Ctx) ( Δ : BCtx) : List Ty → Set where
  Ta : ∀ {ts} → Tail Γ Δ ts → Code Γ Δ ts
  Bind : ∀ {t t1} → Tail Γ Δ t → Code (t ++ Γ) Δ t1 → Code Γ Δ t1
  If : ∀ {t_in1 t_in2 t_out} → Atom Γ Flag
      → ((BlockType t_in1 t_out) ∈ Δ × params Γ t_in1)
      → ((BlockType t_in2 t_out) ∈ Δ × params Γ t_in2 ) → Code Γ Δ t_out
```

As our last data definition in this language, we must specify our code block definition, BlockV. This consists of one constructor BlockVal, which is parameterized by a Code segment, and which is indexed by a DefType, where the input and output types match those of the code block. With this, we can actually define entire programs, rather than being relegated to evaluating single contiguous code blocks!

```
data BlockV (Δ : BCtx) : DefType → Set where
  BlockVal : ∀ {t_in t_out } → Code t_in Δ t_out → BlockV Δ (BlockType t_in t_out)
```

In the above abstract data types, the BCtx was a type context of block types. For the interpreter, we need a context of block definitions at runtime, much like how Ctx requires an Env definition in the evaluator. Our BlockContext is parameterized by a BCtx, Δ, and constructs an All value as the evaluation context equivalent. The definition may be a bit confusing, as the list Δ is an argument both to the BlockV and the All, but this is simply because each block has to be in scope for the code blocks later in the list. Δ both matches the types of each BlockV in the All, and puts their types in scope for the other block definitions in the list. Although this seems circular, it just requires that we supply the list of the block types when constructing a BlockContext.

```
BlockContext : BCtx → Set
BlockContext Δ = All (BlockV Δ) Δ
```

Now that we have all the pieces available, it is worth asking ourselves, what is a program? In this context, it is useful to conceptualize a program in this language as a list of block definitions and an entry point or a main function. We can encode this type as below.

```
Program : ∀ {t_in t_out } → BCtx → Set
Program {t_in} {t_out} Δ = (((BlockType t_in t_out) ∈ Δ) × BlockContext Δ )
```

Consider the fast Fibonacci function as an example of how we can use this data type to define an actual program. It uses a two-member list to return the current

and one previous Fibonacci numbers. This is a kind of tabularization optimization over the original Fibonacci function. This function was chosen as an example because it requires mutual recursion, list return types, and the output to work correctly, as this example writes each Fibonacci number to our output list in order. We must pass in our type signatures as the BlkCtx, where we get the list of types parameterizing the Program from. Directly writing out the abstract syntax tree (AST) results in a difficult-to-read function, especially with De Bruijn indices as variables. To assist in understanding, an equivalent version of the program has been transcribed in standard MIL syntax with highlighting below the Agda AST.

```
fastFib : Program (BlockType [] [ Word ,′′ Word ]′′ ::
                   BlockType [] [ Word ,′′ Word ]′′ ::
                   BlockType [ Word ]′′ [ Word ,′′ Word ]′′ ::
                   BlockType [ Word ]′′ [ Word ,′′ Word ]′′ ::
                   BlockType [ Word ]′′ [ Word ,′′ Word ]′′ ::
                   BlockType [ Word ]′′ [ Word ]′′ :: [])
fastFib = ( there (there (there (there (there (here refl))))) ,′
  (BlockVal
    (Bind (Output [ Val (Constant Word (+ 0)) ]′′)
    (Ta (Return
          [ Val (Constant Word (+ 0)) ,′′ Val (Constant Word (+ 0)) ]′′)))) ::
  (BlockVal
    (Bind (Output [ Val (Constant Word (+ 0)) ]′′)
    (Bind (Output [ Val (Constant Word (+ 1)) ]′′)
    (Ta (Return
```

```
            [ Val (Constant Word (+ 1)) ,″ Val (Constant Word (+ 0)) ]″))))) ::
    (BlockVal
      (Bind (PrimCall Msub [ Var (here refl) ,″ Val (Constant Word (+ 1)) ]″)
      (Bind (BlockCall (there (there (there (here refl)))) [ Var (here refl) ]″)
      (Bind (PrimCall Madd [ Var (there (here refl)) ,″ Var (here refl)]″)
      (Bind (Output [ Var (here refl) ]″)
      (Ta (Return
            [ Var (here refl) ,″ Var (there (here refl)) ]″ ))))))) ::
    (BlockVal
      (Bind (PrimCall Mweq [ Var (here refl) ,″ Val (Constant Word (+ 1))]″)
      (If (Var (here refl))
      ((there (here refl)) ,′ [])
      ((there (there (here refl))) ,′ [ Var (there (here refl))]″ )))) ::
    (BlockVal
      (Bind (PrimCall Mweq [ Var (here refl) ,″ Val (Constant Word (+ 0))]″)
      (If (Var (here refl))
      ((here refl) ,′ [])
      ((there (there (there (here refl)))) ,′ [ Var (there (here refl))]″)))) ::
    (BlockVal
      (Bind (BlockCall
              ((there (there (there (there (here refl))))))
              [ Var (here refl)]″ )
      (Ta (Return [ Var (here refl) ]″)))) :: [])

oneCase : [Word,Word]
```

```
oneCase =
  [] <- output[0]
  [] <- output[1]
  return [1,0]

indCase : [Word] >>= [Word,Word]
indCase [x] =
  [minusOne]   <- sub((x,1))
  [fib1, fib2] <- fibEntry[minusOne]
  sum          <- add((fib1,fib2))
  []           <- output[sum]
  return [sum, fib1]

zeroCase : [Word,Word]
zeroCase =
  [] <- output[0]
  return [0,0]

neZero : [Word] >>= [Word,Word]
neZero [x] =
  [eqOne] <- eq((x,1))
  if eqOne
     oneCase []
     indCase [x]
```

```
fibEntry : [Word] >>= [Word,Word]
fibEntry [x] =
  [eqZero] <- eq((x,0))
  if eqZero
     zeroCase []
     neZero   [x]

fibWrapper : [Word] >>= [Word]
fibWrapper [x] =
   [fib1,fib2] <- fibEntry[x]
   return [fib1]

main = fibWrapper
```

Now that we have gone over the definitions of our updated language and before we move onto the evaluation section, we must consider a complicating factor in writing the evaluator: what do we do about termination?

10.2 Turing completeness and the logic loophole

An important feature of Agda is that all functions are guaranteed to terminate[3], making it a total language [AAC+21f]. This is enforced by structural recursion, where the value being recursed on must be a strict sub-expression of the input

[3]There are co-inductive data types in Agda that capture infinite data structures such as streams [AAC+21b]. Totality requires that progress be made in recursion, so even the existence of co-inductive data types does not allow us to skirt a form of termination checking.

expression [AAC+21f][4]. As each recursive call must be on a strict subset of the input data structure, this structural recursion is guaranteed to terminate. It is important to understand why Agda insists on totality, so let us go back to Haskell for a short segment for some contrast.

We first established that one can define propositions as types and prove those propositions by finding values that inhabit those types in Chapter 4. We also used this technique to prove properties of type-level natural numbers in Haskell, including that addition of natural numbers is commutative. There was a type, `Void`, which is inhabited by no values. Haskell is Turing-complete, and therefore does allow infinite loops, unlike Agda. We can leverage this to inhabit a `Void` as shown below.

```
oops :: Void
oops = oops
```

Why does this work? Well, if we claim that `oops :: Void`, and we need to find a way to create a `Void` value, `oops` is in scope in the body of itself! The type of an infinite loop can be anything, including `Void`. This means that in a non-total language, it is trivial to create a bottom value, which through the principle of explosion can inhabit any type! As a result of this non-termination, in Haskell every type is inhabited by bottom[5]. With the ability to generate a bottom value at will we can prove whatever we want, for example that $1 = 0$.

```
oneEqZero :: Suc Z :~: Z
oneEqZero = principleExplosion oops
```

[4]Note that this version of termination checking does not solve the halting problem. There are functions that do not recurse structurally, but can be proven to terminate on any input, and will be excluded by Agda's termination checker. Because we cannot determine if any arbitrary program will halt, Agda restricts us to a subset of functions that can be termination checked.

[5]This is why the category of Haskell types and functions is sometimes referred to as Hask, distinct from the category of Set. [MT19]

Although hopefully the motivation behind Agda's totality is clear, one may ask why we are focusing on Agda's termination checker at all? It is important to point out that in earlier chapters our evaluators were guaranteed to terminate, as they were at most effectively lists of tails, with the evaluator structurally recursing on the code sequence. As such, we did not need to do anything fancy in order to get the termination checker to approve of our evaluator. With the ability to execute block calls however, it is fully possible to have a code block call itself, or call a chain of blocks that eventually winds up in a cycle. A naive implementation of an interpreter for this language will fail the Agda termination checker. How do we embed a language evaluator that is not guaranteed to terminate in a language that is?

We can amend our evaluator by giving it "fuel": instead of recursing only on some code block or tail call, it can carry along a natural number that represents some limit on the number of computations the interpreter can execute [RPS+19]. For example, one way of implementing this would be decreasing the fuel by one for every recursive evaluator call, and returning a nothing value in the case that we run out of fuel, when we call the evaluation function with a fuel value of zero.

With our language, the only Tail whose execution may not terminate is a BlockCall. As such, in the vast majority of cases, we do not need to decrement the fuel. However, in the case that we evaluate a BlockCall, Agda needs some assurance that this will indeed terminate, and thus we will reduce the fuel value by one each time we evaluate a BlockCall. As we will see below, this is sufficient to convince Agda that our evaluator will terminate.

10.3 A gas-powered evaluator

Now that we have described the strategy for dealing with non-termination in our evaluator, let us implement it! These evaluators are operating in the Maybe Writer monad as described in Chapter 5, and return lists of values, in keeping with the newly list-based return type of Block MIL. There is one difference in the implementation of the »= function, specifically that the with abstraction in the earlier implementation has been extracted to a helper function, writerHelper. This makes equational reasoning easier, as the helper has an explicit type, rather than requiring Agda to try and infer it.

```
writerHelper : ∀ {b} → Maybe (Writer b) → List String → Maybe (Writer b)
writerHelper nothing vs₁          = nothing
writerHelper (just (b , vs₂)) vs₁ = just (b , (vs₁ ++ vs₂))

_»=_ : ∀ {a b}
  → Maybe (Writer a )
  → (a → Maybe (Writer b))
  → Maybe (Writer b)
nothing »= f = nothing
just (a , vs₁) »= f = writerHelper (f a) vs₁
```

The evalAtom function is identical to its version in last chapter, and thus will not be described here. The evalPrimCall does need to be updated to handle our improved PrimCall type. Because our PrimCall type is indexed by a list of input types, we can use All values with the input type, and assure that the lists of values passed in matches the type and fixity specified by the PrimOp.

```
evalPrimCall : ∀ { t_in t_out }
  → PrimOp t_in t_out → All DataVal t_in → DataVal t_out
evalPrimCall Madd (a₁ :: a₂ :: []) = a₁ +ᵈ a₂
evalPrimCall Mand (a₁ :: a₂ :: []) = a₁ ∧ᵈ a₂
evalPrimCall Mmul (a₁ :: a₂ :: []) = a₁ *ᵈ a₂
evalPrimCall MisZero (a :: []) = ≡0? a
evalPrimCall Mweq (a₁ :: a₂ :: []) = a₁ ==ᵈ a₂
evalPrimCall Msub (a₁ :: a₂ :: []) = a₁ -ᵈ a₂
```

To evaluate a block call, we must first take as parameters the fuel, the arguments to the block call, the index of the BlockVal, the BlockContext, and the environment. If the fuel is a zero, we are out of gas, and simply return nothing. Otherwise, we look up the BlockVal in the BlockContext, and evaluate the code segment it contains with the parameters as the context, subtracting one from the fuel value.

```
evalBlockCall :
  ∀ {t t_in Γ Δ}
  → ℕ
  → All DataVal t_in
  → BlockType t_in t ∈ Δ
  → BlockContext Δ
  → Env Γ
  → Maybe (Writer (All DataVal t))
evalBlockCall zero param blk blkCtx env
  = nothing
evalBlockCall (ℕ.suc n) param blk blkCtx env
```

```
    = case lookup blkCtx blk of λ where
      (BlockVal c) → evalCode n c param blkCtx
```

Our evalTail function follows the same pattern, where it takes the BlockContext and the fuel, but otherwise is similar to the Writer MIL interpreter. The primary differences between these interpreters is that we have a function evalAll, which evaluates the list of inputs by mapping evalAtom across the inputs. Other than updating the Tails to accept multiple input values, the only other difference is by adding support for the BlockCall constructor, which simply calls the evalBlockCall function we defined earlier.

```
evalTail : ∀ {t Γ Δ}
  → ℕ
  → Tail Γ Δ t
  → Env Γ
  → BlockContext Δ
  → Maybe (Writer (All DataVal t))
evalTail n (Return args) env blkctx
  = return (evalAll env args)
evalTail {t} n (BlockCall i args) env blkCtx
  = evalBlockCall {t} n (evalAll env args) i blkCtx env
evalTail n (PrimCall primOp as) env blkctx
  = return [ evalPrimCall primOp (evalAll env as) ]′′
evalTail n (Output as) env blkctx with map-All (λ x → (evalAtom env x)) as
... | outs = just ([] , showDV outs )
```

Finally, the code evaluator is almost identical to the Writer MIL code evaluator,

except for two points. First, the Tail in the Bind constructor evaluates to a list of DataVals, rather than a single one as before. To account for this, the result of Tail evaluation has the environment appended to it, rather than using ::, as before. Second, we must deal with the If case. Simply, this evaluates the Atom representing the conditional, and then in the case that it evaluates to F true, it calls the first code block with the matching parameters. Otherwise, it calls the second code block.

```
evalCode : ∀ {t Γ Δ }
  → ℕ
  → Code Γ Δ t
  → Env Γ
  → BlockContext Δ
  → Maybe (Writer ((All DataVal t)))

evalCode n (Ta tail) env blkctx
  = evalTail n tail env blkctx
evalCode n (Bind tail code) env blkCtx
  = (evalTail n tail env blkCtx)
      »= (λ a → evalCode n code ( a ++′ env) blkCtx )
evalCode n (If cond ( blk₁ , arg₁ ) ( blk₂ , arg₂ )) env blkCtx
  with evalAtom env cond
... | F true = evalBlockCall n (evalAll env arg₁) blk₁ blkCtx env
... | F false = evalBlockCall n (evalAll env arg₂) blk₂ blkCtx env
```

Unique to Block MIL, we have one data type above Code on the data type hierarchy, and that is the Program. We need one final evaluator, runProgram, which takes fuel,

the supplied arguments, and a Program, and runs the Program with those arguments. Recalling that a Program consists of a BlockContext and an entry point, this calls the main function with the supplied arguments, the block context, and an empty evaluation context.

```
runProgram : ∀ {t_in t_out Δ}
  → ℕ
  → All DataVal t_in
  → Program {t_in = t_in} {t_out = t_out} Δ
  → Maybe (Writer (All DataVal t_out))
runProgram n args (main , blkCtx) = evalBlockCall n args main blkCtx []
```

Using this function, we can run our fastFib program with an input of 7, and we can see that it evaluates to the correct corresponding Fibonacci number, and that it outputs the 0th to the 7th Fibonacci numbers in the log. Thus, we have a way to run realistic programs in a large subset of MIL!

```
_ : runProgram 1000 ((W (+ 7)) :: []) fastFib
    ≡
    just ([ W (+ 13) ]′
          , [ "0" ,′ "1" ,′ "1" ,′ "2" ,′ "3" ,′ "5" ,′ "8" ,′ "13" ]′)
_ = refl
```

Overall, the changes between this interpreter and Writer MIL are relatively minor, but make a huge difference in terms of the expressiveness of the language. Now, as has been the pattern for this thesis, let us define an optimization based on these new features, and prove that applying it does not change the semantics of the program.

10.4 Optimization, termination, and proofs

Consider what happens when a conditional used in an If statement is known at compile time. We know that the corresponding block and argument combination will always be called no matter the context, and therefore we could simply rewrite the If statement as the block call corresponding to the known conditional!

We can see that in our implementation of this optimization function, eliminate-KnownIf, Ta constructed values are untouched, as are If statements with variables as their conditional. For Bind values, the Tail is untouched, but the function is recursively called on its nested code. In the case of an If with a Val constructor as its conditional, we replace it with a BlockCall of the corresponding block and arguments.

```
eliminateKnownIf : ∀ {t Γ Δ} → Code Γ Δ t → Code Γ Δ t
eliminateKnownIf (Ta x) = Ta x
eliminateKnownIf (Bind x c) = Bind x (eliminateKnownIf c)
eliminateKnownIf (If (Var x) x₁ x₂ ) = (If (Var x) x₁ x₂)
eliminateKnownIf (If (Val (Constant Flag true)) (block_t , args) _)
  = Ta (BlockCall block_t args)
eliminateKnownIf (If (Val (Constant Flag false)) _ (block_c , args))
  = Ta (BlockCall block_c args)
```

Something to note here is the power of composed optimizations. This optimization alone is unlikely to be commonly applicable, because a programmer presumably would not write an If statement with a constant conditional. However, the left monad law optimization can propagate values known at compile time throughout the rest of the Code. Thus the combination of these two optimizations allow for known variables, that is a variable that is bound to a Return, that is later used as a conditional of an If

statement to be optimized by eliminateKnownIf.

This seems like a simple optimization to prove, and in fact, in Writer MIL it would be reflexively so, as evaluation of If is implemented in terms of evalBlockCall. However, this is not Writer Mil, and as such, we have to wrestle with the complexity that possible non-termination brings. Previously we proved correctness by showing that there is a direct correspondence between the result of evaluating the optimized and unoptimized code sequence, but what happens when we do not know if these will evaluate to a value at all?

In Chapter 6, we proved our plusSimplifies optimization correct in an untyped style, in which we phrased the proposition as a statement that if the source expression evaluates to a value, then the optimized program will evaluate to that same value. This approach can be applied to optimizations in an intrinsically-typed interpreter with fuel, because not all optimizations result in the source program evaluating to the same thing as the target program when fuel is involved. For example, an optimization that inlines a known block would reduce the required fuel to reach that code segment by one, and as such, it is possible that optimization may cause a Code that previously evaluated to a nothing to evaluate to a just value. The stronger equivalence property holds in the case of the eliminateKnownIf optimization, but that is a special case. By proving the equality of evaluation over the source and target program, we can derive the weaker proposition of implication of semantic maintenance, as we can prove below that equality implies implication.

```
≡→→ : ∀{A} → {a b c : Set A} → (a ≡ b) → ((a ≡ c) → (b ≡ c))
≡→→ a≡b rewrite a≡b = λ x → x
```

First, we must prove that if some code that is a Bind terminates then the Tail bound

by it terminates as well, in our function bind⇓just⇒tail⇓just. Agda's with abstraction is advanced enough to sense that tail must evaluate to a just value with the proof of code's evaluation in context, and so, recalling that an existential requires both a value to be applied to the predicate and a proof that the predicate holds over that value, we can return the value dv_t, and the proof is reflexive.

```
bind⇓just⇒tail⇓just :
  ∀ {n t t₁ Γ Δ v₁}
  → {tail   : Tail Γ Δ t₁}
  → {code₁ : Code (t₁ ++ Γ) Δ t}
  → (env    : Env Γ)
  → (blkCtx : BlockContext Δ)
  → (code   : Code Γ Δ t)
  → code ≡ Bind {Γ} {Δ} {t₁} tail code₁
  → (evalCode n code env blkCtx ≡ just v₁)
  → ∃ (λ x → evalTail n tail env blkCtx ≡ just x)
bind⇓just⇒tail⇓just {n} env blkCtx (Bind tail code₁) refl cev
  with (evalTail n tail env blkCtx)
... | just dvₜ = dvₜ , refl
```

It should also be clear that if some Bind of tail code evaluates to a just value, then evaluating the code must evaluate to a just value. We can capture this in our bindToEq function, by a manner analogous to that of the previous proof, with the evaluation of tail in the with abstraction substituted for an evaluation of the Code.

```
bindToEq : ∀ {n Γ Δ tₜ tc v₁ log₁ v₂ log₂}
  → {env : Env Γ}
```

```
  → {blkCtx : BlockContext Δ}
  → (tail : Tail Γ Δ t_t)
  → (code : Code (t_t ++ Γ) Δ tc)
  → just (v₁ , log₁) »=
      (λ a → evalCode n code (a ++′ env) blkCtx) ≡ just (v₂ , log₂)
  → ∃[ v₃ ](evalCode n code (v₁ ++′ env) blkCtx ≡ just v₃)
bindToEq
  {n} {v₁ = v₁} {env = env} {blkCtx = blkCtx} tail code tail»=code≡just
  with (evalCode n code (v₁ ++′ env) blkCtx)
... | just dv_c = dv_c , refl
```

We can combine these two proofs in our next lemma, bindtc⇓just→c⇓just. This captures the proposition that if a Tail evaluates to a DataVal v_1 and a log, and a code segment which binds that Tail to a nested code sequence evaluates to a just value, that evaluating the nested code sequence with v_1 added to the front of the to the context evaluates to a just value. We can define a lemma bridge, which we use to manipulate the left-hand side of the equality into a form that bindToEq understands. We can use the proof tail⇓just along with a congruence to convert the just value being bound to an evalTail of tail, and then use our code⇓just to show equality to v_2. We can then pass this lemma, along with tail⇓just to bindToEq to complete the proof.

```
bindtc⇓just→c⇓just : ∀ {n Γ Δ t_t tc v₁ log₁}
  → (v₂ : Writer (All DataVal tc))
  → (env : Env Γ)
  → (blkCtx : BlockContext Δ)
  → (tail : Tail Γ Δ t_t)
```

```
    → (code : Code (tₜ ++ Γ) Δ tc)
    → (evalTail n tail env blkCtx) ≡ just (v₁ , log₁)
    → evalCode n (Bind tail code) env blkCtx ≡ just v₂
    → ∃[ v₃ ](evalCode n code (v₁ ++′ env) blkCtx ≡ just v₃)
  bindtc⇓just→c⇓just
    {n} {v₁ = v₁} {log₁ = log₁} v₂ env blkCtx tail code tail⇓just code⇓just =
    let
      bridge : (just (v₁ , log₁)
                  >>= (λ a → evalCode n code (a ++′ env) blkCtx)) ≡ just v₂
      bridge =
        begin
        (just (v₁ , log₁)
          >>= (λ a → evalCode n code (a ++′ env) blkCtx))
        ≡⟨ cong
             (λ x → x >>= (λ a → evalCode n code (a ++′ env) blkCtx))
             (sym tail⇓just) ⟩
        (evalTail n tail env blkCtx
          >>= (λ a → evalCode n code (a ++′ env) blkCtx))
        ≡⟨ code⇓just ⟩
        just v₂ ∎
    in
      bindToEq {v₁ = v₁} tail code bridge
```

Now that we have proven the important termination relations among code sequences, we must now prove a final one: that if evaluating a code sequence with some amount

of fuel results in a just value being generated, then this is true even after applying the optimization to it. That is to say, the optimization cannot cause a code sequence that previously evaluated to a just value to evaluate to a nothing value. All but one case of this proof is reflexive, the remaining case being the Bind case. We can use our lemmas bind⇓just⇒tail⇓just and bindtc⇓just→c⇓just to prove that code and tail evaluate to just values. We can then use the proof of code's evaluation to recursively call our code⇓just→simplcode⇓just function, giving us our inductive hypothesis, and the result of evaluating the optimized code sequence.

Because this function returns an existential, we need to return a dependent pair containing the result of the evaluation of the optimized code, and a proof that it is that value. The value is easy: it is simply the first projection of the recursive call. We start the proof with the left-hand side of the equality: the evaluation of the the optimized Bind expression. Through a series of equational reasoning steps, using our inductive hypothesis and tail⇓justv$_t$, we can complete this proof.

```
code⇓just→simplcode⇓just : ∀ {n Γ Δ t v₁}
  → {env : Env Γ}
  → {blkCtx : BlockContext Δ}
  → (code : Code Γ Δ t)
  → (evalCode n code env blkCtx ≡ just v₁)
  → ∃[ v₂ ](evalCode n (eliminateKnownIf code) env blkCtx ≡ just v₂)
code⇓just→simplcode⇓just {v₁ = v₁} (Ta x) eq
  = v₁ , eq
code⇓just→simplcode⇓just {v₁ = v₁} (If (Var x) x₁ x₂ ) eq
  = v₁ , eq
```

```
code⇓just→simplcode⇓just {v₁ = v₁} (If (Val (Constant Flag false)) x₁ x₂) eq
  = v₁ , eq
code⇓just→simplcode⇓just {v₁ = v₁} (If (Val (Constant Flag true)) _ _) eq
  = v₁ , eq
code⇓just→simplcode⇓just
  {n} {Γ} {Δ} {t} {v₁} {env} {blkCtx} (Bind {tₜ} tail code) code⇓justv₁ =
  let
    ((vₜ , logₜ) , tail⇓just)    =
      bind⇓just⇒tail⇓just env blkCtx (Bind tail code) refl code⇓justv₁
    (termVal , code⇓justvc) =
      bindtc⇓just→c⇓just v₁ env blkCtx tail code tail⇓just code⇓justv₁
    ((v_c , log_c) , ih)          =
      code⇓just→simplcode⇓just code code⇓justvc
  in
    (v_c , (logₜ ++ log_c)) , (
      begin
      evalCode n (eliminateKnownIf (Bind tail code)) env blkCtx
      ≡⟨ refl ⟩
        evalTail n tail env blkCtx »=
          (λ a → evalCode n (eliminateKnownIf code) (a ++′ env) blkCtx)
      ≡⟨ cong (λ x → x »=
          (λ a → evalCode n (eliminateKnownIf code) (a ++′ env) blkCtx))
          tail⇓just ⟩
        just ( vₜ , logₜ) »=
```

```
      (λ a → evalCode n (eliminateKnownIf code) (a ++′ env) blkCtx)
    ≡⟨ refl ⟩
      writerHelper
        (evalCode n (eliminateKnownIf code) (v_t ++′ env) blkCtx)
        log_t
    ≡⟨ cong (λ x → writerHelper x log_t) ih ⟩
      writerHelper (just (v_c , log_c)) log_t
    ≡⟨ refl ⟩
      just (v_c , ( log_t ++ log_c)) ■)
```

Next, we must prove that if a Bind of tail to code evaluates to nothing, then evaluating code with the DataVal generated from evaluating tail added to the front of the context also evaluates to a nothing. We can begin by inspecting the evaluation of code. In the case this evaluates to a value, we can rewrite the hole to have type nothing ≡ nothing using our proof code⇓nothing, which is then reflexive. In the case where code evaluates to a just value, we need to generate a ⊥ value, as this case should not be reachable. We can do this first by creating bind⇓just, which proves that evaluating Bind tail code in this environment results in a just value. This is straightforward to do, as we have a proof that code evaluates to a just value. Now we have two proofs that contradict: that evalCode n (Bind tail code) env blkCtx is equal to both nothing and a just value. We can use our lemma just≢nothing to create a ⊥ value, and then use ⊥-elim to generate a value of the type we need to finish this proof.

```
  bindtc⇓nothing→c⇓nothing : ∀ {n t_t v_t Γ Δ t_c log_t}
    → (env : Env Γ)
    → (blkCtx : BlockContext Δ)
```

```
    → (tail : Tail Γ Δ t_t)
    → (code : Code (t_t ++ Γ) Δ t_c)
    → evalTail n tail env blkCtx ≡ just (v_t , log_t)
    → evalCode n (Bind tail code) env blkCtx ≡ nothing
    → evalCode n code (v_t ++′ env) blkCtx ≡ nothing
  bindtc⇓nothing→c⇓nothing
    {n} {_} {v_t} {log_t = log_t} env blkCtx tail code ev_t ev_c
    with inspect' (evalCode n code (v_t ++′ env) blkCtx)
  ... | nothing with≡ code⇓nothing rewrite code⇓nothing = refl
  ... | just v_c with≡ code⇓just =
    let
      bind⇓just : ∃[ v ](evalCode n (Bind tail code) env blkCtx ≡ just v)
      bind⇓just = ( (proj₁ v_c) , (log_t ++ proj₂ v_c)) , (
        ( begin
        evalCode n (Bind tail code) env blkCtx
        ≡⟨ refl ⟩
          ((evalTail n tail env blkCtx »=
              (λ a₁ → evalCode n code (a₁ ++′ env) blkCtx)))
        ≡⟨ cong
              (λ x → x »=
                (λ a₁ → evalCode n code (a₁ ++′ env) blkCtx) )
              ev_t ⟩
        writerHelper (evalCode n code (v_t ++′ env) blkCtx) log_t
        ≡⟨ cong (λ x → writerHelper x log_t) code⇓just ⟩
```

```
        writerHelper (just vc) logt
        ≡⟨ refl ⟩
          just ((proj₁ vc) , (logt ++ proj₂ vc)) ■ ))
    in
      ⊥-elim
        (just≢nothing
          (evalCode n (Bind tail code) env blkCtx) (proj₂ bind⇓just) evc)
```

We can now begin to prove that the eliminateKnownIf function does not alter the result of evaluation of the code it is applied to, in the case that the Code evaluates to a just value. Every case is reflexive except for the Bind case, which requires a bit of work to prove. We know that tail evaluates to a just, and generate this proof using bind⇓just⇒tail⇓just and ev≡v₁. We also can produce a proof that the evaluation of code results in a just value, by using our bindtc⇓just→c⇓just lemma. Finally, we can generate a proof that the optimized code will terminate using code⇓just→simplcode⇓just. Using these lemmas, we can step through a relatively straightforward equational reasoning chain to prove our proposition.

```
  code⇓just≡elimKnownIfCode⇓just :
    ∀ {t Γ Δ v₁ v₂}
    → {env : Env Γ}
    → {blkCtx : BlockContext Δ}
    → (n : ℕ)
    → (code : Code Γ Δ t)
    → (evalCode n code env blkCtx ≡ just v₁)
    → evalCode n code env blkCtx
```

```
      ≡ (evalCode n (eliminateKnownIf code) env blkCtx)
code⇓just≡elimKnownIfCode⇓just n (Ta x) ev≡v₁ = refl
code⇓just≡elimKnownIfCode⇓just n (If (Var x) x₁ x₂ ) ev≡v₁ = refl
code⇓just≡elimKnownIfCode⇓just n (If (Val (Constant Flag false)) _ _) ev≡v₁
  = refl
code⇓just≡elimKnownIfCode⇓just n (If (Val (Constant Flag true)) _ _) ev≡v₁
  = refl
code⇓just≡elimKnownIfCode⇓just
  {v₁ = v₁} {env = env} {blkCtx = blkCtx} n (Bind tail code) ev≡v₁ =
  let
     ((vₜ , logₜ) , tail⇓justvₜlogₜ)
       = bind⇓just⇒tail⇓just env blkCtx (Bind tail code) refl ev≡v₁
     (vc , code⇓justvc)
       = bindtc⇓just→c⇓just v₁ env blkCtx tail code tail⇓justvₜlogₜ ev≡v₁
     (vₛ          , simpcode⇓vₛ)
       = code⇓just→simplcode⇓just code code⇓justvc
     ih : evalCode n code (vₜ ++′ env) blkCtx ≡
            evalCode n (eliminateKnownIf code) (vₜ ++′ env) blkCtx
     ih = code⇓just≡elimKnownIfCode⇓just {v₂ = vₛ} n code code⇓justvc
  in
  begin
    (evalTail n tail env blkCtx »=
      (λ z → evalCode n code (z ++′ env) blkCtx))
  ≡⟨ cong
```

$(\lambda\ x \rightarrow x \gg=$

$(\lambda\ z \rightarrow \mathsf{evalCode}\ n\ code\ (z \mathbin{++'} env)\ blkCtx))\ tail{\Downarrow}justv_t log_t\ \rangle$

$\mathsf{writerHelper}\ (\mathsf{evalCode}\ n\ code\ (v_t \mathbin{++'} env)\ blkCtx)\ log_t$

$\equiv\langle\ \mathsf{cong}\ (\lambda\ x \rightarrow \mathsf{writerHelper}\ x\ log_t)\ ih\ \rangle$

$\mathsf{writerHelper}\ (\mathsf{evalCode}\ n\ (\mathsf{eliminateKnownIf}\ code)\ (v_t \mathbin{++'} env)\ blkCtx)\ log_t$

$\equiv\langle\ \mathsf{cong}$

$(\lambda\ x \rightarrow x \gg=$

$(\lambda\ z \rightarrow \mathsf{evalCode}\ n\ (\mathsf{eliminateKnownIf}\ code)\ (z \mathbin{++'} env)\ blkCtx))$

$(\mathsf{sym}\ tail{\Downarrow}justv_t log_t)\ \rangle$

$(\mathsf{evalTail}\ n\ tail\ env\ blkCtx \gg=$

$(\lambda\ z \rightarrow \mathsf{evalCode}\ n\ (\mathsf{eliminateKnownIf}\ code)\ (z \mathbin{++'} env)\ blkCtx))\ \blacksquare$

However, what about the case when the target Code evaluates to a nothing with some given fuel? In this particular case, if a Code evaluates to nothing, the optimized Code evaluates to nothing. So how can we prove it?

We can first case over the code value, and all cases but the Bind case are trivially solved by the proof that evaluating the code results in a nothing value. This makes sense, as in these cases the code block is unchanged, or is changed to a code block that is reflexively equivalent under evaluation. For the Bind case, we can inspect the evaluation of tail. In the case where it evaluates to a nothing value as above, we can rewrite the hole with tail⇓nothing, which results in the hole being reflexive[6]

In the case where tail evaluates to a just value, we must then inspect the result

[6]This looks like it should be solvable by inserting tail⇓nothing in the hole without rewriting, but it is not. tail⇓nothing : evalTail n tail env blkCtx ≡ nothing, and the hole initially has type (evalTail n tail env blkCtx »= (λ a → evalCode n (eliminateKnownIf code) (a ++′ env) blkCtx)) ≡ nothing. Rewriting replaces evalTail n tail env blkCtx with nothing, which allows Agda to refine the goal to nothing ≡ nothing, which is reflexive.

of evaluating code. In the nothing case, we can rewrite this with tail⇓just and the inductive hypothesis to refine the hole to reflexivity. The just case should not be reachable, so we have to reach for our trusty ⊥-elim! We have proofs that code evaluates to a just value, code_1⇓justv_c, and we can generate a proof that it evaluates to nothing given our proof that (Bind tail code) evaluates to nothing, code⇓nothing. With just≢nothing we can then generate the ⊥ value we need to finish this proof.

```
code⇓nothing→simplcode⇓nothing : ∀ {n Γ Δ t}
  → {env : Env Γ}
  → {blkCtx : BlockContext Δ}
  → (code : Code Γ Δ t)
  → evalCode n code env blkCtx ≡ nothing
  → evalCode n (eliminateKnownIf code) env blkCtx ≡ nothing
code⇓nothing→simplcode⇓nothing
  (Ta x) code⇓nothing = code⇓nothing
code⇓nothing→simplcode⇓nothing
  (If (Var x) x1 x2 ) code⇓nothing = code⇓nothing
code⇓nothing→simplcode⇓nothing
  (If (Val (Constant Flag false)) x1 x2 ) code⇓nothing = code⇓nothing
code⇓nothing→simplcode⇓nothing
  (If (Val (Constant Flag true)) x1 x2 ) code⇓nothing = code⇓nothing
code⇓nothing→simplcode⇓nothing
  {n} {env = env} {blkCtx = blkCtx} (Bind tail code) code⇓nothing
  with inspect' (evalTail n tail env blkCtx)
... | nothing with≡ tail⇓nothing rewrite tail⇓nothing = refl
```

```
... | just (a , b) with≡ tail⇓just
  with inspect' (evalCode n code ( a ++′ env) blkCtx)
... | just v_c with≡ code₁⇓justv_c
  = ⊥-elim (
    just≢nothing
      (evalCode n code ( a ++′ env) blkCtx)
      code₁⇓justv_c
      (bindtc⇓nothing→c⇓nothing env blkCtx tail code tail⇓just code⇓nothing))
... | nothing with≡ code₁⇓nothing
  rewrite tail⇓just |
  code⇓nothing→simplcode⇓nothing
  {n} {env = ( a ++′ env)} {blkCtx = blkCtx} code code₁⇓nothing = refl
```

We can then combine our proofs for when our Code evaluates to nothing and when it evaluates to just, to prove that evaluating some code is always equal to evaluating eliminateKnownIf code in the same context. First we can inspect the result of evaluating code. We can prove the nothing case by using our code⇓nothing→simplcode⇓nothing lemma and a rewrite with code⇓nothing. The just case can be solved by directly calling our code⇓just≡elimKnownIfCode⇓just lemma.

```
code⇓≡elimKnownIfCode⇓ :
  ∀ {t Γ Δ}
  → (env : Env Γ)
  → (blkCtx : BlockContext Δ)
  → (n : ℕ)
  → (code : Code Γ Δ t)
```

```
→ evalCode n code env blkCtx
    ≡ (evalCode n (eliminateKnownIf code) env blkCtx)
code⇓≡elimKnownIfCode⇓ env blkCtx n code
  with inspect' (evalCode n code env blkCtx)
... | nothing with≡ code⇓nothing rewrite code⇓nothing =
    sym (code⇓nothing→simplcode⇓nothing code code⇓nothing)
... | just x₁ with≡ code⇓just =
    code⇓just≡elimKnownIfCode⇓just {v₂ = x₁} n code code⇓just
```

With this, we end the chapter, and the technical part of the thesis. In this chapter, we added the ability for our interpreter to initiate block calls, to have conditional branches, and to run a program with a supplied input. We then implemented an optimization that allows for reduction of If statements to block calls if the conditional is known, and proved it correct by reasoning about the semantics of our fuel-based evaluator.

Chapter 11

Conclusion

We have finally concluded our march through the interpreters, and in doing so, ended the technical part of the thesis! Hopefully by gradually working through interpreters of greater complexity, the nuances of the final interpreter were easier to understand than had this thesis started with the full implementation.

11.1 Future works

Obviously, there is more work to do in terms of more completely modeling MIL in this style of interpreter. This includes two primary goals: adding the missing features and proving correct all of the optimizations mentioned in the MIL paper [JBC18] in a complete implementation of the language and the interpreter.

The features that must be added to give a complete implementation of MIL are closures, user-defined data types (and therefore case statements), import (called "require" in MIL) statements, export definitions, and top-level definitions. In addition to requiring additional constructors of existing data types, such as adding the case construct to the `Code` data type, improvements to the fundamental representation of the language may be needed to accommodate these extra features.

For example, the strategy of using an `Any` value to look up variables worked

well for our subset of MIL because there were really two scopes: the block definitions, which were in scope everywhere, and the variables introduced in code segments, which were only in scope in the rest of the code segment. However, this is not a sufficiently advanced scoping system to allow more advanced features, such as imports, with ease. Future work could implement scope graphs, which allow not just a linear indexing of variables as our All based context did, but would also allow us to define paths to in-scope variables as a graph [Cas19].

In addition, this thesis only touched on a subset of the optimizations described in the MIL paper. As such, verification of all the optimization algorithms on a feature-complete implementation of MIL would give maximal assurance that the optimizations do not alter the semantics of the programs they are applied to. Hopefully the basic strategies used in this thesis can inform the future work of proving other optimizations correct.

Further, the optimizations described in the MIL paper are likely not the only valid optimizations that exist. The task of developing new optimizations on this platform and proving them correct, as well as optimizing parameters for graded optimizations such as how long to unroll loops and which block calls to in-line is a daunting but important one. As optimal optimization is undecidable in the general case[1], optimization platforms have traditionally been designed as a series of passes with set orders and parameters. There are more modern techniques that are driven by heuristics to decide what optimizations to call and in what order [CST02]. Integrating these techniques into the MIL optimizer may result in more optimal optimizations

[1]For example, if we have a program consisting of a function call f followed by a sequence of other statements, if the function f does not halt, the compiler could delete everything after the function call. This optimization requires that the optimizer solve the halting problem to optimize to the fullest extent, which we know it cannot do. The halting problem is not a special case, Rice's theorem [Sak21] states that any non-trivial semantic property of a program is undecidable.

than the current approach, although only experimentation will tell.

Although we have discussed MIL in the context of the Habit compiler, there is nothing intrinsic to the language that prevents its use in other functional language compilers. Adding MIL back ends to more established languages may have the benefit of expanding its optimization capabilities to other languages, and allowing other language ecosystems to increase the optimization capabilities of MIL.

11.2 Discussion

The lessons learned from this thesis can be split into three different sections: facts about the language and interpreter design used, background on types, and lessons on the usefulness of typing in a more subjective context.

11.2.1 Intrinsically-typed definitional interpreters

The interpreter design we used conferred some benefits over a less strongly-typed version. In Chapter 6, we implemented intrinsically-typed and untyped interpreters of our SimpleLang language and proved the correctness of corresponding optimizations using the untyped and intrinsically-typed styles. Intrinsically-typed definitional interpreters give us properties of well-typedness and well-scopedness for free, and as such, the proof of correctness for the specific optimization we implemented was shorter than the untyped version. The intrinsically-typed proofs in this thesis did not require us to reason explicitly about these typing and scoping properties. Rather, expressions that violate these properties are not constructable as valid expressions. In the untyped version, however, we had to explicitly prove why we did not have to consider the cases when variable lookups failed or sub-expressions were ill-typed, requiring more lemmas

and a greater number of cases to consider.

In addition to a change in proof style, it is possible that each strategy is better suited for proving different kinds of propositions. For example, straightforward optimizations that did not alter the context were relatively easy to prove in the intrinsically-typed style, but great difficulties arose when modifications to that context were involved (even with the simple De Bruijn indexing, as opposed to scope-graphs). It is possible that propositions where the context is modified would be more easily proven in the untyped style, given the difficulty of reasoning about these in an intrinsically-typed style.

Further work is needed to move these suggestions beyond the world of conjecture. A rigorous study where different kinds of optimizations are proven correct in both styles may find that one is better than the other, or that they are both better suited for different kinds of proofs. If the latter is found to be the case, a further possibility is that one could implement two versions of language definitions and evaluators: untyped and intrinsically-typed. One could then attempt to prove that proofs of optimization correctness over the intrinsically-typed interpreter imply that a type-erased version of the optimization is correct, to leverage the strengths of both styles, as shown in Figure 11.1.

11.2.2 Broader applications of types in industry

Although this thesis primarily focused on dependent types for their ability to prove propositions about code explicitly, dependent types are a powerful tool that allow safe construction of functions that are not possible in less expressive type systems. For example, printf is a function in C whose input type depends on a format string passed to it as a parameter [rnet]. This dependency is not enforced at

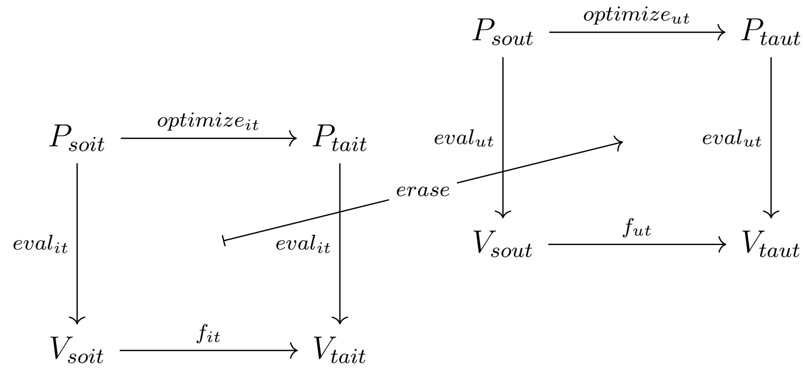

Figure 11.1: A possible path forward with integrating intrinsically-typed and untyped interpreters. In subscripts: *ta* indicates a target, *so* indicates source, *it* indicated intrinsically-typed, *ut* indicates untyped.

compile time in C, however, because the ability to construct an input type based on an argument is only conferred by dependent types, which C does not support. As such, the mismatch between format strings and supplied variables is a source of many vulnerabilities [Scu01] that are not detectable in the C type system. Dependent types provide the ability to generate the type of input parameters from the format string, allowing us to write a type-safe printf [Suz17] and to eliminate this entire class of vulnerabilities. In addition, rather than being relegated to extremely niche languages, mostly restricted to the mathematics and formal methods community, languages like Idris2 [Bra21] and ATS [Xi17] promise a new generation of more practical dependently typed languages, and will hopefully allow us to leverage these capabilities in broader settings.

One does not have to jump straight to dependent types to give greater assurances than a simple type system typically provides. Refinement types, such as those found in LiquidHaskell [VSJ+14], do not allow us to run arbitrary computations on our types, but instead allow one to define predicates that must hold over simple types.

This gives us many of the benefits of dependent types, but without as large of an amount of programmer effort required. Even in less advanced type systems, type-level programming gives us the ability to use type-safe data structure operations — much like dependent types, such as not allowing pop to be called on an empty vector — in languages without dependent types such as Haskell [Chi21] and Rust [Lin20]. Other typing features allow us to restrict the number of uses of variables to limit access or to allow for more aggressive optimization. One does not need to get into esoteric languages to use these: Rust's move semantics is an example of affine types that limit variables to up to one use, and Haskell has an experimental linear types extension that allows restriction of variables to exactly one use [GHC]. There are even ways to lift protocols into the domain of types with session types [HLV$^+$16] and ways of managing permissions with co-effects [Pet17, p.53]. We could spend an entire thesis summarising the different advanced type systems that have appeared in recent years, but the basic thrust of such a paper would be that modern type systems give us a variety of dimensions and magnitudes by which we can make the behavior of our programs more explicit, and get guarantees along these axes.

The discussion on modern type theory research is not to say that fancy type systems are required to reap the benefits of static typing! It appears that dynamically-typed languages that are commonly used in industry have created problems for companies using them due to their lack of compile-time guarantees, with Python being the clearest example of this. As a case study, consider that Python was the most commonly used language at Dropbox, but that "the dynamic typing in Python made code needlessly hard to understand and started to seriously impact productivity" [Leh19]. Over the next four years, Dropbox added gradual type annotations to over four million lines of code, mostly manually, which was seen as a net positive,

and which continues to this day. It was seen as a net benefit to invest in developing a gradual typing framework and manually annotating existing codebases over years, rather than to continue fighting against the dynamically-typed nature of their Python codebase.

Finally, although this is hard to quantify with hard data, for many, types make programming more fun! Although this thesis should have made clear the results of using a language like Agda to write programs, it is hard to illustrate how helpful feedback from the compiler was in obtaining these results. The ability to ask the compiler what type of expression goes in a spot changes debugging from an exercise in sifting through sand in the dark to a conversation with a helpful friend. Even simple types give guarantees that rule out entire classes of frustrating to debug and unpredictable errors, leaving more time for programming and requiring less stepping through code or writing debugging print statements. This is not just a personal opinion: according to the 2020 Stack overflow developers survey, four out of the five "most loved" languages have static types, and the remaining one, Python, has optional type hints, as described above [20]. Although the exact makeup of that list does vary significantly from year to year, it is clear that types are perceived as helpful among a variety of developers.

11.2.3 What was not covered

This thesis focused on one aspect of program hardening: program correctness via static typing. It is important to know that this is not the only way to guarantee correctness with respect to some property. For a simple example, exhaustively testing the input space is one way to prove that such a property holds. There are other methods, such as model checking [BK08] and symbolic execution [DE82], that attempt

to map out the state space of a program and demonstrate that a property holds in each state. Each approach has its own advantages and disadvantages, and the benefits of static typing — that it gives real-time feedback via type checking and is not as susceptible to the state space explosion problem — can definitely be outweighed in terms of labor time if the state space is tractable with automated techniques.

The focus on lack of failure or incorrect behavior at runtime and type-level guarantees captures one aspect of program correctness and resilience. An important consideration is that we have been reasoning about our programs as platonic forms [WN22], existing in a realm where no bits are flipped by cosmic rays, the computer never runs out of memory, and there are no subtle race conditions. What if a program is verified to only return natural numbers in a range, and then a cosmic ray flips a bit in memory that causes the return value to exceed that range? What then?

In many cases, assurance of one of these axes of program resilience is enough. For example, we can imagine a data analysis program that, as long as we have assurance that if it does finish evaluating, it returns the right result, does not need to be assured to run to completion 100% of the time. Does it really matter if it crashes occasionally and we have to re-run it? Alternatively, maybe there is a web server where we do not really care if it operates exactly as intended, as long as it is resilient enough to have close to 100% uptime. There are other cases, for example, planes or implanted medical devices, for which both are critically important.

This is to say that static type systems and verified programs are important tools in preventing catastrophic errors, but that they are one view into program correctness, with the ability to prevent errors along that axis. A holistic approach to correctness moves beyond one tool or one medium and views entire systems together. For example, if we revisit the Therac-25 mentioned in Chapter 1, a holistic approach would recognize

that it is both true that verification of important safety properties in the machine's programming was critically important, and that the hardware interlocks should have been left on.

11.3 Doing it right

When talking about targets for verification, one sees the term "critical system" often used. This generally refers to planes, nuclear reactors, medical devices, or other systems that would directly lead to loss of life, incur significant economic costs, or cause the loss of sensitive data in the case of failure or significant error [HC10]. This thesis opened with the Therac-25 example for exactly this reason: it is a clear case where a buggy system and a dash of hubris led to the unnecessary death or injury of six people.

These obviously critical systems are undoubtedly important, and when the author had to undergo medical imaging, they found themself thinking of the Therac-25. However, it is possible that these systems are not the primary software systems capable of causing harm. Software systems are increasingly being used to determine eligibility for welfare assistance, and in doing so, routinely flag people incorrectly as fraudulent applicants [Gil21]. In this case, the difference between one receiving the housing and food assistance they are qualified for, or otherwise starving in the street, simply hangs in the balance of a system that undoubtedly is not tested and scrutinized to the level of a critical system.

Medical devices themselves are carefully controlled, and great lengths are gone to in order to minimize the likelihood of critical errors. Electronic health records, however, are not subject to this scrutiny, and a recent study conducted on them has

shown that they fail to detect potentially harmful drug interactions and medication dosage errors two-thirds of the time. The study which revealed these problems went as far as to say that "serious safety vulnerabilities persist in these [electronic health records]" [CHC+20]. So, the difference between one being accidentally given a fatal overdose or drug combination, or receiving proper treatment is simply a software error away.

Even for systems that are not critical-system adjacent, there can be major fallout from simple bugs in innocuous-seeming programs. There have been cases where software errors have impacted test results in standardized testing [Wal17] and caused incorrect grades to be entered on students' final grade calculations [Dre19]. Again, the difference between getting into college or forever fighting an uphill battle for gainful employment could be determined by a rounding error.

Not all software is equally impactful or vulnerable to critical errors, but we now have the tools to rule out entire classes of errors and vulnerabilities while not imposing a large burden on the programmers. There is clearly a range between only ad-hoc tests with dynamic types and a program written in a dependently typed language with proofs of its important properties, and hopefully developers choose an appropriate place on this spectrum for their projects. The buildup from simple types to dependent types in this thesis has hopefully helped illustrate at least part of this range.

In the 20th century, humanity unknowingly began a grand and transformative project: to integrate computers into humanity, physically and societally. If this seems like overstating the case, consider how every aspect of our lives has become integrated with computers and the results of computation. Every vocation from mathematics to working at a restaurant has become fundamentally changed as the result of the proliferation of computers, but this is only the beginning. Our schooling has become

increasingly delivered via computers [MIC+20], both by digitizing traditional teaching methods such as online lectures [DP07] and e-books [Cen12], but also by novel teaching methods such as simulating surgery via virtual reality using robotic haptic feedback [HGV+18].

Even our social lives have come to revolve around social networks, messaging services, and other digital interactions. Most of us carry around small computers in the form of mobile phones that have become our diaries, planners, and maps and, in doing so, have become a digital extension of our consciousness. The availability of the internet means that the entire world's collected knowledge is available on any subject imaginable, as fast as one's internet connection will allow, and it is now in our pocket.

Our medications are often developed by high-speed massively parallel drug discovery programs [MPG+21] [SMM11], and their effects are analyzed by various computer based-data analysis programs [BGC+16]. When something is suspected to be medically wrong with us, we use gigantic machines with spinning computerized magnets or radiation emitters to look inside the human body as a living dissection, something unimaginable even a century ago.

We replace our faulty hearts with new mechanical hearts, controlled by computers and designed with the assistance of computers. We implant electrical simulators into our brains to treat diseases [HCE16], and there are even those attempting to integrate our brains with computers so that we can interface with them in a less consciously-mediated way [Mus19].

There is no reason to believe that this pace will slow down or stop, and so the future of humanity, and as an extension, the future of the world, is effectively that of a planet infiltrated by a massively distributed computation network. We now have and continue to develop better tools to understand what exactly these computational

systems will be doing, if only we choose to use them. The question we as a species need to ask ourselves is whether this digital transformation is worth doing. And if so, isn't it worth doing it right?

References

[18] Peano axioms, 2018. URL: `https://encyclopediaofmath.org/index.php?title=Peano_axioms`.

[20] Stack overflow developer survey 2020, 2020. URL: `https://insights.stackoverflow.com/survey/2020`.

[21a] A new functional programming language, 2021. URL: `https://www.habit-lang.org/`.

[21b] Alb, 2021. URL: `https://github.com/habit-lang/alb`.

[21c] Cwe-193: off-by-one error, March 2021. URL: `https://cwe.mitre.org/data/definitions/193.html`.

[21d] Mil-tools: tools for mil, a monadic intermediate language, 2021. URL: `https://github.com/habit-lang/mil-tools`.

[22] Llvm language reference manual, 2022. URL: `https://llvm.org/docs/LangRef.html#phi-instruction`.

[AAC+21a] Andreas Abel, Guillaume Allais, Liang-Ting Chen, Jesper Cockx, Nils Anders Danielsson, Víctor López Juan, Ulf Norell, Andrés Sicard-Ramírez, Andrea Vezzosi, Tesla Ice Zhang, and et al. Agda-stdlib/core.agda at master · agda/agda-stdlib, 2021. URL: `https://github.com/agda/agda-stdlib/blob/master/src/Relation/Binary/PropositionalEquality/Core.agda`.

[AAC+21b] Andreas Abel, Guillaume Allais, Liang-Ting Chen, Jesper Cockx, Nils Anders Danielsson, Víctor López Juan, Ulf Norell, Andrés Sicard-Ramírez, Andrea Vezzosi, Tesla Ice Zhang, and et al. Coinduction, 2021. URL: `https://agda.readthedocs.io/en/v2.6.2.1/language/coinduction.html`.

[AAC+21c] Andreas Abel, Guillaume Allais, Liang-Ting Chen, Jesper Cockx, Nils Anders Danielsson, Víctor López Juan, Ulf Norell, Andrés Sicard-Ramírez, Andrea Vezzosi, Tesla Ice Zhang, and et al. Function definitions, 2021. URL: `https://agda.readthedocs.io/en/v2.6.2.1/language/function-definitions.html`.

[AAC+21d] Andreas Abel, Guillaume Allais, Liang-Ting Chen, Jesper Cockx, Nils Anders Danielsson, Víctor López Juan, Ulf Norell, Andrés Sicard-Ramírez, Andrea Vezzosi, Tesla Ice Zhang, and et al. Pattern synonyms, 2021. URL: https://agda.readthedocs.io/en/v2.6.2.1/language/pattern-synonyms.html.

[AAC+21e] Andreas Abel, Guillaume Allais, Liang-Ting Chen, Jesper Cockx, Nils Anders Danielsson, Víctor López Juan, Ulf Norell, Andrés Sicard-Ramírez, Andrea Vezzosi, Tesla Ice Zhang, and et al. Sort system, 2021. URL: https://agda.readthedocs.io/en/latest/language/sort-system.html#sort-system.

[AAC+21f] Andreas Abel, Guillaume Allais, Liang-Ting Chen, Jesper Cockx, Nils Anders Danielsson, Víctor López Juan, Ulf Norell, Andrés Sicard-Ramírez, Andrea Vezzosi, Tesla Ice Zhang, and et al. What is agda?, 2021. URL: https://agda.readthedocs.io/en/v2.6.2.1/getting-started/what-is-agda.html.

[AAC+21g] Andreas Abel, Guillaume Allais, Liang-Ting Chen, Jesper Cockx, Nils Anders Danielsson, Víctor López Juan, Ulf Norell, Andrés Sicard-Ramírez, Andrea Vezzosi, Tesla Ice Zhang, and et al. With-abstraction, 2021. URL: https://agda.readthedocs.io/en/v2.6.1.3/language/with-abstraction.html#the-inspect-idiom.

[ACD+15] Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, Armando Solar-Lezama, and et al., 2015. URL: https://web.mit.edu/6.005/www/fa15/classes/09-immutability/.

[BGC+16] Parveen Bansal, SupreetKaur Gill, AjayFrancis Christopher, and Vikas Gupta. Emerging role of bioinformatics tools and software in evolution of clinical research. *Perspectives in Clinical Research*, 7(3):115, 2016. DOI: 10.4103/2229-3485.184782.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. The MIT Press, 2008.

[Bla02] Rex Black. *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software testing*. Wiley, 2002.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: design and implementation. *Journal of Functional Programming*, 23(5):552–593, August 2013. DOI: 10.1017/s095679681300018x. URL: https://eb.host.cs.st-andrews.ac.uk/writings/ivor.pdf.

[Bra21] Edwin Brady. Idris 2: quantitative type theory in practice, 2021. arXiv: 2104.00480 [cs.PL].

[Bro20] Cosimo Perini Brogi. Curry-howard-lambek correspondence for intuitionistic belief, 2020. arXiv: 2006.02417 [math.LO].

[BRT+17] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. DOI: 10.1145/3158104. URL: https://doi.org/10.1145/3158104.

[Car97] John Carmack. Doom, 1997. URL: https://github.com/id-Software/DOOM/.

[Cas19] Katherine Imhoff Casamento. *Correct-by-Construction Typechecking with Scope Graphs*, 2019.

[CDD+19] David Thrane Christiansen, Iavor S. Diatchki, Robert Dockins, Joe Hendrix, and Tristan Ravitch. Dependently typed haskell in industry (experience report). *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. DOI: 10.1145/3341704. URL: https://doi.org/10.1145/3341704.

[Cen12] Pew Research Center. The rise of e-reading, April 2012. URL: https://www.pewresearch.org/internet/2012/04/04/the-rise-of-e-reading-5/.

[CGJ+04] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. Core erlang 1.0.3 language specification, November 2004. URL: https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf.

[CHC+20] David C. Classen, A. Jay Holmgren, Zoe Co, Lisa P. Newmark, Diane Seger, Melissa Danforth, and David W. Bates. National Trends in the Safety Performance of Electronic Health Record Systems From 2009 to 2018. *JAMA Network Open*, 3(5):e205547–e205547, May 2020. ISSN: 2574-3805. DOI: 10.1001/jamanetworkopen.2020.5547. eprint: https://jamanetwork.com/journals/jamanetworkopen/articlepdf/2766545/classen_2020_oi_200265.pdf. URL: https://doi.org/10.1001/jamanetworkopen.2020.5547.

[Chi21] Artem M. Chirkin. Easytensor: many-dimensional type-safe numeric ops, 2021. URL: https://github.com/achirkin/easytensor#readme.

[Cro] Cooper Crouse-Hinds. *AN9003 - A Users Guide to Intrinsic Safety.* URL: https://www.mtl-inst.com/images/uploads/datasheets/App_Notes/AN9003.pdf.

[CST02] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23:2002, 2002.

[dBru72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. DOI: 10.1016/1385-7258(72)90034-0.

[DE82] R.B. Dannenberg and G.W. Ernst. Formal program verification using symbolic execution. *IEEE Transactions on Software Engineering*, SE-8(1):43–52, 1982. DOI: 10.1109/TSE.1982.234773.

[Dia07] Iavor Sotirov Diatchki. *High-Level Abstractions for Low-Level Programming*. PhD thesis, 2007.

[Doc06] Robert Dockins. The ghc typechecker is turing-complete, 2006. URL: https://mail.haskell.org/pipermail/haskell/2006-August/018355.html.

[DP07] Stavros Demetriadis and Andreas Pombortsis. E-lectures for flexible learning: a study on their learning efficiency. *Journal of Educational Technology & Society*, 10(2):147–157, 2007. ISSN: 11763647, 14364522. URL: http://www.jstor.org/stable/jeductechsoci.10.2.147.

[Dre19] Jonathan Drew. Software error sends incorrect test grades to nc schools, January 2019. URL: https://www.salisburypost.com/2019/01/18/software-error-sends-incorrect-test-grades-to-nc-schools/.

[EW12] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *SIGPLAN Not.*, 47(12):117–130, September 2012. ISSN: 0362-1340. DOI: 10.1145/2430532.2364522. URL: https://doi.org/10.1145/2430532.2364522.

[Far17] Anton Farmar. A+b=b+a? prove it!, October 2017. URL: https://www.codewars.com/kata/59db393bc1596bd2b700007f.

[GHC] Team GHC. 6.4.17. linear types. URL: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/linear_types.html.

[GHC01a] Team GHC. Data.either, 2001. URL: https://hackage.haskell.org/package/base-4.16.0.0/docs/Data-Either.html.

[GHC01b] Team GHC. Unsafe io operations, 2001. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/System-IO-Unsafe.html.

[GHC20a] Team GHC. 6.2.13. lambda-case, 2020. URL: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/lambda_case.html.

[GHC20b] Team GHC. 6.2.14. empty case alternatives, 2020. URL: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/empty_case.html.

[GHC20c] Team GHC. 6.4.14. visible type application, 2020. URL: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/type_applications.html.

[GHC21] Team GHC. Monad of no return, 2021. URL: https://gitlab.haskell.org/ghc/ghc/-/wikis/proposal/monad-of-no-return.

[Gil21] Michele Gilman. Ai algorithms intended to root out welfare fraud often end up punishing the poor instead, December 2021. URL: https://theconversation.com/ai-algorithms-intended-to-root-out-welfare-fraud-often-end-up-punishing-the-poor-instead-131625.

[Ham04] Paul Hamill. *Unit Test Frameworks: Tools for High-Quality Software Development*. O'reilly, 2004.

[HC10] Mike Hinchey and Lorcan Coyle. Evolving critical systems: a research agenda for computer-based systems. In *2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, pages 430–435, 2010. DOI: 10.1109/ECBS.2010.56.

[HCE16] Todd M. Herrington, Jennifer J. Cheng, and Emad N. Eskandar. Mechanisms of deep brain stimulation. *Journal of Neurophysiology*, 115(1):19–38, 2016. DOI: 10.1152/jn.00281.2015.

[Hen11] Dennis C. Hendershot. Inherently Safer Design: An Overview of Key Elements. *Professional Safety*, 56(02):48–55, February 2011. ISSN: 0099-0027. eprint: https://onepetro.org/PS/article-pdf/56/02/48/2178900/asse-11-02-48.pdf.

[HGV+18] Alexandria M. Hertz, Evalyn I. George, Christine M. Vaccaro, and Timothy C. Brand. Head-to-head comparison of three virtual-reality robotic surgery simulators. *JSLS : Journal of the Society of Laparoendoscopic Surgeons*, 22(1), 2018. DOI: 10.4293/jsls.2017.00081.

[HLV+16] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luıs Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1), April 2016. ISSN: 0360-0300. DOI: 10.1145/2873052. URL: https://doi.org/10.1145/2873052.

[How80] William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[Hur95] Antonius J. Hurkens. A simplification of girard's paradox. *Lecture Notes in Computer Science*:266–278, 1995. DOI: 10.1007/bfb0014058.

[Hut18] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2018.

[JBC18] Mark P. Jones, Justin Bailey, and Theodore R. Cooper. Mil, a monadic intermediate language for implementing functional languages. *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, 2018. DOI: 10.1145/3310232.3310238.

[JJK+17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. DOI: 10.1145/3158154. URL: https://doi.org/10.1145/3158154.

[Joh05] Jeff Johnson. Process safety since bhopal, January 2005. URL: https://cen.acs.org/articles/83/i4/PROCESS-SAFETY-SINCE-BHOPAL.html.

[Jon95] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. *Advanced Functional Programming*:97–136, 1995. DOI: 10.1007/3-540-59451-5_4.

[Kin21] Alexis King. An introduction to typeclass metaprogramming, March 2021. URL: https://lexi-lambda.github.io/blog/2021/03/25/an-introduction-to-typeclass-metaprogramming/.

[Kli72] Morris Kline. *Mathematical thought from ancient to modern times*. Oxford University Press, 1972.

[KN19] Steve Klabnik and Carol Nichols. Unsafe rust, 2019. URL: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[Lan66] Edmund Landau. *Foundations of analysis: The arithmetic of whole, rational, irrational and complex numbers*. Chelsea Publishing Company, 1966.

[Leh19] Jukka Lehtosalo. Our journey to type checking 4 million lines of python, September 2019. URL: https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. Association for Computing Machinery, New York, NY, USA, 1995. ISBN: 0201119722.

[Lin20] Hsiang-Jui Lin. Type-vec: a type-safe vector with type-level length in rust, 2020. URL: https://github.com/jerry73204/rust-type-vec/.

[LS99] Yanhong A. Liu and Scott D. Stoller. From recursion to iteration: what are the optimizations? *SIGPLAN Not.*, 34(11):73–82, November 1999. ISSN: 0362-1340. DOI: 10.1145/328691.328700. URL: https://doi.org/10.1145/328691.328700.

[Mag18] Sandy Maguire. *Thinking with types: type-level programming in Haskell*. Maguire, 2018.

[Mar10] S. Marlow. Haskell 2010 language report. In 2010.

[MIC+20] Neil P. Morris, Mariya Ivancheva, Taryn Coop, Rada Mogliacci, and Bronwen Swinnerton. Negotiating growth of online education in higher education. *International Journal of Educational Technology in Higher Education*, 17(1), 2020. DOI: 10.1186/s41239-020-00227-w.

[MPG+21] Natarajan Arul Murugan, Artur Podobas, Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Stefano Markidis. A review on parallel virtual screening softwares for high performance computers, 2021. arXiv: 2112.00116 [cs.DC].

[MPŠ+20] Sebastián Makó, Marek Pilát, Patrik Šváb, Jaroslaw Kozuba, and Miroslava Čičváková. Evaluation of mcas system. *Acta Avionica Journal*:21–28, 2020. DOI: 10.35116/aa.2020.0003.

[MT19] Bartosz Milewski and Igal Tabachnik. *Category theory for programmers*. Bartosz Milewski, 2019.

[Mus19] Elon Musk. An integrated brain-machine interface platform with thousands of channels. *Journal of Medical Internet Research*, 21(10), 2019. DOI: 10.2196/16194.

[nLa21a] nLab authors. Absorption magma. http://ncatlab.org/nlab/show/absorption%20magma, November 2021. Revision 3.

[nLa21b] nLab authors. Dependent product type. https://ncatlab.org/nlab/show/dependent+product+type, 2021.

[nLa21c] nLab authors. Dependent sum type. https://ncatlab.org/nlab/show/dependent+sum+type, 2021.

[nLa21d] nLab authors. Ex falso quodlibet. http://ncatlab.org/nlab/show/exfalsoquodlibet, August 2021. Revision 2.

[nLa21e] nLab authors. Monoid. http://ncatlab.org/nlab/show/monoid, June 2021. Revision 45.

[nLa21f] nLab authors. Propositions as types. http://ncatlab.org/nlab/show/propositions+as+types, 2021.

[nLa21g] nLab authors. Unital magma. `http://ncatlab.org/nlab/show/unital%20magma`, November 2021. Revision 6.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[Pau96] Lawrence C Paulson. *ML for the working programmer*. Cambridge University Press, 1996.

[Pet17] Tomas Petricek. PhD thesis, 2017. URL: `http://tomasp.net/academic/theses/coeffects/thesis-final.pdf`.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[PLO21] PLOVER. Cwe-125: out-of-bounds read, July 2021. URL: `https://cwe.mitre.org/data/definitions/125.html`.

[Rin21] Talia Ringer. *Proof Repair*. PhD thesis, University of Washington, 2021. URL: `https://homes.cs.washington.edu/~djg/theses/ringer_dissertation.pdf`.

[rnet] The C++ resources network. Printf. URL: `https://www.cplusplus.com/reference/cstdio/printf/`.

[RPS+19] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. Qed at large: a survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019. ISSN: 2325-1107. DOI: `10.1561/2500000045`. URL: `http://dx.doi.org/10.1561/2500000045`.

[Sak21] Alex Sakharov. Rice's theorem, 2021. URL: `https://mathworld.wolfram.com/RicesTheorem.html`.

[Sco16] Michael Lee Scott. *Programming language pragmatics*. Morgan Kaufmann, 4th edition, 2016.

[Scu01] Scut. Exploiting format string vulnerabilities, 2001. URL: `https://cs155.stanford.edu/papers/formatstring-1.2.pdf`.

[SMM11] Paweł Szymański, Magdalena Markowicz, and Elżbieta Mikiciuk-Olasik. Adaptation of high-throughput screening in drug discovery—toxicological screening tests. *International Journal of Molecular Sciences*, 13(1):427–452, 2011. DOI: `10.3390/ijms13010427`.

[Suz17] Paulo Suzart. Dependent types and safer code, June 2017. URL: `https://paulosuzart.github.io/blog/2017/06/18/dependent-types-and-safer-code/`.

[VSJ+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. *SIGPLAN Not.*, 49(9):269–282, August 2014. ISSN: 0362-1340. DOI: 10.1145/2692915.2628161. URL: https://doi.org/10.1145/2692915.2628161.

[Wad92] Philip Wadler. The essence of functional programming. *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '92*, 1992. DOI: 10.1145/143165.143169.

[Wal17] Erin Walden. Software error affects istep+ math scores across state, June 2017. URL: https://www.newsandtribune.com/news/software-error-affects-istep-math-scores-across-state/article_42f1358c-586e-11e7-acc9-33a8047b1fed.html.

[Web12] Lauren Weber. Your résumé vs. oblivion. *The Wall Street Journal*, January 2012. URL: https://www.wsj.com/articles/%5CnSB10001424052970204624204577178941034941330.

[WKS20] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. July 2020. URL: http://plfa.inf.ed.ac.uk/20.07/.

[WN22] Quinn Wilton and Na. Private correspondence, January 2022.

[XH01] Hongwei Xi and Robert Harper. A dependently typed assembly language. *ACM SIGPLAN Notices*, 36(10):169–180, 2001. DOI: 10.1145/507669.507657.

[Xi17] Hongwei Xi. Applied type system: an approach to practical programming with theorem-proving, 2017. arXiv: 1703.08683 [cs.PL].

[XL20] Ningning Xie and Daan Leijen. Effect handlers in haskell, evidently. *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, 2020. DOI: 10.1145/3406088.3409022.

[Zav21] Vladislav Zavialov. Type families in haskell: the definitive guide, April 2021. URL: https://serokell.io/blog/type-families-haskell.

[ZEE16] Mark Ziemann, Yotam Eren, and Assam El-Osta. Gene name errors are widespread in the scientific literature. *Genome Biology*, 17(1), 2016. DOI: 10.1186/s13059-016-1044-7.