University of Nottingham
UK | CHINA | MALAYSIA

Dependent Types for Compiler Calculation

Thesis submitted to the University of Nottingham for the degree of **Doctor of Philosophy, June 2026.**

Mitchell Pickard

14281173

Supervised by

Graham Hutton

Signature _____

Date _____ / _____ / _____

Abstract

Compilers are the foundation of all modern software, translating high-level code that programmers write into the machine code that computers execute. However, compilers are difficult to write, and difficult to get right. Formal verification of compilers is important, as it allows us to mathematically prove that our compiler performs a correct translation.

This thesis explores a path to compiler verification not through writing a compiler and verifying it after-the-fact, but by *calculating* a compiler directly from its specification, which necessarily gives us a proof that the compiler meets this specification. We present a refined methodology for this based on dependent types, which allows us to consider typed source and target languages, and aids in the process of mechanising the calculation. We then extend the methodology to support the use of the partiality monad, which allows us to consider non-terminating languages.

We develop the methodology gradually, starting with a simple expression-based language to show the core foundation of the technique, then adding non-local control flow with exceptions, and calculating a compiler for the simply-typed lambda calculus. Finally, we calculate a compiler for a language with a non-terminating loop construct, using the partiality monad.

Table of contents

Chapter 1

Introduction

Compilers are a fundamental tool underpinning almost all software. We often assume they are correct, and write all of our code based on these assumptions, but what if we are wrong? Compiler bugs can cause subtle but hard-to-find problems in our software.

However, compilers are complex programs which underpin almost all of the software that is written, and verifying their correctness can be a mammoth task. This thesis will explore the history of compiler verification, and present a new methodology based on dependent types, to calculate verified correct compilers for typed languages directly from the specification of their source languages.

This thesis is aimed at readers with an understanding of functional programming, but not necessarily knowledge of dependent types or Agda. All aspects of dependent types and features of Agda that are used will be explained.

1.1 What is a compiler?

For a program written in some programming language to be executed, it must be translated to a set of lower-level instructions known as machine code, which processors understand. There are two types of tools for doing this translation: interpreters and compilers. Interpreters run when the user actually wants to run the program: they process the source code during program execution. In contrast, compilers fully process and transform the source code ahead-of-time, producing as an output executable code that is independent from the source code of the program, and from the compiler itself.

The output of a compiler can be machine code that can be directly executed on a real machine. Alternatively, the output may be instructions for an abstract machine that does not correspond directly to hardware, or source code for another programming language. In this thesis, we consider compilers which produce code for abstract machines.

As an example demonstration of a compiler, consider the follwing expression, containing a reference to a variable x and some arithmetic operators:

$$(x + 5) * 7$$

A compiler might translate this to a sequence of machine instructions:

```
load 0x4000   ; load the value of x (at memory location 0x4000)
              ; into the accumulator
add acc, 5    ; add 5 to the accumulator
mul acc, 7    ; multiply the accumulator by 7
```

Even for this simple example, the compiler has had to perform a number of functions, including determining the order of operations and assigning memory locations to variables.

Usually, the first step a compiler performs is lexical analysis, also known

as “lexing”, “scanning”, or “tokenisation”. This takes the unstructured sequence of characters in a file, and combines them into tokens, which represent atomic syntactic units of the program. A token will typically be a keyword, operator, identifier, or semantic whitespace/indentation. At this stage, the tokens are still a flat list without additional structure, but simplifies the next step of the process.

The next step is parsing, which takes this flat list of tokens, and determines the syntactic structure of the program, such as functions, blocks of code, expressions, and operator precedence. This information is stored in an abstract syntax tree: a nested, structured representation of the program with information about what language constructs are used and how they are combined.

Compilers also perform various checks, either on the abstract syntax tree of the source program, or at a later stage on the intermediate representations. For example, scope checking makes sure that all variables are accessible from the point in the program where they are referenced, while type checking ensures that the language's typing rules (if any) are followed. Type checking can be as simple as checking that basic data types like integers and strings are handled properly, or as complex as dependent type systems that allow arbitrary (terminating) computation.

The core part of the compiler's job is code generation: translating this source abstract syntax tree into the target language. This step can vary a lot between compilers and is intertwined with various steps of checking and optimisation, but will typically involve a number of translation/simplification steps, often through one or more intermediate languages.

Optimising compilers aim to not just do a direct translation, but to do the translation in a way that minimises or maximises certain aspects of the resulting program. Common aspects to optimise are the runtime speed, memory usage, and size of the code produced.

If the compiler is generating machine code rather than a language which is intended to be processed further, it will include additional architecture-specific code generation stages, such as allocating registers and determining the memory layout of program structures.

The compiler can also perform additional functions. It may be responsible for combining multiple files from a project or libraries, or invoking another program called a linker to do this. Compilers also produce warnings and errors for the programmer, helping them debug problems and improve their code, and they may also provide an interface for assisting the programmer during writing the code, or a live environment to test code line-by-line in a read-eval-print loop (REPL).

1.2 Why compiler correctness matters

Modern compilers are large and complex programs, and like any other large programs, are constantly having bugs found and fixed (Sun et al., 2016). Despite this, bugs that lead to incorrect programs are rare, and most programmers can go throughout their day assuming the compiler is correct.

A correct compiler is one which preserves the semantics of the source program in the resulting program. For example, suppose the semantics of the source language is specified by an interpreter, and the semantics of the target language is specified by an abstract machine. Then, a correct compiler is one where any compiled program exhibits the same behaviour when run in the abstract machine as the source program run in the interpreter.

However, semantic bugs in compilers are not unheard of, and can have serious consequences—no matter how perfect your source code is, if your compiler emits incorrect code, your program might not behave correctly. This is much more serious than a bug in a single program, because ev-

ery program produced by an incorrect compiler has the potential to be incorrect—one compiler bug can infect thousands of programs.

The problem is compounded by the fact that compilers often compile other compilers. When a compiler is written in a high-level language, it needs to be compiled wth a compiler itself in a bootstrapping process. This means that there is typically a chain of compilers going all the way back to a hand-written assembly compiler. As discussed in Thompson's Turing Award paper (Thompson, 1984), a problem in any one of these steps may persist, potenitally infecting any compiler later on in the chain.

1.3 Approaches to compiler correctness

Bugs in compilers can be extremely tricky to track down, as they can be the result of many complex optimisation passes interacting, and may only manifest themselves in obscure usage of the language or rare runtime situations (Sun et al., 2016).

There are many different approaches to identifying and preventing compiler bugs. As compilers are often large pieces of software, standard software testing practices can be applied, both automated and manual. Testing techniques include unit testing, integration testing, and end-to-end tests (Ammann and Offutt, 2016).

Fuzzing is a software testing technique where random inputs are generated, including invalid and unexpected inputs, and compared to expected outputs (Liang et al., 2018). Fuzzing has been sucessfully applied as a means of testing compilers. A recent survey showed how compiler fuzzing techniques have been used to uncover numerous bugs in widely-used compilers, which manifested as bugs in and test suite failures when used to compile popular software packages (Marcozzi et al., 2019).

However, as famously remarked by Dijkstra, "Program testing can be used to show the presence of bugs, but never to show their absense!" (Dijkstra, 1970) If we want our compilers to be guaranteed free of bugs, we need to mathematically prove that this is the case. Chapter 2 gives an overview of the history of compiler verification.

Verification is often performed post-hoc, once a compiler is written. In contrast, program calculation, which is the focus of this thesis, aims to *calculate* a program (in this case, a compiler) from its specification, producing simultaneously the prgram itself and the proof of its correctness. The calculation is driven by a correctness theorem which guides the development of the compiler, and gives a principled means of discovering what the resulting program will look like. An overview of compiler calculation is given in Chapter 3.

1.4 Overview of the thesis

Part I: Background

Chapter 2 introduces the field of compiler verification, outlining the history and methods used to verify compilers. It starts with McCarthy and Painter's (1967) seminal compiler verification, through to modern projects such as CompCert (Leroy, 2009)'s fully verified optimising C compiler.

Chapter 3 introduces the compiler calculation methodology that the rest of the work in this thesis builds upon. This chapter reviews the literature of the method, and gives a high-level overview of the technique.

In Chapter 4 we give an overview of dependent types, for readers who know a functional programming language such as Haskell, but are not familliar with dependent types. We give an overview of the type theory and the various constructs which are available, and how they translate to

the dependently-typed language and proof assistant Agda.

Agda is used throughout the rest of this thesis as both the implementation language of the compilers, and the proof assistant used to verify the calculations. All of the Agda concepts needed to understand this thesis will be introduced, so no prior knowledge is needed.

Part II: Methodology and applications

Part II begins with Chapter 5, which demonstrates the methodology by calculating a dependently-typed compiler for a simple expression language. The results are contrasted with the non-dependent calculation, showing the benefits of adding dependent types to the methodology.

Chapter 6 extends this simple expression language with exceptions, showing how the technique can be used to handle non-local control flow, and how the use of dependent types is beneficial for keeping track of the stack.

Chapter 7 explores a more real-world example language, by presenting a calculation of the simply-typed lambda calculus. However, this requires a more complex proof based on relational semantics to prove to Agda that the proof is indeed terminating.

Chapter 8 solves this need for the relational-style proof by merging the prior technique with the method presented in Bahr and Hutton (2022), on using coinduction for compiler calculation. This also allows us to compile languages with non-termination, giving us a solid base upon which a greater variety of languages could be handled by the technique.

Part III: Conclusions

To conclude, Part III gives a summary of the techniques developed and compilers calculated in the thesis, and outlines the benefits of the technique over previous methods. It then outlines potential areas of further work on generalising and mechanising our technique.

Part IV: Appendices

All of the Agda code used in this thesis, for both the compilers and the calculations, is included in Appendix A.

1.5 Contributions

This thesis makes the following contributions:

- Firstly, Chapter 5 shows how Bahr and Hutton's (2015) compiler calculation methodology can be generalised to a dependently-typed setting. In particular, we show that the use of dependent types allows us to naturally support typed source languages, ensure that all compilation components are type-safe, and makes the resulting calculations easier to mechanically verify.
- In Chapter 6 we show how this extended methodology allows a compiler for a language with exceptions to be calculated. Previously it had proven difficult to even write a dependently-typed compiler for such a language, but the methodology helped discover how this could be done.
- To further demonstrate the generality of the technique, in Chapter 7 we calculate a compiler for the simply-typed lambda calculus. The

use of dependent types allows the types of lambda abstraction, function application, and variable environments to be precisely specified.

- Finally, we show how we can extend the technique with coinductive reasoning, allowing us to handle typed, non-terminating languages. This is demonstrated by calculating a compiler for a non-terminating language with a looping construct. An alternative mechanisation of the simply-typed lambda calculus calculation is performed with this technique, with the code in the appendix.

The content of chapters 5 and 6 was published previously, in Pickard and Hutton (2021). I was the lead author on this paper and presented the work at the International Conference on Functional Programming.

Part I

Background

Chapter 2

Compiler verification

What does it mean for a compiler to be "correct"? In essence, we want to ensure that the semantics of the source language are preserved through compilation to the target language, so the program does exactly what the programmer expects. We can do this by having a specification of the source and target languages, and ensuring that, for every possible input, the compilation from source to target language gives the same result according to the specifications.

Compiler verification is an important and varied topic, with a long history. This chapter will survey some of the most important papers in the field; a more comprehensive historical account of works on compiler verification can be found in Dave (2003).

2.1 Early work

McCarthy and Painter (1967) is the seminal paper in the field of compiler verification, demonstrating that it is possible to formally verify a compiler. The paper considers a minimal language, consisting of constants, variables, and a single binary operator. The compilation target is a register-based

machine with an accumulator register, *ac*, and numbered registers. A subset of these numbered registers is reserved for variables, with an unlimited number of temporary registers available for use during execution. The result of executing the program will be stored in the accumulator register.

The compiler correctness theorem is stated as follows:

$$outcome(compile(e, t), \eta) =_t a(ac, value(e, \xi), \eta),$$

where *compile* is a function which compiles a source-language expression, e, into a machine-language representation; *outcome* is a function which returns the result of executing the given machine code on a machine state η; $a(x, \alpha, \eta)$ denotes the state obtained by replacing the value of variable x with the value α in the state η; and $value(e, \xi)$ denotes the result of expression e evaluated in the state ξ according to the source-language semantics.

$=_t$ is a partial equality on machine states, holding true if and only if the values of all numbered registers less than t are equal. This allows the correctness theorem to disregard the state of temporary registers which are used during the execution.

The compiler correctness theorem states that the compiler is correct if, for all possible expressions to be compiled, the result of executing a compiled program is a machine state where the variable registers less than t are unchanged from the initial state of the machine, and the accumulator stores the value which is obtained by evaluating the source expression directly. Verification is performed by induction over the expression being compiled.

The concepts introduced by McCarthy and Painter are similar to that which are used throughout the rest of this thesis, with the correctness theorem essentially following the same form. However, we will target a stack machine supporting multiple types, rather than the unityped register machine presented by McCarthy and Painter.

Kaplan (1967) proves the correctness of a compiler for an ALGOL-like

language, augmenting the previous example of arithmetic expressions with variable assignments, conditional statements, and non-linear control flow with arbitrary jumps. They note that it would be straightforward to extend the source language with a larger set of arithmetic and logical operations, and that higher-level control statements such as `while` could be specified in terms of the primitive jumps which have already been proven correct. Painter (1967) gives a similar verification of a compiler for an ALGOL-like language with arbitrary jumps.

Burstall and Landin (1969) explored the use of algebraic approaches in compiler verification, in particular defining functions as homomorphisms instead of giving recursive or iterative definitions. Algebraic approaches for compiler verification were further explored in Morris' (1972) thesis. London (1971, 1972) showed that even in the early days of the field compiler verification techniques could be applied to realistic langugaes, proving the correctness of two compilers for a subset of LISP.

2.2 Machine-checked proofs

Prior work was mostly limited to manually-verified compiler proofs, but was building towards being able to mechanically verify a proof for a correct compiler. Diffie (1972) was the first to carry out such a machine-checked proof, using a proof-checking program for the First Order Predicate Calculus to verify McCarthy and Painter (1967)'s compiler.

Milner and Weyhrauch (1972) demonstrated a machine-checked proof for a substantial part of a compiler for an ALGOL-like language with assignments, conditionals, and non-linear control flow. They opted for structured `while` loops, in contrast to the the arbitrary jumps used in the prior examples by Kaplan and by Painter, embracing the 'goto-less' programming style advocated by Dijkstra (1968). The proof is done using a mechanised ver-

sion of a logic for computable functions, LCF (Milner, 1972a,b; Weyhrauch and Milner, 1972).

Hannan and Pfenning (1992) show how a compiler can be verified within the LF Logical Framework and the Elf programming language. They verify a compiler for a simple functional programming language to an abstract machine. In their system, compiler correctness can be mechancially checked, while verification of totality needs to be done by hand.

Necula and Lee (1998) take a different approach to verifying the results of compilation. Instead of verifying the compiler itself, they augment a compiler for a type-safe subset of the C programming language with a certifier that verifies the type-correctness of the resulting assembly code. This is not a full verification, as the compiler could still produce results that do not match the specification of the lanugage but are type-safe, but it is generally easier to verify a specific result of a computation than to verify the computation itself. It also allows the compiler to be much more agrressive with optimisations, as each of the optimisation steps does not need to be verified, just the final output. This also demonstrates the advantages of involving types in the verification process.

There also exist several formally-verified compilers targeting real-world languages. CompCert (Leroy, 2009) compiles a subset of C, and is verified in the Coq proof assistant, using Coq's code extraction feature to produce the compiler binary. CakeML (Kiam Tan et al., 2019) is a compiler for a language of the same name, based on a subset of Standard ML. The CakeML language is specified in HOL, the compiler is written in HOL, and the compiler code can compile itself, allowing for a formally-verified compiler bootstrapping. Both of these compilers use multiple intermediate languages and contain many passes, including optimisations. The DeepSpec project (Appel et al., 2015) aims to develop a fully-verified computing toolchain, spanning the full spectrum from programming language compilers through to operating systems and processors. The aim of this thesis

is quite different to these projects—we are seeking to develop principled techniques for calculating compilers in a systematic manner, rather than trying to build real-world verified compilers.

2.3 Dependent types

This thesis primarily explores the role and benefits of dependent types in compiler verificaiton strategies. There is some existing literature in this area.

Brady and Hammond (2006) show an approach to using dependent types for compiler verification. Rather than building a compiler by treating an interpreter as a semantics, they directly modify this interpreter, by adding staging annotations to determine which parts of the code are run at compile-time and which at run-time, to generate a compiler directly from the interpreter. As the interpreter is verified with the use of dependent types, the resulting compiler is also verified.

McKinna and Wright (2006) show how dependent types can be used to verify a compiler, using the dependently-typed language Epigram. They define an intrinsically-typed representation of expressions, where the Epigram type of an expression carries the the source-language type with it, allowing functions and proofs to depend on the underlying type of the expression.

The source language they consider is as follows. Note that the *Exp* type carries around its type with it, which is exploited in the *eval* function. The return type of recursive calls to *eval* does not need to be checked, as it is known statically from the type of the expressions being passed around.

```
data TyExp : * where
  nat  : TyExp
  bool : TyExp
```

```
data (T : TyExp) → Exp T : * where
  val  : (v : Val T) → Exp T
  plus : (e1 : Exp nat) → (e2 : Exp nat) → Exp nat
  if   : (b : Exp bool) → (e1 : Exp T) → (e2 : Exp T) → Exp T


eval : (e : Exp T) → Val T
eval e ⇐ rec e
eval (val v) ⇒ v
eval (plus e1 e2) ⇒ (eval e1) + (eval e2)
eval (if b e1 e2) ⇒ cond (eval b) (eval e1) (eval e2)
```

This intrinsically-typed representation is the foundation for representing the source languages throughout this thesis. A more detailed explanation of the dependent types used here is given in Chapter 4.

Chlipala (2007) developed a certified compiler from the lambda calculus to assembly language, using a series of compilers between intermediate languages. The representation of each of the intermediate languages is dependently-typed, ensuring that at each stage only well-formed and well-typed programs can be represented.

McBride (2011) extended a language of arithmetic expressions with ornaments, representing the types and semantics of expressions. These ornaments are then used to automatically derive a correctness proof for a compiler. However, he notes that a more complicated language with extra control flow, such as conditional expressions, would require extra manual proofs to be added.

Pardo et al. (2018) apply a similar methodology to McBride to derive compilers for more complicated source languages, including an imperative language with loops and mutable assignment. They perform verification by internalising the evaluation semantics into an index of the expression type, so much of the correctness proof is automatically derived by the type-checker.

However, as with McBride's work, this is a verification of a compiler, not a calculation.

More recently, Rouvoet et al. (2021) showed how dependent types can be used to ensure that jump labels are always used in an appropriate manner in a well-typed compiler. This is done by using linear, dependent types to ensure that labels are only defined once, and defining labels in a nameless manner that allows proof terms to remain compositional.

Chapter 3

Compiler calculation

Compilers are large and complicated tools, and writing and subsequently verifying them can be a significant challenge. This thesis explores compiler calculation, an alternative method to producing correct compilers. The method that we consider starts with a specification of what it means for a compiler to be correct, and derives a correct-by-construction compiler directly from the specification. This chapter surveys relevant literature, starting with the early work in the field, then considering Bahr and Hutton's approach on which this thesis is based, and finally other approaches to compiler calculation.

3.1 Early work

The idea of deriving a compiler was first considered by Wand (1982a), building on the pioneering work of Reynolds (1972) on continuations and defunctionalisation. Wand did not originally consider the issue of correctness, but later showed how the resulting compilers could also be proved correct (Wand, 1982b). In his PhD thesis, Meijer (1992) developed an algebraic approach to compiler derivation, based on the use of fold operators (Meijer et al., 1991). However, using fold operators complicates

the methodology, and his approach requires significant up-front knowledge about the behaviour of the final compiler.

For many years, partial evaluation (Jones et al., 1993) has also offered the prospect of automatically producing compilers from interpreters by applying a partial evaluator to itself (Futamura, 1999). However, there are many challenges that arise in realising this transformation, as explored in the long-standing workshop series Partial Evaluation and Program Manipulation.

More recently, Danvy et al. (2003) developed an approach in which Reynold's techniques are first used to derive an abstract machine, which is then factorised into a compiler and a virtual machine. This work assumes that all the transformations are semantics preserving, but does not provide a formal proof of compiler correctness.

Despite significant advances, the idea of deriving compilers in a principled way from semantics had remained notoriously challenging, and none of the approaches that have been developed had yet given rise to practically useful methodologies. As a result, the field of compiler derivation had not received much attention in recent years, until it was reawakened in Bahr and Hutton's work.

3.2 Bahr and Hutton's approach

Bahr and Hutton (2015) presented a new approach to developing compilers whose correctness is guaranteed by the manner in which they are constructed. The basic procedure is as follows. It begins by defining the syntax of the source language, and a semantics for this language. The aim then is to define three further components: the syntax of the target language, a semantics for this language, and a compiler that translates from the source to the target language. The desired relationship between the

components is captured by a compiler correctness theorem that formalises the intuitive idea that compiling a source program does not change its semantics.

Given such a setting, one then attempts to calculate suitable definitions for the additional components, in much the same way as one calculates solutions to equations in mathematics. Using the approach developed by Bahr and Hutton (2015) we can use elementary reasoning techniques to permit the calculation of compilers for a wide range of source language features and their combination. An introduction to the approach is given later on, in Chapter 5.

Bahr and Hutton's methodology was originally developed for stack-based target languages, but has also been extended to register-based languages (Hutton and Bahr, 2017; Bahr and Hutton, 2020). More recently, the approach has been extended to consider non-terminating and concurrent source languages (Bahr and Hutton, 2022, 2023) and graph-based target languages (Bahr and Hutton, 2024).

3.3 Other approaches

Elliott (2017) shows how category theory can be used as an intermediate stage of compilation for functional languages based on the simply-typed lambda calculus. The compilation target can then be defined simply by implementing a categorical interface for this target, ensuring that the laws corresponding to the chosen categorical interface are satisfied. Elliott (2018) then shows how these categorical laws can be used as the specification for a compiler, allowing the compiler to be calculated in a similar way to the compilers in this thesis. He shows how to calculate a compiler targeting a stack machine supporting primitive functions and simple stack operations using suitable categorical laws.

Gibbons (2022) revisits Wand's approach to compiler derivation in a modern setting. Gibbons begins with an evaluation semantics, and transforms it using continuation-passing style and defunctionalisation to derive an abstract machine. This is then transformed into a compiler by exploiting associativity of a generalised function composition operator. The key contributions of this work are bringing associativity to the forefront, and showing how the generalised function composition operator can naturally be implemented in the dependently-typed language Idris (Brady, 2013).

Chapter 4

Dependent types

This chapter begins with an overview of dependent types, explains why they are useful, and introduces common concepts in dependent types that will be used in the rest of this thesis. It then gives a brief overview of the dependently-typed programming language Agda, which is used throughout the rest of this thesis for both writing the compilers, and for verifying the calculations.

No specific concepts for the compiler calculation methodology are introduced in this chapter, so it can be safely skipped by readers who are already familliar with dependent types and Agda.

4.1 What are dependent types?

Type systems allow us to encode invariants in programs, which are checked by the compiler. Types typically occupy a completely separate space to program values, existing only at compile-time. Dependent type systems extend these type systems by allowing types to depend on values in the program.

The extra power of dependent type systems allows us to be more precise

about our types. We can express a wider range of invariants about our program, such as restricting the input domain, or enforcing a condition on the output. It also allows us to avoid the use of partial functions which may cause errors at runtime, or verify that functions satisfy certain conditions. Dependent types can be used to specify predicates and proofs, so we can write proofs about programs in the same language that the programs are written in. These proofs will be checked by the compiler, allowing us to prove properties for all inputs of a program which may otherwise only be tested for a limited subset of inputs, or would require external proofs to be written.

Another benefit of dependent types is that they give the compiler more information to work with, even before the program is completed. Many dependent type systems such as Agda and Idris allow an interactive development experience, with the programmer leaving unfilled holes in the program, and querying the compiler for information about the type of the hole and relevant in-scope definitions, and in some cases even filling in the whole automatically when it is suitable to do so. This helps guide the programmer towards the creation of a correct program, increasing their productivity and their confidence in the correctness of their output.

A common example of a dependent type is the `Vector` type. This is analagous to a `List` type storing a collection of values, which also stores the length of the list in the type. This allows us to write functions that express properties relating to the length of input or output lists, such as a precondition that input lists are non-empty, or an invariant that a sorting algorithm will return a list of the same length as the input.

The other view of dependent types is as a mathematical foundation. We can express a predicate as a type, and prove this predicate by writing a program that fits the given type. Using types as this basis allows us to write a type-checker that acts as a mechanical proof assistant. If a written proof does not type-check for a given predicate, the proof assistant will

reject the proof.

In this thesis, we will take advantage of both of these aspects of dependent types. The compilers we calculate will use dependent types to ensure the functions are total, and the proofs will be mechanically verified. We use the programming language Agda both as the language the compilers are implemented in, and as the proof assistant for verifying of the proofs.

4.2 Non-dependent types

We'll begin by defining some basic non-dependent types, which are expressible in a simple type system.

Top and bottom

The unit type, top, or $\top$, is the type inhabited by a single element. This is equivalent to a trivially true proposition, as there is always an element of this type. The following declaration says that the type of $\top$ is $Type$, which is the type that types have. It then says that there is a single element tt of type $\top$.

$$\top : Type$$
$$tt : \top$$

We can also define the empty type, bottom, or $\bot$. This has no elements, so represents a false proposition: we can never construct an element of this type.

$$\bot : Type$$

Sums and products

A sum type combines two types, such that an element of the sum $A + B$ is either an element of A, or an element of B. For any two types A and B, we can define the sum $A + B$, with a constructor for the *left* type A, and one for the *right* type, B. The arrow $\rightarrow$ is a function type, which will be defined later in this section.

$$\begin{aligned} A + B &: Type \\ \mathit{left} &: A \rightarrow A + B \\ \mathit{right} &: B \rightarrow A + B \end{aligned}$$

Product types also combine two types, but require an element of *both* types, rather than one or the other. For two types A and B:

$$\begin{aligned} A \times B &: Type \\ \mathit{pair} &: A \rightarrow B \rightarrow A \times B \end{aligned}$$

Inductive types

Many types can be constructed using the basic building blocks we have just defined: $\top$, $\bot$, $+$, and $\times$; for example the sum type $\top + \top$. This type has two members: $\mathit{left}(\mathit{tt})$ and $\mathit{right}(\mathit{tt})$. Having two members, we can use this as a boolean type by attributing a true value to one element and a false value to the other element.

However, we can also define types directly by specifying their elements; for

example, a direct definition of the booleans:

$$Bool : Type$$
$$true : Bool$$
$$false : Bool$$

Inductive types, as the name suggests, allow us to define types inductively, constructing types with an infinite number of elements. Natural numbers are an example of an infinite type: they are either *zero*, or the successor to some other natural number (written as *succ*). For example, two can be represented as $succ\ (succ\ zero)$. We can define recursive types by referring to the type being defined in the type of one or more of the constructors.

$$\mathbb{N} : Type$$
$$zero : \mathbb{N}$$
$$succ : \mathbb{N} \to \mathbb{N}$$

Functions

Functions define a relation between two types, mapping every element of the input type to an element of the output type. Partial functions are not allowed in some dependent type systems, such as Agda—the function must be defined for every element of the input type.

For example, we can define a function that maps Booleans to natural numbers, by providing a definition for the function applied to each Boolean

value:

$$
\begin{aligned}
f &: \mathit{Bool} \to \mathbb{N} \\
f\ \mathit{false} &= \mathit{zero} \\
f\ \mathit{true} &= \mathit{succ}\ \mathit{zero}
\end{aligned}
$$

We can also use lambda notation to define functions, where the input is bound as a variable, and can be used in the result of the function. The simplest example of this is the identity function, which just returns its input.

$$
\begin{aligned}
&\mathit{id} : \mathbb{N} \to \mathbb{N} \\
&\mathit{id} = \lambda(x : \mathbb{N}).\ x
\end{aligned}
$$

4.3 Dependent types

So far all of the types we have introduced have been non-dependent types, expressible in a simple type system. We will now introduce some dependent types.

Dependent functions

The first dependent type we introduce is the dependent function, also known as a Pi-type. This is analagous to the standard non-dependent function $\to$, but the *type* of the output depends on the *value* of the input.

To introduce the dependent function type, it will be helpful to have a regular function that operates on types. This function just maps the values of *Bool* to different types. In dependently-typed languages, types are first-class: we can write functions that operate on and return types just as if

they were regular values.

$$
\begin{aligned}
&F : Bool \to Type \\
&F\ true \quad = \top \\
&F\ false \quad = \mathbb{N}
\end{aligned}
$$

This function F can be used in the following dependent function, which takes an input b of type $Bool$, and returns a value of type $F\ b$. That is, the return *type* is dependent on the input *value*. In the $true$ case, the return type is $F\ true = \top$, and in the $false$ case, the return type is $F\ false = \mathbb{N}$. If, for example, both of the cases below returned tt, it would not type check as the type of the *false* case would be wrong.

$$
\begin{aligned}
&f : \prod_{b:Bool} F\ b \\
&f\ true \quad = tt \\
&f\ false \quad = succ\ zero
\end{aligned}
$$

Previously, when we defined the identity function, it only worked on a single type, and a new function would need to be written for each type we wanted to operate on. With dependent types, we can implement a polymorphic form of the identity function which works on any type. The first argument to this function is the type we want to operate on, and the second argument is returned unchanged.

$$
\begin{aligned}
&id\ : \prod_{A:Type} \prod_{x:A} A \\
&id\ = \lambda(A : Type).\ \lambda(x : A).\ x
\end{aligned}
$$

Dependent functions are a generalisation of non-dependent functions, and we can implement the latter in terms of the former, by ignoring the input value in the return type. For example, the function type $A \to B$ can be

written as:

$$\prod_{x:A} \lambda(x : A).\ B$$

Sum types can also be written in terms of dependent functions, with a function that chooses which type to return based on a Boolean value. The type $A + B$ is equivalent to

$$\prod_{b:Bool} \lambda\ \{true \mapsto A; false \mapsto B\}$$

Dependent products

As their name suggests, dependent products, or dependent pairs, are the dependent versions of the product type. As we have seen, the non-dependent product type, $A \times B$, is inhabited by an inhabitant of A, and an inhabitant of B.

The dependent product generalises this, to allow the type of the second element to depend on the value of the first. In the following example, p is a pair with two elements: x of type A, and the type returned by B applied to x.

$$p : \sum_{x:A} B\ x$$

As with depdendent functions, we can implement non-dependent products in terms of dependent products. The type $A \times B$ is equivalent to a dependent pair where the type of the second element ignores the value of the first element:

$$\sum_{x:A} \lambda(x : A).\ B$$

Inductive families

Inductive families extend the idea of inductive types by allowing them to be indexed by a value. This generates a *family* of types, one for each possible value of the index.

A classic example of inductive families is the *Vector* type, which is a list indexed over its length. Inductive families can be defined recursively, just like inductive types. The following definition defines the *Vector* type, as well as two constructors: *nil* (the empty *Vector*, of length zero), and *cons* (prepend an element to an existing *Vector*, increasing its length by one).

$$
\begin{aligned}
&\mathit{Vector} : \mathit{Type} \to \mathbb{N} \to \mathit{Type} \\
&\mathit{nil} \quad : \prod_{A:\mathit{Type}} \mathit{Vector}\ A\ \mathit{zero} \\
&\mathit{cons} \quad : \prod_{A:\mathit{Type}} \prod_{n:\mathbb{N}} A \to \mathit{Vector}\ A\ n \to \mathit{Vector}\ A\ (\mathit{succ}\ n)
\end{aligned}
$$

4.4 Propositions as types

Propositions can be represented as types in a type theory (Wadler, 2015), with inhabitants of the types being proofs of the propositions. This allows us to encode proofs in the same programming language we use to write programs. To specify a theorem, we write a type, built up from the basic blocks explained earlier in this chapter. For example, to specify the theorem $\forall P.\ P \implies P$, we would write the type $\prod(P : \mathit{Type}).\ P \to P$. To prove a theorem, we construct an element of the corresponding type. For example, to prove the previous theorem, we could specify the element $\lambda(P : \mathit{Type}).\ \lambda(x : P).\ x$. The following table gives the equivalences between concepts from logic, and their type-theoretical interpretations:

Logic	Type theory
Propositions	Types
Proofs	Programs
True	$\top$
False	$\bot$
$A \Rightarrow B$	$A \to B$
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$
$\forall x \in A.\ P(x)$	$\prod_{(x:A)} P(x)$
$\exists x \in A.\ P(x)$	$\sum_{(x:A)} P(x)$
$\neg A$	$A \to \bot$

If we wish to disprove a theorem, we simply construct an element of the theorem's negation. $\neg P$ is defined as $P \to \bot$, that is, if P were true, we could prove falsehood. Or, the alternative interpretation in type theory says that if we have an element of P, we could obtain an element of any type, using the rule $absurd : \forall A.\ \bot \to A$.

4.5 Equality

When writing proofs, we often want to prove that one thing is equal to another. This is done in type theory with an equality type. If we have an inhabitant of the type $x \equiv_A y$, that is a proof that x and y, both of type A, are equal. This allows us to substitute one for another in proofs.

Here we use homogeneous equality, in which both arguments have to be the same type. The first argument is the type at which the equality operates, and the second and third arguments are the values which are to be proved equal.

$$\equiv : \prod_{A:Type} \prod_{a,b:A} Type$$

In simple dependent type theory, the only possible inhabitant of $x \equiv_A y$ is *reflexivity*: that is, any given element is equal to itself.

$$\mathit{refl} : \prod_{A:\mathit{Type}} \prod_{x:A} x \equiv_A x$$

Other type theories, such as Homotopy Type Theory (Univalent Foundations Program, 2013) allow non-trivial forms of equality with structure beyond simple reflexivity, but that is outside the scope of this thesis.

4.6 Agda

Agda (Norell, 2007) is a programming language and proof assistant which implements dependent type theory. It is the language used throughout this thesis: for the specification of the languages to be compiled, the specification of the correctness equations, the implementation of the compilers, and the proofs that the compilers are correct.

Agda is similar in syntax to the general-purpose functional programming language Haskell. There are a few key syntax differences between Haskell and Agda, which are worth pointing out explicitly for readers familliar with Haskell but not Agda. First of all, the symbols `:` and `::` are interchanged: `x : A` means '`x` is of type `A`', and `x :: xs` means 'the list with `x` as head and `xs` as tail'. Secondly, capitalisation has no syntactic meaning in Agda; the convention is that types are capitalised while values are not, unlike in Haskell where constructors must be capitalised.

Types and universes

In Agda, the type of types is called `Set`. To prevent logical inconsistencies arising from `Set` having itself as its type, the type of `Set` is `Set₁`. In turn,

`Set₁ : Set₂`, `Set₂ : Set₃`, etc. forming an infinite hierarchy of universe levels, where $\texttt{Set}_i : \texttt{Set}_{i+1}$.

Data types

We can define inductive data types as follows:

```
data Bool : Set where
  true : Bool
  false : Bool
```

This defines a new type, `Bool`, and two elements of this type, `true` and `false`. We can also define constructors which accept arguments. For example, we could define the natural numbers:

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

The constructor `succ` has a function type, such that given a `ℕ`, it will return a value of `ℕ`. We can pattern match on this constructor to recover the `ℕ` that it was constructed with, as in the predecessor function:

```
pred : ℕ → ℕ
pred zero = zero
pred (succ n) = n
```

This requires adding a case for `zero`, to ensure the function is total.

Inductive families

We can also define inductive families in Agda, such as in the definition of the `Vector` type:

```
data Vector (A : Set) : ℕ → Set where
  [] : Vector A zero
  _::_ : (x : A) → (n : ℕ) → Vector A n → Vector A (succ n)
```

This defines a family of types, rather than just a single type. Vectors are indexed by their length of type `ℕ`, and there is one type for each element of `ℕ`. In this case, it is an infinite family, as `ℕ` has an infinite number of inhabitants.

Implicit arguments

As arguments often depend on other arguments, it can get unwieldy to explicitly pass every argument at every function call site. Often, some variables don't need to be passed explicitly, as they can be inferred from the context, such as inferring the type of a variable, or the length of a list.

In Agda, implicit parameters to functions and constructors are indicated by surrounding them with curly braces, as in `{A : Set} → (x : A) → A`. In this example, the first parameter can be inferred from the type of the second, so it can be omitted when calling the function. Implicit arguments can be passed explicitly by surrounding the argument with curly braces.

We can also define implicit variables that range over the scope of a single file, by using a `variable` block. For example:

```
variable
  A : Set
```

This defines `A` as a generalised variable, which automatically adds `{A : Set}` as an implicit parameter wherever `A` is used, such as turning `f : A → A` into `f : {A : Set} → A → A`.

Mixfix operators and Unicode

Agda varies syntactically from a lot of other languages, in that it allows user-defined arbitrary mixfix operators. Operators are defined by defining a function, where underscores in the name represent the positions of

arguments. For example,

```
_⊗_ : ℕ → ℕ → ℕ
x ⊗ y = x
```

defines a binary infix operator `⊗` which takes two natural numbers and returns a natural number. As well as infix operators, this can be used to define prefix, postfix, or other arbitrary mixfix operators, and is not limited to two arguments.

Agda also fully supports Unicode. All Unicode symbols that are used in this thesis, such as the function arrow `→` and the operator `⊗`, are valid Agda code.

Indexed stack

As a more complex example of an Agda data type, we will introduce the indexed stack type, which will be used later in the complier calculations. At a high level, this is a heterogeneous list of elements, where the types and ordering of the individual elements is specified in the type of the stack. Thus, a `Stack S` has elements with types described by the list `S`.

The `List` type is an ordered, homogeneous list, defined in the Agda standard library. Without providing the full definition, it is constructed from the empty list `[]`, and the cons operator `::`. Consider the following examples:

```
1 :: 2 :: 3 :: [] : List ℕ
ℕ :: Bool :: [] : List Set
```

The first is a list containing the first three natural numbers, and the second is a list containing two types: `ℕ` and `Bool`. Being in a dependently-typed setting is what allows us to store types in a list, just as we would with numbers.

We can define a stack indexed by a list of types:

```
data Stack : List Set → Set where
  ε : Stack []
  _▷_ : {T : Set}{S : List Set} → T → Stack S → Stack (T ∷ S)
```

This defines the `Stack` type as an inductive family indexed by a list of types that define the types of elements in the stack. The first constructor, `ε`, is an empty stack, as the list of types contained in the stack is empty. The second constructor, `_▷_`, is an infix operator which pushes an element with type `T` on to the top of the stack, and extends the type of the stack with the type of this element. Note that the types `T` and `S` passed as arguments to the `_▷_` constructor are implicit as indicated by the curly braces, as they can be inferred from the latter arguments.

The following is an example of a stack with two elements, the first of which is a natural number and the second is a Boolean.

```
1 ▷ true ▷ ε : Stack (ℕ :: Bool :: [])
```

Part II

Methodology and applications

Chapter 5

Compiling a simple expression language

This chapter introduces the methodology for calculating a compiler from a language's specification, extended from Bahr and Hutton (2015)'s methodoly by moving to a dependently-typed setting. We begin by defining a simple expression-based source language and its semantics, together with a specification of what it means for a compiler to be correct. We then do an inductive calculation over this specification, to construct an abstract machine, and a compilation function from the source language to this abstract machine's code. As the compiler is derived from the source language's specification, we are also guaranteed that the compiler will be correct by construction.

The first calculation in this chapter produces machine instructions which are explicitly concatenated together using a separate constructor. We then simplify the calculation process in the second example, replacing the need for an explicit concatenation operator by introducing code continuations.

The calculations are performed in the setting of the dependently-typed language Agda (Norell, 2007), a brief overview of which was given in chapter 4.

All code examples are verified in Agda, the code for which is available in appendix A.

5.1 Introduction

The compiler calculation methodology from Bahr and Hutton (2015) is developed in the setting of Haskell (Marlow, 2010), a statically-typed, purely-functional programming language. For simplicity the source language for the compiler is assumed to be untyped. Specifically, the language is restricted to having a single base type, the integers. This simplification played a key role in the development of the approach, but places limitations on the range of source languages that can be considered, and means that the benefits of static typing are not available in the methodology.

Using an untyped source language also gives rise to partiality problems. For example, the semantics of the untyped lambda calculus is partial as lambda terms may be badly formed or fail to terminate. Even for a minimal source language comprising only integers and addition, in which all source terms are well-formed, the semantics for the target language becomes partial as a stack may underflow or a register may be empty. Having to deal with such partiality issues complicates the methodology, particularly if the calculations are to be mechanised in a proof assistant.

In this chapter, we show how Bahr and Hutton's approach to calculating compilers can naturally be adapted to well-typed source and target languages, by moving from the setting of Haskell to the dependently-typed language Agda (Norell, 2007). In particular, the additional precision that is afforded by using dependent types allows us to statically ensure that only valid source and target terms can be constructed. As we shall see, this not only guarantees that all the components are type-safe, but also makes the calculations easier to mechanically check.

5.2 Expression language

We introduce our approach using the simple expression language used by McKinna and Wright (2006) in their development of a type-correct, stack-safe, provably correct expression compiler. We show how our approach can be used to calculate their well-typed compiler, with the correctness proof and all of the required compilation machinery falling naturally out of the calculation process.

This expression language is formed out of natural numbers and Boolean values. It is not type-safe in a simply-typed setting, but can be made type-safe using inductive families in a dependently-typed setting, as shown by McKinna and Wright (2006).

We begin by defining the language and its semantics in Agda, then specify what it means for the compiler to be correct, and finally show how the compiler can be calculated.

5.2.1 Source language

Consider a language of simple arithmetic expressions built up from natural numbers using an addition operator. In Agda, the syntax of such a language can be captured by the following type declaration, where `ℕ` is the type of natural numbers:

```
data Exp : Set where
  val : ℕ → Exp
  add : Exp → Exp → Exp
```

This defines a new type, `Exp`, with two new constructors, `val` and `add`. Recall that the declaration `Exp : Set` is defining `Exp` as a new type. The two constructors allow us to construct an expression as a natural number value, or the addition of two expressions.

Every expression in the above language is well-formed, because there is only one type: the natural numbers. However, we may encounter problems if the language contains more than one type. For example, consider extending the language with Boolean values, and a conditional operator that chooses between two expressions based on a Boolean value supplied as its first argument:

```
data Exp : Set where
  val  : ℕ → Exp
  add  : Exp → Exp → Exp
  bool : Bool → Exp
  if   : Exp → Exp → Exp → Exp
```

While this language allows us to construct expressions that contain both numbers and Boolean values, it does not guarantee they are always valid. For example, `add (bool true) (val 1)` is ill-formed, because the first argument to `add` is a Boolean value, not a natural number.

Using the additional expressive power provided by a dependently-typed language such as Agda, these ill-formed expressions can be made unrepresentable. We achieve this by indexing `Exp` with a type `T`, such that `Exp T` is only inhabited by expressions of type `T`. For simplicity we initially allow `T` to be any `Set`, but this will be restricted to a smaller universe of types in later examples. To avoid having to repeatedly state the type of such parameters throughout the chapter, we use Agda's generalised variable feature to first declare that the variable `T` is always of type `Set`:

```
variable
  T : Set


data Exp : Set → Set where
  val : T → Exp T
  add : Exp ℕ → Exp ℕ → Exp ℕ
  if  : Exp Bool → Exp T → Exp T → Exp T
```

In this manner, `Exp` is now an *inductive family* (Dybjer, 1994), indexed by

the type of the expression. The constructor `val` now lifts a value of type `T` into an expression of this type, which avoids the need for separate constructors for natural numbers and Boolean values. In turn, the constructor `add` takes two expressions of type `ℕ`, and returns another expression of this type. Finally, `if` takes a Boolean expression and two expressions of type `T`, and returns an expression of type `T`.

The new version of `Exp` guarantees that all expressions are well-formed by construction. For example, the invalid expression `add (val true) (val 1)` can no longer be constructed, because `val true : Exp Bool`, whereas `add` requires both its arguments to have type `Exp ℕ`. We note that `Exp` could also be defined using Haskell's generalised algebraic data types (GADTs), as these provide similar functionality to inductive families. However, Agda provides better facilities for working with such types, including dependent pattern matching and a development environment that allows for interactive development of programs based on their types.

We conclude this section by defining an evaluation semantics for expressions. Because `Exp` is indexed by the expression type `T`, the evaluation function can return a value of this type. This follows a similar approach to the well-typed interpreter of Augustsson and Carlsson (1999), ensuring that the function is type-safe and total, because there is never a need to check that the returned value has the expected type. For example, in the case for `add x y` in the `eval` function below, we know statically that `eval x : ℕ`, which allows this term to be used directly as an argument to the `+` function on natural numbers.

```
eval : Exp T → T
eval (val x)    = x
eval (add x y)  = eval x + eval y
eval (if b x y) = if eval b then eval x else eval y
```

In this manner, using dependent types not only makes the code type-safe, but also makes it more concise by eliminating the need to check for type

errors at run-time.

5.2.2 Compiler specification

We now seek to calculate a compiler for our well-typed expression language, using the approach developed by Bahr and Hutton (2015). In that setting, the target language for the compiler is a stack-based virtual machine in which a stack is represented as a list of numbers. In our setting, we can take advantage of the power of dependent types and use a more refined stack type that is indexed by the types of the elements it contains (Poulsen et al., 2018):

```
variable
  S S' S'' : List Set


data Stack : List Set → Set where
  ε : Stack []
  _▷_ : T → Stack S → Stack (T :: S)
```

A stack is either empty, written as `ε`, or is formed by pushing a value `a` of type `T` onto an existing stack `s` of type `S` to give a new stack of type `T :: S`, written as `a ▷ s`. For example, the stack `5 ▷ True ▷ ε` has type `Stack (ℕ :: Bool :: [])`, which makes precise the fact that it contains a natural number and a Boolean value.

Our aim now is to define three further components: a code type that represents the syntax of the target language, a compilation function that translates an expression into code, and an execution function that runs code. In the setting of Bahr and Hutton (2015), code is an algebraic data type (rather than an inductive family), the compiler is a function from expressions to code, and the execution function runs code using an initial stack of numbers to give a final stack of numbers. In our dependently-typed setting, we can be more precise about all of these types.

First of all, as in McKinna and Wright (2006), we can index code by the types of the input and output stacks it operates on, by means of the following declaration for the `Code` type:

```
data Code : List Set → List Set → Set₁
```

The idea is that `Code S S'` represents code that takes an input stack of type `S` and produces an output stack of type `S'`. The result type is `Set₁` (where `Set : Set₁`) to prevent inconsistencies from arising, but this does not affect the rest of the calculation. We don't yet define any constructors for this type, as these will be derived during the calculation process.

Using the code type, we can now specify the type of the execution function in a manner that makes the types of the input and output stacks precise:

```
exec : Code S S' → Stack S → Stack S'
```

Again, we don't yet define this function, as the definition will also be calculated.

The type of the compilation function can now make explicit that the compiled code for an expression of type `T` pushes a value of type `T` onto the top of the input stack:

```
compile : Exp T → Code S (T :: S)
```

Finally, the desired relationship between the source semantics (`eval`), the compilation function (`compile`), and the target semantics (`exec`) is captured by the following equation:

```
exec (compile e) s = eval e ▷ s
```
(5.1)

That is, compiling an expression and then executing the resulting code on a given stack gives the same result as pushing the value of the expression onto the stack. This is the same correctness condition used by Bahr and Hutton (2015), except that by using Agda rather than Haskell we are able to be more precise about the underlying types of the components. As we shall see, this additional precision also brings a number of benefits during

the calculation process.

5.2.3 Compiler calculation

To calculate the compiler, we proceed from Equation (5.1) by structural induction on the the form of the source expression `e`. In each case, we start with the right-hand side `eval e ▷ s` of the equation and seek to transform it by equational reasoning into the form `exec c s` for some code `c`, such that we can take `compile e = c` as a defining equation for the `compile` function. In order to do this we will find that we need to define constructors for the `Code` type, along with their interpretation by the execution function `exec`. It is important to point out here that this is how we will find the definitions for the the `Code` type and the `exec` function—through calculating each case as the need arises.

Case: **`e = val x`**

We begin with the right-hand side of Equation (5.1), and apply the definition of `eval`:

```
    eval (val x) ▷ s
=   { definition of eval }
    x ▷ s
```

To reach the required form, we must now rewrite the resulting term `x ▷ s` into the form `exec c s` for some code `c`. That is, we need to solve the following equation:

```
exec c s = x ▷ s
```

Note that we can't use this equation as a definition for the function `exec`, because the value `x` would be unbound in the body of the definition as

it does not appear on the left-hand side. The solution is to package this variable up in the code argument `c` — which can freely be instantiated as it is existentially quantified — by introducing a new code constructor `PUSH` that takes this variable as an argument, and defining a new equation for `exec` as follows:

```
exec (PUSH x) s = x ▷ s
```

That is, executing code of the form `PUSH x` pushes the value `x` onto the stack, hence the choice of the name for the new code constructor. Using the above ideas, it is now straightforward to complete the calculation using the new definition for `exec`:

```
    x ▷ s
=   { define: exec (PUSH x) s = x ▷ s }
    exec (PUSH x) s
```

The final term now has the form `exec c s`, where `c = PUSH x`. For this to fit the form of our compiler specification, we must define the following case for compile:

```
compile (val x) = PUSH x
```

That is, compiling a value `x` results in code that pushes this value onto the stack. From this definition and the type `Exp T → Code S (T :: S)` for the function `compile`, we can infer that:

```
PUSH : T → Code S (T :: S)
```

That is, the constructor `PUSH` transforms a value of type `T` into code that returns a value of this type on top of the stack, which is the expected behaviour.

Case: `e = if b x y`

We begin again by applying the evaluation function:

```
    eval (if b x y) ▷ s
=   { definition of eval }
    (if eval b then eval x else eval y) ▷ s
```

No further definitions can be applied at this point. However, as we are performing an inductive calculation, we can make use of the induction hypotheses for the three argument expressions `b`, `x` and `y`, which have the following form for any stack `s'`:

```
exec (compile b) s' = eval b ▷ s'

exec (compile x) s' = eval x ▷ s'

exec (compile y) s' = eval y ▷ s'
```

In order to use these hypotheses, we first exploit the fact that function application distributes over conditional expressions, which is captured by the following equation:

```
f (if b then x else y) = if b then f x else f y
```

This property is always valid in a total, terminating language such as Agda. In our case, it allows us to promote the function `(▷ s)` into both branches of the conditional expression, and hence to apply the induction hypotheses for the arguments `x` and `y`:

```
    (if eval b then eval x else eval y) ▷ s
=   { distributivity }
    if eval b then eval x ▷ s else eval y ▷ s
=   { induction hypotheses for x and y }
    if eval b then exec (compile x) s else exec (compile y) s
```

Now we are stuck again. In order to reach the required form, we must transform the term we are manipulating into the form `exec c s` for some code `c`. That is, we need to solve the equation:

```
exec c s = if eval b then exec (compile x) s
                     else exec (compile y) s
```

First of all, we generalise this equation from the specific Boolean value `eval b`, code `compile x` and code `compile y`, to give the following simpler equation:

```
exec c s = if b then exec c1 s else exec c2 s
```

Once again, we can't use this equation as a definition for `exec`, this time because `b`, `c1` and `c2` would be unbound. We must bind them on the left-hand side of the equation, but we have a choice of whether to store them in the stack argument `s` or the code argument `c`. All values on the stack will be evaluated at runtime, but code may be conditionally evaluated depending on the rest of the program. In this case, we package `c1` and `c2` up in the code argument `c`, as we want to choose which one to evaluate, and we store `b` on the stack because it should always be evaluated. This is achieved by introducing a new code constructor `IF`, such that:

```
exec (IF c1 c2) (b ▷ s) = if b then exec c1 s else exec c2 s
```

That is, executing code of the form `IF c1 c2` given a stack with a Boolean value `b` on top proceeds by executing one of the two pieces of code in the remaining stack, depending on the value of `b`. We can now continue the calculation and apply the induction hypothesis for the argument `b`:

```
    if eval b then exec (compile x) s else exec (compile y) s
=   { define: exec (IF c1 c2) (b ▷ s) =
                  if b then exec c1 s else exec c2 s }
    exec (IF (compile x) (compile y)) (eval b ▷ s)
=   { induction hypothesis for b }
    exec (IF (compile x) (compile y)) (exec (compile b) s)
```

We are now stuck, but are guided again by our aim to obtain a term of the

form `exec c s` for some code `c`. That is, we now need to solve the equation:

```
exec c s = exec (IF (compile x) (compile y))
                (exec (compile b) s)
```

In a similar manner to previously, we first generalise over the specific code arguments that are supplied to the function `exec` on the right-hand side, to give:

```
exec c s = exec c2 (exec c1 s)
```

This equation can then be solved by packaging `c1` and `c2` up in the code argument `c` using a new code constructor `+++` that takes these pieces of code as arguments. This represents concatenation of two pieces of code to be executed one after another:

```
exec (c1 +++ c2) s = exec c2 (exec c1 s)
```

It is now straightforward to complete the calculation:

```
  exec (IF (compile x) (compile y)) (exec (compile b) s)
=   { define: exec (c1 +++ c2) s = exec c2 (exec c1 s) }
  exec (compile b +++ IF (compile x) (compile y)) s
```

The final term now has the required form `exec c s`, from which we conclude that Equation (5.1) is satisfied in the case of conditional expressions by defining:

```
compile (if b x y) = compile b +++ IF (compile x) (compile y)
```

That is, a conditional expression is compiled by compiling the condition and using the resulting value to make a choice between the compiled code for the two branches. The two new code constructors used in this definition have the following types:

```
_+++_ : Code S S' → Code S' S'' → Code S S''
```

```
IF : Code S S' → Code S S' → Code (Bool :: S) S'
```

We can see from the type of the `+++` constructor that it allows two pieces of code to be concatenated, as long as the output stack of the first piece of code matches the input stack of the second. In turn, the `IF` constructor

transforms two pieces of code of the same type into a single piece of code that uses a Boolean value on the stack to choose between them.

Case: `e = add x y`

We begin in the same way as the previous two cases, by starting with the right-hand side of the specification, and applying the evaluation function:

```
    eval (add x y) ▷ s
=   { definition of eval }
    (eval x + eval y) ▷ s
```

In order to rewrite the resulting term above into the form `exec c s` for some code `c`, we need to solve the equation `exec c s = (eval x + eval y) ▷ s`. First of all, we generalise over the specific natural numbers `eval x` and `eval y` to give the equation `exec c s = (n + m) ▷ s`. The numbers `m` and `n` in this new equation are unbound, but can be packaged up in the stack argument `s` by means of a new code constructor `ADD` and the definition:

```
exec ADD (m ▷ n ▷ s) = (n + m) ▷ s
```

That is, executing an `ADD` proceeds by adding the two natural numbers on the top of the stack. It is now straightforward to complete the calculation, during which we use `+++` from the previous case to bring the term being manipulated into the required form:

```
    (eval x + eval y) ▷ s
=   { define: exec ADD (m ▷ n ▷ s) = (n + m) ▷ s }
    exec ADD (eval y ▷ eval x ▷ s)
=   { induction hypothesis for x }
    exec ADD (eval y ▷ exec (compile x) s)
=   { induction hypothesis for y }
    exec ADD (exec (compile y) (exec (compile x) s))
=   { definition of +++ }
```

```
    exec ADD (exec (compile x +++ compile y) s)
=   { definition of +++ }
    exec ((compile x +++ compile y) +++ ADD) s
```

The final term now has the form `exec c s`, from which we conclude that the following definition satisfies Equation (5.1) in the case of adding two numeric expressions:

```
compile (add x y) = (compile x +++ compile y) +++ ADD
```

That is, an addition is compiled by compiling the two argument expressions and adding the resulting two numbers on the stack. The type of the `ADD` constructor makes explicit that it transforms a stack with two numbers on top into a stack with a single number on top, as expected:

```
ADD : Code (ℕ :: ℕ :: S) (ℕ :: S)
```

In summary, we have calculated the following definitions:

```
data Code : List Set → List Set → Set₁ where
  PUSH  : T → Code S (T :: S)
  _+++_ : Code S S' → Code S' S'' → Code S S''
  IF    : Code S S' → Code S S' → Code (Bool :: S) S'
  ADD   : Code (ℕ :: ℕ :: S) (ℕ :: S)

compile : Exp T → Code S (T :: S)
compile (val x)    = PUSH x
compile (if b x y) = compile b +++ IF (compile x) (compile y)
compile (add x y)  = (compile x +++ compile y) +++ ADD

exec : Code S S' → Stack S → Stack S'
exec (PUSH x) s         = x ▷ s
exec (IF c1 c2) (b ▷ s) = if b then exec c1 s else exec c2 s
exec (c1 +++ c2) s      = exec c2 (exec c1 s)
exec ADD (m ▷ n ▷ s)    = (n + m) ▷ s
```

5.2.4 Reflection

The above definitions are essentially the same as those developed by McKinna and Wright (2006), except that we calculated the definitions in a manner that ensures they are *correct by construction*. In contrast, McKinna and Wright gave the definitions and then separately proved correctness. As we have seen, using our methodology each of the definitions arose in a systematic way, with all of the required compilation machinery falling naturally out of the calculation.

The use of dependent types does not complicate the calculation, but ensures that the stack and code types are completely type-safe. As a result, the `exec` function is total, because the types guarantee that the stack can never underflow, and that stack elements are always of the expected types. In contrast, in the original setting of Bahr and Hutton (2015), the `exec` function is partial, which causes difficulties when mechanising the calculations, because proof assistants often require all functions to be total. For example, the partial `exec` function had to be reformulated as a small-step execution relation prior to mechanisation in Coq, to avoid such totality issues.

In our total setting, however, the mechanisation in Agda proceeds in essentially the same manner as the pen-and-paper version. For example, the case for addition is illustrated below (with other cases omitted for brevity):

```
correct : (e : Exp T) → (s : Stack S) →
    eval e ▷ s ≡ exec (compile e) s
correct (val x)    s = ...
correct (if b x y) s = ...
correct (add x y)  s = begin
    eval (add x y) ▷ s
  -- definition of eval
  ≡⟨ refl ⟩
```

```
    (eval x + eval y) ▷ s
  -- define: exec ADD (m ▷ n ▷ s) = (n + m) ▷ s
  ≡⟨ refl ⟩
    exec ADD (eval y ▷ eval x ▷ s)
  -- induction hypothesis for x
  ≡⟨ cong (λz → exec ADD (eval y ▷ z)) (correct x s) ⟩
    exec ADD (eval y ▷ exec (compile x) s)
  -- induction hypothesis for y
  ≡⟨ cong (exec ADD) (correct y (exec (compile x) s)) ⟩
    exec ADD (exec (compile y) (exec (compile x) s))
  -- definition of +++
  ≡⟨ refl ⟩
    exec ADD (exec (compile x +++ compile y) s)
  -- definition of +++
  ≡⟨ refl ⟩
    exec ((compile x +++ compile y) +++ ADD) s
  ∎
```

The above calculation is written using Agda's equational reasoning facility, which is provided in the standard library. Note that some steps are reduced to reflexivity because Agda automatically applies definitions, and we sometimes need to make explicit where transformations are applied, using properties such as congruence, written as `cong` in Agda, to apply an equality in a particular context.

Finally, we remark that the compiler we have calculated produces tree-structured rather than linear code, due to the use of the `+++` constructor for combining pieces of code. We could add a post-processing phase to rearrange the code from a tree into a list. However, in the next section we will see how the use of code continuations achieves the same result in a more direct manner.

5.3 Code continuations

One of the key ideas in Bahr and Hutton's methodology (2015; 2020) is the use of *code continuations* to streamline the resulting compiler calculations. In our well-typed setting, this can be achieved by generalising the function `compile : Exp T → Code S (T ∷ S)` to a function `comp` that takes code to be executed after the compiled code as an additional argument:

```
comp : Exp T → Code (T ∷ S) S' → Code S S'
```

The additional argument of type `Code (T :: S) S'` can be viewed as a code continuation that will be run after the argument expression is run. The continuation takes as its input a stack with the evaluated argument expression on top, and returns the final stack to be output. Later we will calculate a function `compile` that does not require a code continuation, in terms of `comp`. The correctness equation `exec (compile e) s = eval e ▷ s` for the compiler must now be modified to use the `comp` function with the code continuation as an extra argument:

```
exec (comp e c) s = exec c (eval e ▷ s)
```
(5.2)

That is, compiling an expression and executing the resulting code followed by some code `c`, gives the same result as executing `c` with the value of the expression on the top of the stack. This is the same generalised correctness condition used by Bahr and Hutton (2015), except that once again the use of Agda allows us to be more precise about the types of the component functions.

To calculate the compiler we proceed from Equation (5.2) by structural induction on the expression `e`. In each case, we aim to transform the right-hand side `exec c (eval e ▷ s)` into the form `exec c' s` for some code `c'`, such that we can then take `comp e c = c'` as a defining equation for `comp` in this case. The calculation itself proceeds in a similar manner to the previous section, with new `Code` constructors and their interpretation by `exec` being

introduced along the way. As before, the driving force for introducing these new components is the desire to rewrite the term being manipulated into the required form, or to facilitate applying an induction hypothesis.

Case: **`e = val x`**

```
    exec c (eval (val x) ▷ s)
=   { definition of eval }
    exec c (x ▷ s)
=   { define: exec (PUSH x c) s = exec c (x ▷ s) }
    exec (PUSH x c) s
```

Case: **`e = if b x y`**

```
    exec c (eval (if b x y)) ▷ s
=   { definition of eval }
    exec c ((if eval b then eval x else eval y) ▷ s)
=   { distributivity }
    exec c (if eval b then eval x ▷ s else eval y ▷ s)
=   { distributivity }
    if eval b then exec c (eval x ▷ s) else exec c (eval y ▷ s)
=   { induction hypotheses for x and y }
    if eval b then exec (comp x c) s else exec (comp y c) s
```

```
  =  { define: exec (IF c1 c2) (b ▷ s) =
                  if b then exec c1 s else exec c2 s }
     exec (IF (comp x c) (comp y c)) (eval b ▷ s)
  =  { induction hypothesis for b }
     exec (comp b (IF (comp x c) (comp y c))) s
```

Case: **`e = add x y`**

```
     exec c (eval (add x y) ▷ s)
  =  { definition of eval }
     exec c ((eval x + eval y) ▷ s)
  =  { define: exec (ADD c) (m ▷ n ▷ s) = exec c ((n + m) ▷ s) }
     exec (ADD c) (eval y ▷ eval x ▷ s)
  =  { induction hypothesis for y }
     exec (comp y (ADD c)) (eval x ▷ s)
  =  { induction hypothesis for x }
     exec (comp x (comp y (ADD c))) s
```

The base case calculation above is similar to the previous version, but the two inductive cases are simpler because the use of code continuations means that the `+++` constructor for composing two pieces of code is no longer required. This is an example of the well-known idea of 'making append vanish' (Wadler, 1989) by rewriting a recursive definition using an accumulating parameter, here taking the form of a code continuation that is applied within a dependently-type setting.

We conclude the calculation by returning to the top-level compilation function `compile`, whose correctness was captured by the following equation.

```
exec (compile e) s = eval e ▷ s
```

As previously, we aim to rewrite the right-hand side of this equation into the form `exec c s` for some code `c`, such that we can then define `compile e = c`. In this case there is no need to use induction as simple calculation suffices,

during which we introduce a new code constructor `HALT` to transform the term being manipulated into a form that allows Equation (5.2) to be applied:

```
    eval e ▷ s
=   { define: exec HALT s = s }
    exec HALT (eval e ▷ s)
=   { apply Equation (5.2) }
    exec (comp e HALT) s
```

In summary, we have calculated the following definitions:

```
data Code : List Set → List Set → Set₁ where
  PUSH : T → Code (T ∷ S) S' → Code S S'
  ADD  : Code (ℕ ∷ S) S' → Code (ℕ ∷ ℕ ∷ S) S'
  IF   : Code S S' → Code S S' → Code (Bool ∷ S) S'
  HALT : Code S S

comp : Exp T → Code (T ∷ S) S' → Code S S'
comp (val x)    c = PUSH x c
comp (if b x y) c = comp b (IF (comp x c) (comp y c))
comp (add x y)  c = comp x (comp y (ADD c))

compile : Exp T → Code S (T ∷ S)
compile e = comp e HALT

exec : Code S S' → Stack S → Stack S'
exec (PUSH x c) s          = exec c (x ▷ s)
exec (IF c1 c2) (b ▷ s)    = if b then exec c1 s else exec c2 s
exec (ADD c) (m ▷ n ▷ s)   = exec c ((n + m) ▷ s)
exec HALT s                = s
```

5.3.1 Reflection

Once again the above definitions do not require separate proofs of correctness, as they are correct by construction. As well as eliminating the need for the `+++` constructor, the use of code continuations streamlines the resulting calculation by reducing the number of steps that are required. Moreover, the use of dependent types to specify the stack behaviour of code statically guarantees that the code continuation supplied to the compiler will be able to operate on its input stack without underflowing or assuming the wrong type for an element. This additional level of safety ensures that the execution function is total, independent of the code that is supplied to it or the compiler, which is not the case in the original, untyped methodology (Bahr and Hutton, 2015).

Note that the code produced by the above compiler is still not fully linear, because the `IF` constructor takes two pieces of code as arguments. This branching structure corresponds to the underlying branching in control flow in the semantics of conditional expressions, but duplicates the continuations following the branch. If desired, however, we can transform the compiler to eliminate this duplication by modifying `IF` to take a code pointer as its first argument rather than code itself (Rouvoet et al., 2021; Bahr and Hutton, 2024).

Chapter 6

Adding exceptions

We now consider a source language that provides support for throwing and catching exceptions. The addition of such features makes compilation more challenging, as they disrupt the normal flow of control, and it may not be known at compile-time where control will resume after an exception is thrown. In our setting, however, we can use the additional power provided by dependent types to ensure that source expressions and target code are always well-formed. For example, we can ensure that evaluation of an expression never results in an uncaught exception, and that execution of an exception handler always returns the stack to the expected form.

In the past it proved difficult even to write a dependently-typed compiler for exceptions, so it is pleasing that our new methodology will allow us to calculate such a compiler in a systematic manner.

6.1 Source language

We start with a simple source language of addition and numeric values as in the previous chapter, to which we add `throw` and `catch` constructors to allow for exception handling. Informally, `throw` throws an exception, which

can be caught by `catch`. In the expression `catch x h`, the first expression is evaluated. If no exception is thrown, the result of the overall expression is the value of `x`. If an exception is thrown, the exception handler `h` is evaluated and that is the value of the overall expression. Note that the handler may throw an exception too, which would need to be caught by an outer `catch`.

As in the exception language considered by Bahr and Hutton (2015), we use only trivial exceptions with no data attached for simplicity of calculation. However, our expression type is indexed with a Boolean value that indicates whether the given expression can throw an exception or not:

```
variable
  a b : Bool


data Exp : Bool → Set where
  val   : ℕ → Exp false
  add   : Exp a → Exp b → Exp (a ∨ b)
  throw : Exp true
  catch : Exp a → Exp b → Exp (a ∧ b)
```

The `Bool` indices in the `Exp` declaration express that a numeric value cannot throw an exception whereas `throw` can, while an addition throws an exception if either argument does, and a catch throws an exception if both arguments do (Spivey, 1990). In this manner, `Exp false` is the type of well-formed expressions that never result in an uncaught exception, and `Exp true` is the type of exceptions that are permitted (but not required) to throw an exception. Within such expressions, exceptions may be freely thrown as long as they are always caught. For example, the expression `catch throw (val 1)` has type `Exp false`, as the exception is caught by the handler.

To simplify the presentation of this example, an expression that does not result in an uncaught exception returns a natural number. This assumption

allows us to focus on the essential aspects of compiling exceptions in a dependently-typed setting.

6.2 Semantics

We define two evaluation semantics for expressions: `eval?` for the case when the result may be an uncaught exception, and `eval` for the case when evaluation is guaranteed to succeed. For the former, we utilise the built-in type `Maybe a`, with values of the form `just x` representing successful evaluation, and `nothing` representing failure due to an uncaught exception:

```
eval? : Exp b → Maybe ℕ
eval? (val n)     = just n
eval? (add x y)   = case eval? x of
                      just n  → case eval? y of
                                  just m  → just (n + m)
                                  nothing → nothing
                      nothing → nothing
eval? throw       = nothing
eval? (catch x h) = case eval? x of
                      just n  → just n
                      nothing → eval? h
```

Note that Agda does not support `case` expressions as in Haskell, but we use them here to simplify the presentation — the actual code uses the standard 'fold' operator for the `Maybe` type. The above definition states that numeric values evaluate to themselves, addition evaluates both of its argument expressions and only succeeds if both results are successful, `throw` will always throw an exception, and `catch` returns the result of evaluating its first argument expression if it is successful, and otherwise handles the resulting exception by evaluating its second argument.

Whereas `eval?` can take any expression as its argument, `eval` only takes

expressions that cannot result in an uncaught exception, witnessed by the proof `b ≡ false`. Because evaluation cannot fail in this case, the return type of `eval` is simply `ℕ` rather than `Maybe ℕ`:

```
eval : b ≡ false → Exp b → ℕ
eval p (val n)                    = n
eval p (add {false} {false} x y)  = eval refl x + eval refl y
eval p (catch {false} {a} x h)    = eval refl x
eval p (catch {true} {false} x h) = case eval? x of
                                      just n  → n
                                      nothing → eval refl h
```

There are a few further points to note about this definition. First of all, for technical reasons we cannot simply write `Exp false → ℕ` as the type for the `eval` function. Rather, we need to provide an additional argument `p : b ≡ false` as a proof that the index is `false`, which in recursive calls is given simply by a proof of reflexivity, `refl : false ≡ false`.

Secondly, the patterns in curly brackets, such as `{false}`, match implicit arguments; in this case, the `Bool` indices for the argument expressions. For example, in the `add` case both indices must be `false`, because if an overall addition expression cannot throw an exception, neither can either of its arguments. The implicit argument patterns occur in the same order as the expression arguments, so the first implicit pattern corresponds to the first expression argument, and likewise for the second.

Note that the `eval` function does not need to check for the possibility of failure in recursive calls to `eval`. Instead, the proof arguments passed in statically ensure that the recursive calls cannot result in uncaught exceptions. Similarly, no cases are required for `throw` or for addition where sub-expressions can throw an exception, as the type checker can statically verify that these cases are not possible. In this way, the use of dependent types reduces the need for redundant failure checking, and makes it clear where failure does need to be considered.

Finally, `eval` is defined in terms of `eval?` because in the `catch` case the expression `x` may still throw an exception, which is then caught by the handler. If `eval` was used here, type checking would fail as no proof can be provided that `x` cannot throw an exception.

6.3 Compiler specification

We now seek to calculate a compiler for our well-typed language with exceptions, which produces code for a stack-based virtual machine. In our previous examples, we utilised a stack type that was indexed by the types of the elements that it contains, namely `Stack : List Set → Set`. For this compiler, however, we use an alternative representation for the element type in the form of a custom *universe* of types, which we denote by `Ty`:

```
Stack : List Ty → Set
```

The reason for this change is that during the calculation the idea of putting code onto the stack will arise, which causes problems with universe levels and positivity checking if `Set` is the index type. Using a custom type universe avoids this problem. We could also start with the original `Stack` type and observe during the calculation that we need to change the definition because of typing issues. Indeed, this is precisely what happened when we attempted this calculation for the first time.

We begin with a universe `Ty` with a single constructor `nat` that represents the natural numbers, whose meaning is given by a function `El`—the standard name for this function in type theory, abbreviating 'element'—that gives the interpretation for our universe in terms of Agda's `Set`:

```
data Ty : Set where
  nat : Ty


El : Ty → Set
El nat = ℕ
```

During the calculation we will extend `Ty` with a new constructor, together with its interpretation by `El`. Using the above ideas, our original stack type can now be refined as follows:

```
variable
  T : Ty
  S S' S'' : List Ty


data Stack : List Ty → Set where
  ε : Stack []
  _▷_ : El T → Stack S → Stack (T :: S)
```

This is the same as the `Stack` type from Chapter 5, with `Ty` used instead of `Set`, and `El` used for the interpretation of types in Agda. For example, the stack `1 ▷ 2 ▷ ε` has type `Stack (nat :: nat :: [])`. In turn, the types for `Code` and the `exec` function are the same as previously, except that in the former case we now use `List Ty` rather than `List Set`, and `Code` is now in `Set` rather than `Set₁` because of our use of a custom universe:

```
data Code : List Ty → List Ty → Set


exec : Code S S' → Stack S → Stack S'
```

However, we will calculate a new set of operations for the virtual machine for our language with exceptions, so we don't yet define any constructors for the `Code` type.

Just as we defined two evaluation semantics for expressions, `eval?` and `eval`, so we seek to define two compilation functions: `comp?` for the case when the expression may result in an uncaught exception, and `comp` for the case when success is guaranteed. We begin with the latter case, for which the type `comp : Exp T → Code (T :: S) S' → Code S S'` from Section 5.3 is adapted to our language with exceptions in the following straightforward manner:

```
comp : b ≡ false → Exp b → Code (nat :: S) S' → Code S S'
```

That is, the compiler now requires a proof that the expression argument

cannot throw an exception, and the code continuation takes the resulting natural number on top of the stack. In turn, the correctness equation `exec (comp e c) s = exec c (eval e ▷ s)` for the compiler `comp` just requires passing a proof `p : b ≡ false`, that the expression cannot throw an exception, to the `comp` and `eval` functions:

```
exec (comp p e c) s = exec c (eval p e ▷ s)        (6.1)
```

We will return to the type and correctness equation for `comp?` later on.

6.4 Compiler calculation

As previously, we calculate the compiler from Equation (6.1) by induction on the expression `e`. In each case, we aim to transform the right-hand-side of the equation, `exec c (eval p e ▷ s)`, into the form `exec c' s` for some code `c'`. We can then take `comp p e c = c'` as a defining equation for `comp`.

Case: **`e = throw`**

The use of dependent types in our source language allows us to eliminate this case, as `throw` has type `Exp true`, but from the type of the compilation function `comp` we have a proof that the index of expression argument is `false`, and hence this case cannot arise.

Case: **`e = val n`**

The calculation for values proceeds in the same way as previously,

```
    exec c (eval p (val n) ▷ s)
=    { definition of eval }
    exec c (n ▷ s)
```

```
  =   { define: exec (PUSH n c) s = exec c (n ▷ s) }
    exec (PUSH n c) s
```

giving us the definition of the first case for the compiler:

```
comp p (val n) c = PUSH n c
```

Case: **`e = add x y`**

As well as taking expression arguments `x` and `y`, addition takes implicit arguments specifying if these arguments can throw an exception or not. Because of the proof passed to the `comp` function, we know that `add x y` cannot throw an exception, so neither of the argument expressions can. This allows us to pattern match on the implicit arguments in the pattern `add {false} {false} x y`, and safely eliminate the impossible cases where `x` or `y` throw an exception:

```
    exec c (eval p (add {false} {false} x y) ▷ s)
  =   { definition of eval }
    exec c (eval refl x + eval refl y ▷ s)
```

As in previous sections, we can introduce a new code constructor `ADD` to add two natural numbers on the top of the stack, which allows us to apply the induction hypotheses,

```
    exec c (eval refl x + eval refl y ▷ s)
  =   { define: exec (ADD c) (m ▷ n ▷ s) = exec c ((n + m) ▷ s) }
    exec (ADD c) (eval refl y ▷ eval refl x ▷ s)
  =   { induction hypothesis for y  }
    exec (comp refl y (ADD c)) (eval refl x ▷ s)
  =   { induction hypothesis for x  }
    exec (comp refl x (comp refl y (ADD c))) s
```

giving us the second case for the compiler:

```
comp p (add {false} {false} x y) c
  = comp refl x (comp refl y (ADD c))
```

Case: **`e = catch x h`**

Because of the proof passed to `comp`, `catch x h` cannot throw an exception, and hence there are only two cases to consider. The first is when `x` cannot throw an exception, regardless of whether `h` can, as the handler will never be called. The second is when `x` can throw an exception, but `h` cannot, as `h` must handle a possible exception and be guaranteed not to throw another. As with the case for `add`, we can pattern match on the implicit arguments to `catch` to distinguish these cases. In the first case, where the expression `x` cannot throw an exception, the handler is ignored,

```
    exec c (eval p (catch {false} {_} x h) ▷ s)
=   { definition of eval }
    exec c (eval refl x ▷ s)
=   { induction hypothesis for x }
    exec (comp refl x c) s
```

giving us another case for the compiler:

```
comp p (catch {false} {_} x h) c = comp refl x c
```

In the second case, where `x` may throw an exception, we are left with a `case` expression to determine whether `x` successfully returned a value or threw an exception:

```
    exec c (eval p (catch {true} {false} x h) ▷ s)
=   { definition of eval }
    exec c ((case eval? x of
      just n  → n
      nothing → eval refl h) ▷ s)
```

At this point, we seem to be stuck. In particular, the resulting term refers to `eval?`, the evaluation function that may throw an exception, but Equation (6.1) for `comp` does not provide us with any means of manipulating this. The solution is to consider the correctness equation for `comp?`, the compilation function for expressions that may throw an exception.

We do not yet know the type of `comp?`, because we have not yet calculated how exceptions will be treated by our compilers, but we can formulate a specification for its behaviour. In the case that `x` does not throw an exception, the specification should be equivalent to Equation (6.1). When `x` does throw an exception, it not clear how `comp?` should behave. Following the lead of Bahr and Hutton (2015), we make this explicit by introducing a new, but as yet undefined, function `fail` to handle this case, which takes the stack `s` as its argument. In summary, we now have the following *partial specification* for the function `comp?` in terms of an as yet undefined function `fail`:

```
exec (comp? e c) s = case eval? e of
    just n → exec c (n ▷ s)
    nothing → fail s
```
(6.2)

To continue the calculation, we must transform the term that we are manipulating into a form that matches Equation (6.2). Our term already performs case analysis on `eval? x`, so the natural strategy is to attempt to transform it to match the right-hand side of the equation. First of all, we can use the fact that function application distributes over `case` to push the `exec` function into each branch, which then allows the induction hypothesis for `h` to be applied:

```
    exec c ((case eval? x of
      just n  → n
      nothing → eval refl h) ▷ s)
=   { distributivity }
```

```
  case eval? x of
    just n  → exec c (n ▷ s)
    nothing → exec c (eval refl h ▷ s)
=  { induction hypothesis for h }
  case eval? x of
    just n  → exec c (n ▷ s)
    nothing → exec (comp refl h c) s
```

Now we are stuck again. However, in order to match Equation (6.2), the `nothing` branch must have the form `fail s'` for some stack `s'`. That is, we need to solve the equation:

```
fail s' = exec (comp refl h c) s
```

First of all, we generalise from `comp refl h c` to give the following simpler equation:

```
fail s' = exec h' s
```

However, we can't use this as a defining equation for `fail`, because `h'` and `s` would be unbound in the body of the definition. The solution is to package these two variables up in the stack argument `s'` by means of the following defining equation for `fail`:

```
fail (h' ▷ s) = exec h' s
```

That is, if `fail` is invoked with handler code on top of the stack, this code is executed using the remaining stack. Using this idea, we can now rewrite the `nothing` branch:

```
  case eval? x of
    just n  → exec c (n ▷ s)
    nothing → exec (comp refl h c) s
=  { define: fail (h' ▷ s) = exec h' s }
  case eval? x of
    just n  → exec c (n ▷ s)
    nothing → fail (comp refl h c ▷ s)
```

The resulting term is close to matching the right-hand side of Equation (6.2), except that the stacks in the two branches don't match up. In particular, the stack under the numeric result in the success branch, `s`, should match the stack in the failure branch, `comp refl h c ▷ s`. To achieve this match, we introduce a new code constructor, `UNMARK`, to modify the stack in the success branch to remove the unused handler from the stack, thereby allowing Equation (6.2) to be applied:

```
    case eval? x of
      just n  → exec c (n ▷ s)
      nothing → fail (comp refl h c ▷ s)
=   { define: exec (UNMARK c) (x ▷ h ▷ s) = exec c (x ▷ s) }
    case eval? x of
      just n  → exec (UNMARK c) (n ▷ comp refl h c ▷ s)
      nothing → fail (comp refl h c ▷ s)
=   { Equation (6.2) for x }
    exec (comp? x (UNMARK c)) (comp refl h c ▷ s)
```

The final step is to bring the equation into the form `exec c' s`, by modifying the stack. This can be done by introducing another code constructor, `MARK`, that pushes the compiled handler code onto the stack, similarly to the `PUSH` constructor for numeric values,

```
    exec (comp? x (UNMARK c)) (comp refl h c ▷ s)
=   { define: exec (MARK h c) s = exec c (h ▷ s) }
    exec (MARK (comp refl h c) (comp? x (UNMARK c))) s
```

resulting in our final defining equation for `comp`:

```
comp p (catch {true} {false} x h)
  = MARK (comp refl h c) (comp? x (UNMARK c))
```

6.5 Compiler specification II

We now turn our attention to the second compilation function, `comp?`, for expressions that may result in an uncaught exception. We begin by recalling the partial specification (6.2) for its behaviour that we developed during the calculation for `comp` in the previous section:

```
exec (comp? e c) s = case eval? e of
   just n  → exec c (n ▷ s)
   nothing → fail s
```

Now that we have calculated that exceptions will be compiled by putting handler code onto the stack, we can infer that `comp?` will have a type of the following form,

```
comp? : Exp b → Code (nat :: ??? :: S) S'
  → Code (??? :: S) S'
```

where `???` is a placeholder for the type of the handler code. The input stack type of the handler should be the same as the underlying stack type `S`, to ensure it can be used in this setting. Similarly, its output stack type should be `S'`, to ensure it results in the same form of stack as the case when an exception has not been thrown. To realise these ideas, we define a new constructor `han` in our universe `Ty` that represents handler code with given input and output stack types,

```
han : List Ty → List Ty → Ty
```

whose meaning is given by simply converting the constructor `han` into `Code`:

```
El (han S S') = Code S S'
```

Substituting in `han S S'` for `???`, the function `comp?` has the following type:

```
comp? : Exp b → Code (nat :: han S S' :: S) S'
  → Code (han S S' :: S) S'
```

That is, in order to compile an expression that may throw an exception, there must be a handler of the appropriate type on top of the stack. However, for the purposes of calculating the definition of `comp?`, requiring the

handler to be the top element turns out to be too restrictive. As we will see during the calculation, we need to generalise to allow any number of stack elements prior to the handler. This can be achieved by supplying `comp?` with an extra implicit argument, `S'' : List Ty`, which is appended to the stack using append operator for lists, denoted by `++`:

```
comp? : Exp b → Code (nat :: (S'' ++ han S S' :: S)) S'
  → Code (S'' ++ han S S' :: S) S'
```

6.6 Compiler calculation II

We proceed from Equation (6.2) by induction on the expression `e`.

Case: **`e = val n`**

The calculation for values is straightforward as it can never fail, and concludes by utilising the `PUSH` constructor that was introduced during the calculation for the `comp` function,

```
    case eval? (val n) of
      just n' → exec c (n' ▷ s)
      nothing → fail s
=   { definition of eval? }
    exec c (n ▷ s)
=   { definition of exec }
    exec (PUSH n c) s
```

which gives our first case for the definition of `comp?`:

```
comp? (val n) c = PUSH n c
```

Case: **`e = throw`**

This time around we need to consider the failing case, as the expression is permitted to throw an exception. We begin by applying the evaluation function, after which we introduce a new code constructor, `THROW`, to transform the term into the required form,

```
    case eval? throw of
      just n  → exec c (n ▷ s)
      nothing → fail s
=   { definition of eval? }
    fail s
=   { define: exec THROW s = fail s }
    exec THROW s
```

which gives our second case for the compiler:

```
comp? throw c = THROW
```

Note that this case discards the code continuation `c`, indicating that execution will continue elsewhere, in this case from the handler code that must already be present on the stack.

Case: **`e = add x y`**

Unlike the addition case for `comp`, now either argument expression could throw an exception. Nevertheless, we begin in the usual way by applying the evaluation function:

```
    case eval? (add x y) of
        just n  → exec c (n ▷ s)
        nothing → fail s
=   { definition of eval? }
    case eval? x of
```

```
        just n  → case eval? y of
            just m  → exec c ((n + m) ▷ s)
            nothing → fail s
        nothing → fail s
```

To proceed, we need to transform the nested `case` into the form of Equation (6.2). We transform the inner `just` branch by rewriting it using the `ADD` constructor introduced previously:

```
    case eval? x of
        just n  → case eval? y of
            just m  → exec c ((n + m) ▷ s)
            nothing → fail s
        nothing → fail s
=   { definition of exec }
    case eval? x of
        just n  → case eval? y of
            just m  → exec (ADD c) (m ▷ n ▷ s)
            nothing → fail s
        nothing → fail s
```

The inner `case` is nearly in the correct form, but the stack following `m` in the success case, `n ▷ s`, needs to match the stack in the failure case, `s`. That is, we need to solve the following equation:

```
fail (n ▷ s) = fail s
```

Although this equation appears to be in the correct form to be taken as a defining equation for `fail`, the left-hand side `fail (n ▷ s)` has the same form as that for our previous definition of `fail` for executing a handler, namely `fail (h' ▷ s)`. To allow these two definitions type-check and avoid overlapping patterns, we give `fail` the following type,

```
fail : Stack (S'' ++ han S S' :: S) → Stack S'
```

which states that it takes a stack containing a handler and any number

of elements prior to the handler, and returns the stack that is returned by the handler. This is the point in the calculation where we need the generalisation allowing multiple stack elements prior to the handler that was introduced at the end of the previous section. The two cases for `fail` can then be distinguished by pattern matching on the implicit argument `S''`, the list of types of elements prior to the handler, to determine whether the handler is on the top of the stack or not:

```
fail : Stack (S'' ++ han S S' :: S) → Stack S'
fail {[]}    (h' ▷ s) = exec h' s
fail {_ :: _} (n  ▷ s) = fail s
```

We can now apply this second definition of `fail` to continue the calculation,

```
    case eval? x of
        just n  → case eval? y of
            just m  → exec (ADD c) (m ▷ n ▷ s)
            nothing → fail s
        nothing → fail s
=   { define: fail {_ :: _} (n ▷ s) = fail s }
    case eval? x of
        just n  → case eval? y of
            just m  → exec (ADD c) (m ▷ n ▷ s)
            nothing → fail (n ▷ s)
        nothing → fail s
=   { induction hypothesis for y }
    case eval? x of
        just n  → exec (comp? y (ADD c)) (n ▷ s)
        nothing → fail s
=   { induction hypothesis for x }
    exec (comp? x (comp? y (ADD c))) s
```

which gives our third case for the compiler:

```
comp? (add x y) c = comp? x (comp? y (ADD c))
```

Note that, despite the possibility of the argument expressions `x` and `y` throwing exceptions, the case for `add` ends up having the same structure as the definition calculated in Section 5.3. In particular, because the type of `comp?` requires a handler to already be present on the stack, we don't have to explicitly worry about handling exceptions in this definition.

Case: `e = catch x h`

Finally, we consider the case for `catch`, for which both the expression and the handler could throw an exception. Although the solution to the case where the handler throws an exception is not immediately obvious, it falls naturally out of the calculation. We begin by applying `eval?`:

```
    case eval? (catch x h) of
        just n → exec c (n ▷ s)
        nothing → fail s
=   { definition of eval? }
    case eval? x of
        just n → exec c (n ▷ s)
        nothing → case eval? h of
            just n → exec c (n ▷ s)
            nothing → fail s
```

This time around, the nested `case` expression occurs on the failure branch, which is already in the correct form for the induction hypothesis to be applied:

```
    case eval? x of
        just n → exec c (n ▷ s)
        nothing → case eval? h of
            just n → exec c (n ▷ s)
            nothing → fail s
```

```
=   { induction hypothesis for h }
  case eval? x of
      just n → exec c (n ▷ s)
      nothing → exec (comp? h c) s
```

To match the form of our specification, we need to turn `exec (comp? h c) s` into a call to the function `fail`. This can be done by rewriting it using the definition for `fail` invoked with handler code on top of the stack that we introduced previously:

```
  case eval? x of
      just n → exec c (n ▷ s)
      nothing → exec (comp? h c) s
=   { definition of fail }
  case eval? x of
      just n → exec c (n ▷ s)
      nothing → fail (comp? h c ▷ s)
```

Now the stack for the failure branch, `comp? h c ▷ s`, has the handler on top, so we cannot apply the induction hypothesis yet. However, we can rewrite the success branch using the `UNMARK` constructor, which allows the induction hypothesis to be applied:

```
  case eval? x of
      just n → exec c (n ▷ s)
      nothing → fail (comp? h c ▷ s)
=   { definition of exec }
  case eval? x of
      just n → exec (UNMARK c) (n ▷ comp? h c ▷ s)
      nothing → fail (comp? h c ▷ s)
=   { induction hypothesis for x }
  exec (comp? x (UNMARK c)) (comp? h c ▷ s)
```

Finally, we can then use the `MARK` constructor to transform the term being manipulated into the same form as the left-hand side of our specification,

```
    exec (comp? x (UNMARK c)) (comp? h c ▷ s)
=   { definition of exec }
    exec (MARK (comp? h c) (comp? x (UNMARK c))) s
```

which gives our final case for the definition of `comp?`:

```
comp? (catch x h) c = MARK (comp? h c) (comp? x (UNMARK c))
```

We now see that the case where the handler throws an exception is handled by the recursive call to `comp?`, which requires a handler to already be on the stack to handle the exception. This means there can be multiple handlers on the stack at the same time: any handler which can itself throw an exception requires another handler to be present to catch the resulting exception.

The top-level compilation function `compile` can be calculated in a similar manner to Section 5.3, which gives rise to a new code constructor `HALT` as previously.

```
compile : b ≡ false → Exp b → Code S (nat :: S)
compile p e = comp p e HALT
```

6.7 Summary

In conclusion, we have now calculated the target language, compiler, and virtual machine for our expression language with exceptions, as summarised below.

Target language:

```
data Ty : Set where
  nat : Ty
  han : List Ty → List Ty → Ty
```

```
El : Ty → Set
El nat        = ℕ
El (han S S') = Code S S'


data Code : List Ty → List Ty → Set where
   PUSH  : ℕ → Code (nat :: S) S' → Code S S'
   ADD   : Code (nat :: S) S'  → Code (nat :: nat :: S) S'
   THROW : Code (S'' ++ han S S' :: S) S'
   MARK  : (h : Code S S') → (c : Code (han S S' :: S) S')
     → Code S S'
   UNMARK : Code (T :: S) S'  → Code (T :: han S S' :: S) S'
   HALT   : Code S S
```

Compiler:

```
comp? : Exp b → Code (nat :: (S'' ++ han S S' :: S)) S'
  → Code (S'' ++ han S S' :: S) S'
comp? (val n)   c = PUSH n c
comp? (add x y) c = comp? x (comp? {S'' = nat :: _} y (ADD c))
comp? throw     c = THROW
comp? (catch x h) c
  = MARK (comp? h c) (comp? {S'' = []} x (UNMARK c))


comp : b ≡ false → Exp b → Code (nat :: S) S' → Code S S'
comp p (val x) c
   = PUSH x c
comp p (add {false} {false} x y ) c
   = comp refl x (comp refl y (ADD c))
comp p (catch {false} {a} x h) c
  = comp refl x c
comp p (catch {true} {false} x h) c
   = MARK (comp refl h c) (comp? {S'' = []} x (UNMARK c))
```

```
compile : b ≡ false → Exp b → Code S (nat ∷ S)
compile p e = comp p e HALT
```

Virtual machine:

```
data Stack : List Ty → Set where
  ε : Stack []
  _▷_ : El T → Stack S → Stack (T ∷ S)

exec : Code S S' → Stack S → Stack S'
exec (PUSH x c) s           = exec c (x ▷ s)
exec (ADD c)    (m ▷ n ▷ s) = exec c ((n + m) ▷ s)
exec THROW      s           = fail s
exec (MARK h c) s           = exec c (h ▷ s)
exec (UNMARK c) (x ▷ _ ▷ s) = exec c (x ▷ s)
exec HALT       s           = s

fail : Stack (S'' ++ han S S' ∷ S) → Stack S'
fail {[]}   (h' ▷ s) = exec h' s
fail {_ ∷ _} (n ▷ s) = fail s
```

Note that in order for Agda to be able to type check these definitions, some extra type annotations are sometimes added to make the form of the stack explicit. For example, the annotation `{S'' = []}` in `comp` states that in this call to `comp?` there are no additional elements on the stack on top of the handler code pushed by `MARK`. In each case these additional annotations are straightforward to infer using Agda's automatic proof search mechanism.

6.8 Reflection

Our compiler for expressions that may throw an uncaught exception, `comp?`, is essentially the same as that developed by Bahr and Hutton (2015). How-

ever, its type here is much more precise, requiring an appropriate handler on the stack to catch the exception. This ensures that the resulting code will never cause a stack underflow by looking for a missing handler. The type of the handler is the key to this example, as it allows the compilation and execution functions to be total even with expressions which may throw exceptions. In the past it proved difficult even to write a total, dependently-typed compiler for exceptions, let alone prove it correct or calculate it.

Although the `exec` function is total, the Agda system cannot verify this automatically, because handler code taken from the stack is executed in a manner that is not structurally recursive. We therefore need to prove that the execution function is total. One approach (Bahr and Hutton, 2015) is to convert the execution function into a relation encoded as an inductive family. However, this approach has the drawback that the mechanised version of the calculation is then quite different to the pen-and-paper version. Another approach is to augment the execution function to make it structurally recursive, so that Agda can verify totality automatically. We adopted this approach, by providing two additional parameters to `exec`: a natural number representing an upper bound of the total size of the code and stack, and a proof of this bound. The execution function can then recurse on this natural number, establishing that the function is total.

Finally, we note that our Agda verification of the compiler calculation for exceptions requires the axiom of function extensionality—that two functions should be considered equal if they output the same value for every input—which is not present in the base type theory used by Agda. We added this property as an axiom, but an alternative method would have been to use a type theory which includes function extensionality, such as homotopy type theory.

Chapter 7

Simply-typed lambda calculus

Having demonstrated the core methodology in previous chapters, we will now calculate a compiler for a more complicated language: the simply-typed lambda calculus. This will further show the benefits of dependent types for the methodology, as they allow the types of lambda abstraction, function application, and the variable environment to be precisely specified.

Mechanising the calculation in Agda requires a different technique to prior examples, as using the same equational reasoning technique as before does not satisfy Agda's termination checker. We therefore perform the calculation by defining the semantics of the source and target languages as relations between their inputs and outputs, rather than functions. Chapter 8 will introduce a technique that will allow us to return to defining the language specifications as functions.

7.1 Source language

The source language we will consider in this chapter is the simply-typed lambda calculus (Church, 1940).

We begin with some implicit variable declarations:

```
variable
  n : ℕ
  s : Setting
  T : Set
```

We then define the set of types that expressions can have. There are just two constructors: base types and functions. For simplicity, we allow base types to map directly to Agda types, but defining a restricted set of base types would not affect the methodology. Functions in the simply-typed lambda calculus map one type to another. Expression types live in `Set₁` to prevent inconsistencies later on.

```
data ExpTy : Set₁ where
  ty : (T : Set) → ExpTy
  fn : (a : ExpTy) → (b : ExpTy) → ExpTy
```

The environment type `EnvT` is a vector of expression types. This will allow environments to be precisely typed, with an element for each type in the vector. We use de Bruijn indices (De Bruijn, 1972) for indexing into the variable environment for simplicity and to avoid issues with name shadowing.

```
EnvT : (n : ℕ) → Set₁
EnvT n = Vec ExpTy n
```

As usual, we define some implicit variables which scope over the whole file to avoid having to re-define them every time they are used:

```
variable
  A B : ExpTy
  envT envT' envT'' : EnvT n
```

Expressions in the simply-typed lambda calculus are indexed by the variable environment that the expression operates in, and the expression's type. This allows expressions to be typed precisely, and allows lookups into the environment to be total functions.

```
data STLC : EnvT n → ExpTy → Set₁ where
```

We will now define the constructors for the `STLC` type, which corresponds

to the building blocks of the language.

A value lifts an Agda type to a base type in the source language, and can exist in any environment.

```
val : T → STLC envT (ty T)
```

Variables require an index into the environment. We use Agda's `Fin` type here, which is a finite set of size `n`, providing a total index into the environment (which also has size `n`). This allows the expression to have the type which corresponds to the referenced variable in the environment.

```
var : (i : Fin n) → STLC envT (lookup envT i)
```

A lambda abstraction is a function `A → B`. The argument to the constructor is an expression returning type `B`, requiring a variable in the environment of type `A`, and the fully-applied constructor is an expression with a function type, `fn A B`:

```
lam : (STLC (A Vec.∷ envT) B) → STLC envT (fn A B)
```

A term with a function type `A → B` can be applied to an argument `A`, giving a resulting type `B`:

```
app : STLC envT (fn A B) → STLC envT A → STLC envT B
```

To simplify conversions between source-language expressions and target-language code, we define a `Setting` type to specify which setting an expression or value exists in.

```
data Setting : Set where
  E : Setting --source-language expressions
  C : Setting --target-language code


variable
  s : Setting
```

The type `Lift` allows us to lift a type `T` into an arbitrary universe level. This will be used in the next set of definitions to lift values to $\texttt{Set}_1$.

```
data Lift {l} : Set → Set l where
  lift : T → Lift T
```

We also define a function, `Val`, to give us the interpretation of types in Agda, and an `Env` type for the variable environment. These are parameterised by the `Setting` we are considering.

Base types, `ty T`, are simply lifted to their corresponding Agda type.

Functions in the source-language are interpreted as closures, containing the expression that forms the body of the function, and the environment the function was created in. This is represented as a tuple, containing the environment type (and its length), the environment, and the expression containing the function body.

We do not define the interpretation of functions in the target language yet, as tihs will be defined during the calculation.

`Val` is mutually-defind with the `Env` type, as the representation for functions contains an environment, and the environment contains types given by `Val`. The types of elements in the environment are given by an `EnvT n`, and it is defined in a similar way to the `Stack` types we have previously defined.

```
  mutual
    Val : Setting → ExpTy → Set₁
    Val s (ty T) = Lift T
    Val E (fn a b) =
      Σ ℕ
        (λ n → Σ (EnvT n)
                 (λ envT → Env E env × STLC (a Vec.∷ envT) b))
    Val C (fn a b) = ?


    data Env : Setting → EnvT n → Set₁ where
      εₛ : Env s Vec.[]
      _▷ₛ_ : {A : ExpTy}
           → (a : Val s A)
           → Env s envT
```

```
          → Env s (A Vec.∷ envT)


  variable
    env : Env C envT
```

The semantics of the source language are defined as follows:

```
  get : (i : Fin n) → Env s envT → Val s (lookup envT i)
  get zero (x ▷ₛ s) = x
  get (suc i) (x ▷ₛ s) = get i s


  {-# TERMINATING -#}
  eval : STLC envT A → Env E envT → Val E A
  eval           (val x)   env = lift x
  eval           (var i)   env = get i env
  eval {n}{envT} (lam e)   env = n , env , env , e
  eval           (app x y) env = case (eval x env) of
    (n , envT' , env' , e) = eval e (eval y env ▷ₛ env')
```

We mark this function as `TERMINATING` because it is not structurally recursive, so Agda's type-checker can't prove that it is terminating. This problem will be addressed in the next chapter.

7.2 Specification

As before, we will define the specification of what it means for a compiler to be correct, and use that to calculate the compiler.

We again choose to target a well-typed stack machine, where the type of the stack is tracked by the code type. The stack type is a simple list of expressions, and the stack is a list indexed by this list:

```
  StackTy : Set₁
  StackTy = List ExpTy
```

```
variable
  S S' S'' S0 S1 S2 : StackTy


data Stack : StackTy → Set₁ where
  ε : Stack []
  _▷_ : Val C A → Stack S → Stack (A :: S)


variable
  sta : Stack S
```

Our target-language code type is now extended to be indexed over not only the stack input and output types, but also by the type of the variable environment in which the code operates. This allows type-safe handling of variable lookup.

Note that values are stored on the stack according thier type's interpretation by `Val C`. As noted before, we have not yet defined any cases for `Val C` yet, as these will be discovered during the calculation process.

As before, we define the signature of the `Code` type and the `exec` and `comp` functions here, but the constructors will be discovered as part of the calculation process.

```
data Code : EnvT n → StackTy → StackTy → Set₁ where


exec : Code envT S S' → Env envT → Stack S → Stack S'


comp : STLC envT A → Code envT (A :: S) S' → Code envT S S'
```

We also need some functions to convert from the expression representation of values to the code representation.

```
valExpToCode : Val E A → Val C A
valExpToCode {ty T} x = x
valExpToCode {fn A B} (n , envT , env , e) = ?
```

The case for simple values is trivial, as the source- and target-language

representation of values is the same. We have not yet defined our target-language representation of functions, so we will not define this case yet.

Converting a whole environment simply requires mapping over each of the variables in the environment:

```
envExpToCode : Env E envT → Env C envT
envExpToCode εₛ = εₛ
envExpToCode (a ▷ₛ x) = (valExpToCode a) ▷ₛ (envExpToCode x)
```

Our compiler correctness equation is similar to the previous examples, with the appropriate conversion functions added:

```
exec (comp e c) (envExpToCode env) sta
  = exec c (envExpToCode env) (valExpToCode (eval env e) ▷ sta)
```

(7.1)

That is, evaluating an expression `e` and then executing a code continuation `c` is equivalent to compiling `e` with continuation `c`, and executing it.

7.3 Calculation

We perform the calculation as before, by induction over the expression type. We start with the right-hand side of Equation (7.1) and aim to transform it into the form `exec c' (envExpToCode env) sta` for some code `c'`. We can then take `comp e c = c'` as a defining equation for our compiler `comp`.

Case: **`e = val x`**

The base case for values is straightforward:

```
    exec c (envExpToCode env) (valExpToCode (eval env (val x)) ▷ sta)
=   { apply eval }
```

```
  exec c (envExpToCode env) (valExpToCode (lift x) ▷ sta)
=  { apply valExpToCode where A = ty A }
  exec c (envExpToCode env) (lift x sta)
=  { define: exec (PUSH x c) env sta =
                exec c env (lift x ▷ sta) }
  exec (PUSH x c) (envExpToCode env) sta
```

Giving us the first case for the compiler: `comp (val x) c = PUSH x c`.

Case: `e = var i`

The base case for variables begins as all cases do:

```
  exec c (envExpToCode env) (valExpToCode (eval env (var i)) ▷ sta)
=  { apply eval }
  exec c (envExpToCode env) (valExpToCode (get i env) ▷ sta)
```

We can continue here by noticing that `valExpToCode` distributes over `get`. Specifically, `valExpToCode (get i env) = get i (envExpToCode env)`. This can be proved by induction over `i`:

```
valExpDistributes
  : (i : Fin n) → (env : Env E envT)
  → (valExpToCode (get i env) ≡ get i (envExpToCode env))
valExpDistributes zero (a ▷ₛ env) = refl
valExpDistributes (suc i) (a ▷ₛ env) = valExpDistributes i env
```

This allows us to continue the calculation:

```
=  { valExpToCode distributes over get }
  exec c (envExpToCode env) (get i (envExpToCode env)) ▷ sta)
=  { define: exec (LOOKUP i c) env sta =
                exec c env (get i env ▷ sta) }
  exec (LOOKUP i c) (envExpToCode env) sta
```

This gives us the next case for the compiler: `comp (var i) c = LOOKUP i c`.

Case: `e = lam e`

In the case for lambda abstractions, we begin as usual with the right-hand side of Equation (7.1), and apply functions where possible.

```
    exec c (envExpToCode env)
      (valExpToCode (eval env (lam e)) ▷ sta)
=   { apply eval }
    exec c (envExpToCode env)
      (valExpToCode (n , envT , env , e) ▷ sta)
```

We are now stuck, as we have not defined the case for `valExpToCode` for functions, and our `exec` call does not match any of the existing cases. However, we can follow the usual pattern of aiming to translate this into `exec c' (envExpToCode env) sta` for some code `c`. This is captured by the following equation. Note that `n` and `envT` are implicit arguments to `exec`, but here they are made explicit by wrapping in braces, as they are passed explicitly to the closure.

```
exec {n}{envT} c' (envExpToCode env) sta =
  exec c (envExpToCode env)
    (valExpToCode (n , envT , env , e) ▷ sta)
```

We could solve this equation now, packaging up the unbound variable `e` in a constructor on the left-hand side. However, this would require passing around source-language expressions and doing compilation at run-time. While the methodology doesn't prevent us from writing a compiler that does this, we will make a decision here that we do not want uncompiled source expressions existing at runtime, and will aim to find a way to package up the code into a closure on the stack, in its compiled form.

We can find a definition of `valExpToCode` which matches this constraint. The type of `valExpToCode : Val E A → Val C A` requires us to know what `Val C A` evaluates to for functions, but we have not defined this yet. The input value is a 4-tuple of the size of the environment, the environment type, the environment, and the expression. We can convert this to a representation in the abstract machine by converting the environment with `envExpToCode`, and compiling the input expression into abstract machine code.

However, to compile an expression, we need a code continuation. This code continuation will run at the end of the function, returning control back to the original context. Therefore, we will call the new constructor for this continuation `RET`.

Before we define the type of `RET`, we need to provide some more definitions. The new constructor will involve manipulating closures on the stack, so we will need to define a new constructor to `ExpTy`, `cloT`, representing the type of of closures.

```
data ExpTy : Set₁ where
  ...
  cloT : EnvT n → StackTy → StackTy → ExpTy
```

We need to define a runtime representation for these. Closures of this form will not exist in the expression domain, so we can lift the bottom type to the appropriate universe level. In the domain of the abstract machine code, we define a new type, `Clo`, which packages up the closure's code and the environment it is closed over.

```
data Clo : EnvT n → StackTy → StackTy → Set₁ where
  clo : Env C envT → Code envT S S' → Clo envT S S'


Val : Setting → ExpTy → Set₁
...
Val E (cloT envT S S') = Lift ⊥
```

```
Val C (cloT envT S S') = Clo envT S S'
```

We can now begin to define the representation of functions in the domain of the abstract machine code.

Again, it is a 4-tuple consisting of the size of the environment, and the type of the environment. The environment itsel is now in the abstract machine code domain.

The code's type begins with the environment, which is extended with the function's parameter, `a`. The code requires the input stack to be pre-populated with a closure, which can be executed on the resulting stack after the function body has translated the input `a` into a `b`. Running this closure will return the final output type, `S'`.

```
Val : Setting → ExpTy → Set₁
...
Val C (fn a b) = {S S' : StackTy}{n : ℕ}{envT' : EnvT n} →
  Σ ℕ (λ n → Σ (EnvT n) (λ envT → Env C envT
         × Code (a Vec.:: envT) (cloT envT' (b :: S) S' :: S) S'))
```

We can now define the relevant case for `valExpToCode`, but we need the new code continuation, `RET`, to compile the input expression, `e`, with. As mentioned previously, this will execute the closure on the top of the stack, to return to the original context.

```
data Code : EnvT n → StackTy → StackTy → Set₁ where
  ...
  RET : {envT' : EnvT n}
    → Code envT (A :: cloT envT' (A :: S) S' :: S) S'
```

The case for `valExpToCode` will compile the input expression with this new code constructor as a code continuation.

```
valExpToCode {fn A B} (n , envT , env , e)
  = n , envT , (envExpToCode env) , (comp e RET)
```

We now return to our equation.

```
exec {n}{envT} c' (envExpToCode env) sta =
  exec c (envExpToCode env)
    (valExpToCode (n , envT , env , e) ▷ sta)
```

We can continue by applying the new definition of `valExpToCode`.

```
exec {n}{envT} c' (envExpToCode env) sta =
  exec c (envExpToCode env)
    ((n , envT , envExpToCode env , comp e RET) ▷ sta)
```

All instances of `env` are now guarded behind a call to `envExpToCode`, so we can simplify these calls to just `env`.

```
exec {n}{envT} c' env sta =
  exec c env ((n , envT , env , comp e RET) ▷ sta)
```

We can abstract over `comp e RET`, replacing it with `x`:

```
exec {n}{envT} c' env sta = exec c env (n , envT , env , x ▷ sta)
```

Now the code `x` and the code continuation `c` are the only variables which do not appear on the left-hand side, so we can package them up in a new constructor `ABS`, replacing the existentially-quantified `c'`:

```
exec {n}{envT} (ABS x c) env sta
  = exec c env (n , envT , env , x ▷ sta)
```

We now have a definition we can apply to allow us to finish the calculation.

```
exec c (envExpToCode env)
  (n , envT , envExpToCode env , comp e RET ▷ sta)
=  { define exec {n}{envT} (ABS x c) env sta
       = exec c (envExpToCode env) ((n , envT , env , x) ▷ sta) }
exec (ABS (comp e RET) c) (envExpToCode env) sta
```

This gives us our third case for the `comp` function:

```
comp (lam e) c = ABS (comp e RET) c
```

Notice that, as mentioned earlier, we have avoided doing compilation at run-time, and moved it to a recursive call at compile-time.

Case: `e = app x y`

We begin as usual for the `app` case, applying functions until we are stuck.

```
    exec c (envExpToCode env)
      (valExpToCode (eval env (app x y)) ▷ sta)
=   { apply eval }
    exec c (envExpToCode env) (valExpToCode
      (case (eval env x) of (n , envT', env', e)
        → eval e (eval y env ▷ₛ env'))
      ▷ sta)
=   { lift case to top level }
    case (eval env x) of
      (n , envT', env', e) → exec c (envExpToCode env)
          (valExpToCode (eval e (eval y env ▷ₛ env')) ▷ sta)
```

We are stuck, but we can continue by defining a case for `exec` to continue as normal. We can define the `exec` case for the `RET` constructor that was defined in the previous section, which pops the closure from the stack, and runs the closure's code.

In this definition, the environment on the left-hand side, `env`, is not matched on the right-hand side. In the execution of the abstract machine, that means the environment is no longer needed and is thrown away, as we continue execution from the environment contained in the return closure on the stack. In the calculation, this means we are free to instantiate it as we like. It is not clear at this point what to choose, but the future calculation step of attempting to apply the induction hypothesis for `y` makes

the choice clear: it shoud be the result of evaluating `y`, prepended onto the environment.

```
    case (eval env x) of (n , envT', env', e)
      → exec c (envExpToCode env)
          (valExpToCode (eval e (eval y env ▷ₛ env')) ▷ sta)
=   { define exec RET env (x ▷ (clo env' c' ▷ sta))
               = exec c' env' (x ▷ sta) }
    case (eval env x) of (n , envT', env', e)
      → exec RET (envExpToCode (eval y env ▷ₛ env))
          (valExpToCode (eval e (eval y env ▷ₛ env'))
          ▷ clo env c
          ▷ sta)
```

We can now apply the induction hypothesis for `e`, which comes from the result of evaluating `x`:

```
=   { induction hypothesis for e }
    case (eval env x) of (n , envT', env', e)
      → exec (comp e RET) (envExpToCode ((eval y env) ▷ₛ env'))
          (clo (envExpToCode env) c ▷ sta)
```

We are stuck again. However, we can define a new constructor, `APPLY`, which will take an argument an a closure from the stack, and execute the closure's code with the argument prepended to the environment. It will also push a return closure to the stack, needed by `RET` to return to the current environment and code continuation later.

The rest of the calculation follows naturally, manipulating the subject to a form where the induction hypothesis for each variable can be applied.

```
=   { define exec (APPLY c) env
```

```
                (v ▷ (n , envT' , env' , c') ▷ sta)
          = exec c' (v ▷ₛ env') (clo (envExpToCode env) c ▷ sta) }
  case (eval env x) of (n , envT', env', e)
    → exec (APPLY c) (envExpToCode env)
        (valExpToCode (eval y env)
        ▷ (n , envT' , envExpToCode env' , comp e RET)
        ▷ sta)
=  { induction hypothesis for y }
  case (eval env x) of (n , envT', env', e)
    → exec (comp y (APPLY c)) (envExpToCode env)
        ((n , envT' , envExpToCode env' , comp e RET) ▷ sta)
=  { unapply valExpToCode }
  case (eval env x) of (n , envT' , env' , e)
    → exec (comp y (APPLY c)) (envExpToCode env)
        (valExpToCode (n , envT' , env' , e) ▷ sta)
=  { collapse case statement }
  exec (comp y (APPLY c)) (envExpToCode env)
    (valExpToCode (eval env x) ▷ sta)
=  { induction hypothesis for x }
  exec (comp x (comp y (APPLY c))) (envExpToCode env) sta
```

This gives us our final case for `comp`:

```
comp (app x y) c = comp x (comp y (APPLY c))
```

7.4 Mechanisation

Unfortunately, Agda does not recognise the source language semantics as terminating, due to the recursive call to `eval` in the `app` case, which is not structurally recursive.

```
eval (app x y) = case (eval x env) of (n , envT' , env' , e)
```

```
                → eval e (eval y env ▷ₛ env')
```

To verify this calculation in Agda, we implement evaluation as a big-step relation, rather than a function.

We first define evaluation of the source language as a big-step operational semantics. The relation `e ⇓[ env ] v` means that the expression `e`, when evaluated in the environment `env`, produces the value `v`.

```
data _⇓[_]_ : STLC envT A → Env E envT → Val E A → Set₁ where
  evalVal : {x : T}{env : Env E envT}
    → val x ⇓[ env ] lift x
  evalVar : {i : Fin n}{env : Env E envT}
    → var i ⇓[ env ] get i env
  evalLam : {n : ℕ}{envT : EnvT n}
    {x : STLC (A Data.Vec.∷ envT) B}{env : _}
    → lam x ⇓[ env ] (n , envT , env , x)
  evalApp : {env : Env E envT}{env' : Env E envT'}
    {x : STLC envT (fn A B)}{y : STLC envT A}{y' : _}
    {e : STLC (A Vec.∷ envT') B}{xy : _}
    → x ⇓[ env ] (n , envT' , env' , e)
    → y ⇓[ env ] y'
    → e ⇓[ y' ▷ₛ env' ] xy
    → app x y ⇓[ env ] xy
```

We then translate the target-machine execution function that we calculated in section 7.3 into a small-step operational semantics. The relation `c , env , s ==> c' , env' , s'` says that executing one step of the target machine on the triple `c , env , s`, produces the triple `c' , env , s'`.

```
data _,_,_==>_,_,_ : Code envT S S'' → Env C envT → Stack S
  → Code envT' S' S'' → Env C envT' → Stack S' → Set₁ where
  execHALT : HALT , env , sta ==> HALT , env , sta
  execLOOKUP : {i : Fin n}
    {c : Code envT (lookup envT i ∷ S) S'}
```

```
    → LOOKUP i c , env , sta ==> c , env , (get i env ▷ sta)
  execPUSH : {x : T}{c : Code envT (ty T :: S) S'}
    → PUSH x c , env , sta ==> c , env , (lift x ▷ sta)
  execABS : {x : {S S' : StackTy}{n : ℕ}{envT' : EnvT n}
        → Code (A Vec.:: envT) (cont envT' (B :: S) S' :: S) S'}
    {c : Code envT (fn A B :: S) S'}
    → ABS x c , env , sta
      ==> c , env , ((_ , (_ , (env , x))) ▷ sta)
  execAPPLY : {c : Code envT (B :: S) S'}{v : Val C A}
    {f : Val C (fn A B)}
    → {env : Env C envT}
    → {sta : Stack S}
    → {env' : Env C envT'}
    → {c' : Code (A Vec.:: envT') (cont envT (B :: S) S' :: S) S'}
    → f {S} {S'} ≡ (n , envT' , env' , c')
    → APPLY c , env , (v ▷ (f ▷ sta))
      ==> c' , (v ▷ₛ env') , (clo env c ▷ sta)
  execRET : {S S' : StackTy}{sta : Stack S}{x : Val C A}
    {env' : Env C envT'}{c' : Code envT' (A :: S) S'}
    → RET , env , (x ▷ (clo env' c' ▷ sta))
      ==> c' , env' , (x ▷ sta)
```

We now define execution of the target machine as big-step operational semantics, in terms of the prior small-step operational semantics. This is the actual relation we will use in the mechanisation of the calculation. The relation `c , env , s =>> c' , env' , s'` means that the former triple can become the latter triple after any number of execution steps.

We define the big-step operational semantics of execution with a reflexivity and a transitivity relation, as well as some helper functions to make it easier to mechanise the calculation later.

```
data _,_,_=>>_,_,_ : Code envT S S'' → Env C envT → Stack S
  → Code envT' S' S'' → Env C envT' → Stack S' → Set₁ where
```

```
  execRefl : {c : Code envT S S'}{env : Env C envT}
    {sta : Stack S}
    → c , env , sta =>> c , env , sta
  execTrans : {c : Code envT S0 S}{env : Env C envT}
    {sta : Stack S0} {c' : Code envT' S1 S}{env' : Env C envT'}
    {sta' : Stack S1}{c'' : Code envT'' S2 S}
    {env'' : Env C envT''}{sta'' : Stack S2}
    → c , env , sta ==> c' , env' , sta'
    → c' , env' , sta' =>> c'' , env'' , sta''
    → c , env , sta =>> c'' , env'' , sta''


execLift : {c : Code envT S S''}{env : Env C envT}
    {sta : Stack S} {c' : Code envT' S' S''}
    {env' : Env C envT'}{sta' : Stack S'}
    → c , env , sta ==> c' , env' , sta'
    → c , env , sta =>> c' , env' , sta'
execLift p = execTrans p execRefl


execTrans' : {c : Code envT S0 S}{env : Env C envT}
    {sta : Stack S0}{c' : Code envT' S1 S}
    {env' : Env C envT'}{sta' : Stack S1}
    {c'' : Code envT'' S2 S}{env'' : Env C envT''}
    {sta'' : Stack S2}
    → c' , env' , sta' =>> c'' , env'' , sta''
    → c , env , sta =>> c' , env' , sta'
    → c , env , sta =>> c'' , env'' , sta''
execTrans' y execRefl = y
execTrans' z (execTrans x y) = execTrans x (execTrans' z y)
```

We can now proceed to converting the calculation into a form that uses the operational semantics we have just defined.

We first define the helper function `getConvEnv` to prove that looking up a

value from an environment in the code setting is the same as looking up a value from said environment in the expression setting and converting it into the code setting.

```
getConvEnv : (i : Fin n)(env : Env E envT)(sta : Stack S)
  → (get i (envExpToCode env) ▷ sta)
  ≡ (valExpToCode (get i env) ▷ sta)
getConvEnv zero (a ▷ₛ env) sta = refl
getConvEnv (suc i) (a ▷ₛ env) sta = getConvEnv i env sta
```

The `calculation` function, when given an expression `e`, a proof that `e` evaluates to `e'` in a given environment, and a code continuaiton `c`, gives a proof that compiling `e` with code continuation `c` and running it on our target machine will, after some finite number of execution steps, result in a machine state that will execute `c` with the result of evaluating `e` on the top of the stack. In other words, running the compiled code for `e` will push the result of evaluating `e` onto the stack, which is a proof of correctness of our compiler.

```
calculation :
  (e : STLC envT A)
  → {e' : _}
  → (env : Env E envT)
  → (sta : Stack S)
  → (c : Code envT (A :: S) S')
  → e ⇓[ env ] e'
  → comp e c , envExpToCode env , sta
    =>> c , envExpToCode env , (valExpToCode e' ▷ sta)
```

The first few cases are straightforward conversions from the calculation in section 7.3. The steps are annotated with comments to show how the steps relate to the original calculation.

```
calculation (val i) env sta c evalVal = execTrans'
  -- exec c env (eval (val x) env ▷ sta)
  --   apply eval
```

```
      execRefl
      -- exec c env (lift x ▷ sta)
      --   define exec (PUSH x c) env sta
      --            = exec c env (lift x ▷ sta)
      (execLift execPUSH)
      -- exec (PUSH i c) env sta
    calculation (var i) env sta c evalVar
      rewrite sym (getConvEnv i env sta) = execTrans'
      -- exec c env (eval (var i) env ▷ sta)
      --   apply eval
      execRefl
      -- exec c env (get i env ▷ sta)
      --   define exec (LOOKUP i c) env sta
      --            = exec c (get i env ▷ sta)
      (execLift execLOOKUP)
      -- exec (LOOKUP i c) env sta
    calculation (lam e) env sta c evalLam = execTrans'
      -- exec c env (eval (lam e) env ▷ sta)
      --   apply eval
      execRefl
      -- exec c env (valExpToCode (n , envT , env , e) ▷ sta)
      --   apply valExpToCode
      -- exec c env (n , envT , envExpToCode env , comp e RET ▷ sta)
      --   define exec {n}{envT} (ABS x c) env sta
      --            = exec c env ((n , envT , env , x) ▷ sta)
      (execLift execABS)
      -- exec (ABS (comp e RET) c) env sta
```

The `with` notation in the application case allows us to bind intermediate results to new names: in this case, binding the induction hypotheses for `x`, `y`, and `e` to `ihX`, `ihY`, and `ihE` respectively.

```
    calculation (app x y) env sta c
```

```
  (evalApp {env' = env'}{y' = y'}{e = e}{xy = xy}
     evalx evaly evale)
  with calculation x env sta (comp y (APPLY c)) evalx
  | calculation y env (_ ▷ sta) (APPLY c) evaly
  | calculation e (y' ▷ₛ env') (clo (envExpToCode env) c ▷ sta)
      RET evale
calculation (app x y) env sta c (evalApp evalx evaly evale)
  | ihX | ihY | ihE =
  -- exec c env (eval (app x y) env ▷ sta)
  --   apply eval
  -- exec c env (xy ▷ sta)
  --   define exec RET env (x ▷ (clo env' c' ▷ sta))
  --             = exec c' env' (x ▷ sta)
  execTrans' (execLift execRET)
  -- exec RET (valExpToCode y' ▷ₛ env') (xy ▷ clo env c ▷ sta)
  --   induction hypothesis for e
  (execTrans' ihE
  -- exec (comp e RET) (valExpToCode y' ▷ₛ env')
  --   (clo env c ▷ sta)
  --   define exec (APPLY c) env
  --             (v ▷ ((n , envT' , env' , c') ▷ sta))
  --             = exec c' (v ▷ₛ env') (clo env c ▷ sta)
  (execTrans' (execLift (execAPPLY refl))
  -- exec (APPLY c) env (valExpToCode y'
  --                     ▷ ((n , envT' , env' , comp e RET)
  --                     ▷ sta))
  --   induction hypothesis for y
  (execTrans' ihY
  -- exec (comp y (APPLY c)) (envExpToCode env)
  --   (valExpToCode y'
  --    ▷ (n , envT' , env' , comp e RET)
```

```
--    ▷ sta)
--   induction hypothesis for x
ihX)))
-- exec (comp x (comp y (APPLY c))) (envExpToCode env) sta
```

7.5 Summary

In this chapter, we calculated a compiler for the simply-typed lambda calculus. Use of depedent types allowed us to precisely track the types of expressions as usual, ensuring that function applications are type-correct, and also allows us to track the types of variable environments for type-safe lookups from the environment.

Our initial attempt at the calculation encoding big-step operational semantics as Agda functions as we did previously did not allow us to mechanise the calculation in Agda, as the evaluation function was not structurally-recursive, and therefore Agda could not recognise it as terminating. Instead, we converted the evaluation and execution functions into relations that encode their big-step operational semantics. This allowed us to mechanise the calculation in a way that Agda recognised was terminating.

Chapter 8

Non-termination

In this chapter, we revisit the issue of non-termination, which made it difficult to mechanise the calculation of the compiler in the last chapter. We present an alternative way to handle non-termination by using the partiality monad, and by defining a notion of bisimilarity for possibly non-terminating terms. We initially demonstrate the technique in a simplified setting by extending the expression language from Chapter 5 with an infinitely-looping construct. The technique is demonstrated in a more complex example by revisiting the calculation of a compiler for the simply-typed lambda calculus, which is included in the appendices.

8.1 Introduction

In Chapter 7 we saw how to calculate a compiler for the simply-typed lambda calculus. However, the mechanisation of the calculation had to be implemented in a relational style, rather than using the equational reasoning we used to do the calculation, due to Agda's termination checker not being able to verify that the language was terminating. This, along with wishing to consider languages that are non-terminating, motivates the next extension of the compiler calculation technique.

We follow the example of Bahr and Hutton (2022), who introduced a compiler calculation methodology that allows us to consider the divergent behaviour of programs, as well as the convergent behaviour. The technique encodes the possible non-termination of both the source language and target abstract machine using the coinductively-defined partiality monad (Capretta, 2005). This permits non-termination by making the evaluation and execution functions productive. That is, they always either terminate, or can produce another part of a result, even if the overall result is infinite, and would therefore not terminate if the whole computation needed to be performed. Strict equality is not suitable for performing calculations on coinductive datatypes, but we can define a notion of bisimilarity to allow calculations to be performed with the partiality monad.

Extending our technique of calculating compilers for typed languages with support for non-total languages brings us closer to calculating compilers for real-world languages, as we can now consider non-termination.

In this chapter we begin by defining the partiality monad, bisimilarity, and the related notion of i-bisimilarity. We then specify a minimal language which encodes non-termination using the partiality monad, and calculate a compiler for this language. Finally, we show how this style of calculation can be used to calculate a compiler for the simply-typed lambda calculus, which can be mechanised directly in Agda.

8.2 Partiality monad

The partiality monad (Capretta, 2005; Danielsson, 2012) allows us to consider the behaviour of partial functions. Conceptually, the partiality monad is a coinductive data type, allowing it to be infinitely-sized. A coinductive type is given by a greatest fixpoint, instead of the least fixpoint that is used for inductive types. If Agda supported use of a `codata` keyword to define

coinductive types, we could define it as follows:

```
codata Partial (A : Set) : Set where
  now   : A → Partial A
  later : Partial A → Partial A
```

The idea is that the `now` constructor encodes a value that is immediately available, while the `later` constructor delays a computation by one timestep.

Because the partiality monad is a coinductive data type, elements of the type do not need to have a `now` constructor as a base case. Therefore, the following definition is a valid and useful inhabitant of `Partial A`, representing a non-terminating computation:

```
never : {A : Set} → Partial A
never = later never
```

The above definitions capture the key idea of the partiality monad. However, Agda does not support defining coinductive types with the `codata` keyword. Instead, we can define coinductive types using coinductive record types, which is now recommended over the 'musical notation' that was used in the past (Agda Developers, 2024). The following implementation is adapted from the code of Bahr and Hutton (2022), modified to be universe-polymorphic by generalising over the universe level `ℓ : Level`, allowing support for partial types in `Set₁`.

```
variable
  ℓ : Level


mutual
  data Partial (A : Set ℓ) (i : Size) : Set ℓ where
    now   : A → Partial A i
    later : ∞Partial A i → Partial A i


  record ∞Partial (A : Set ℓ) (i : Size) : Set ℓ where
    coinductive
```

```
    constructor delay
    field
      force : {j : Size< i} → Partial A j
```

The definition consists of a pair of mutually-defined types: an inductive data type `Partial` encoding the product of the `now` and `later` constructors, and a coinductive record `∞Partial`. The `force` field of `∞Partial` ensures that the size of the recursive `Partial` element always decreases.

Our `Partial` type and the helper type `∞Partial` are indexed by sized types (Hughes et al., 1996), which are used to allow Agda's positivity checker to admit functions on these types. `Size` is the type of possibly-infinite natural numbers, which allows elements of the `Partial` type to either be finite, ending in a `now` constructor, or to be an infinite stream of `later` constructors. The implicit argument `j : Size< i` to the `force` constructor ensures that the guarded component is structurally smaller if we are working with a finite object, or permits an infinite chain for an infinite object.

We also have a new definition of `never`, again defined as a mutual pair of definitions on `Partial` and `∞Partial`:

```
mutual
  never : ∀ {A i} → Partial {ℓ} A i
  never = later ∞never

  ∞never : ∀ {A i} → ∞Partial {ℓ} A i
  force ∞never = never
```

`Partial` forms a monad (Moggi, 1989). Return is simply the `now` constructor, and the bind operator, `_>>=_`, is defined as follows:

```
mutual
  _>>=_ : ∀ {i A B} →
    Partial {ℓ} A i → (A → Partial {ℓ} B i) → Partial B i
  now   x >>= f = f x
  later x >>= f = later (x ∞>>= f)
```

```
_∞>>=_ :  ∀ {i A B} →
  ∞Partial {ℓ} A i → (A → Partial {ℓ} B i) → ∞Partial B i
force (x ∞>>= f) = force x >>= f
```

Intuitively, the bind operator replaces the `now` constructor (if there is one) with a function, and `later` constructors remain in place.

Monads also need to satisfy a set of monad laws. Usually the laws hold up to equality, `≡`, but for coinductive types such as the partiality monad, the laws only hold up to bisimilarity, `≅`, which is defined in the next section:

```
return x >>= f   ≅ f x
mx >>= return    ≅ mx
(mx >>= f) >>= g ≅ mx >>= (λ x → (f x >>= g))
```

8.3 Bisimilarity

We define bisimilarity by first defining an inductive relation `p ⇓ v` which specifies when a computation `p : Partial a` converges to a value `v : a`:

$$\frac{}{\texttt{now v} \Downarrow \texttt{v}} \qquad\qquad \frac{\texttt{p} \Downarrow \texttt{v}}{\texttt{later p} \Downarrow \texttt{v}}$$

Two computations `p q : Partial a` are *weakly* bisimilar, written `p ≈ q`, if they converge to the same set of values:

$$\texttt{p} \approx \texttt{q} \quad \Leftrightarrow \quad \forall \texttt{v} : \texttt{a}.\ \ \texttt{p} \Downarrow \texttt{v} \ \ \textit{iff} \ \ \texttt{q} \Downarrow \texttt{v}$$

However, using weak bisimilarity with a transitive approach to reasoning leads to contradiction, so it is not suitable for compiler calculation (Bahr and Hutton, 2022). Instead, we define the notion of *strong* bisimilarity, or simply bisimilarity, by requiring the computations to also take the same number of steps. We start by defining a step-indexed convergence relation

$\mathtt{p} \Downarrow_i \mathtt{v}$, where the index $i : \mathbb{N}$ gives an upper bound for the number of steps taken to converge:

$$\frac{}{\mathtt{now\ v} \Downarrow_i \mathtt{v}} \qquad \frac{\mathtt{p} \Downarrow_i \mathtt{v}}{\mathtt{later\ p} \Downarrow_{i+1} \mathtt{v}}$$

The base case does not start at $i \equiv 0$ as the index is an upper bound on the number of steps rather than a strict count.

Two computations `p q : Partial a` are then bisimilar, $\mathtt{p} \cong \mathtt{q}$ if they have the same step-counting convergence behaviour:

$$\mathtt{p} \cong \mathtt{q} \quad \Leftrightarrow \quad \forall \mathtt{v} : \mathtt{a}, i : \mathbb{N}.\ \ \mathtt{p} \Downarrow_i \mathtt{v} \ \ \textit{iff} \ \ \mathtt{q} \Downarrow_i \mathtt{v}$$

8.4 i-Bisimilarity

Strong bisimilarity is the notion of equality used for coinductive types. However, for proving bisimilarities, we use the related notion of step-indexed bisimilarity, or i-bisimilarity, written as $\mathtt{p} \cong_i \mathtt{q}$:

$$\mathtt{p} \cong_i \mathtt{q} \quad \Leftrightarrow \quad \forall \mathtt{v} : \mathtt{a}, j < i.\ \ \mathtt{p} \Downarrow_j \mathtt{v} \ \ \textit{iff} \ \ \mathtt{q} \Downarrow_j \mathtt{v}$$

We can prove bisimilarity by proving i-bisimilarity for all step counts i:

$$\mathtt{p} \cong \mathtt{q} \quad \Leftrightarrow \quad \forall i : \mathbb{N}.\ \ \mathtt{p} \cong_i \mathtt{q}$$

8.5 Looping language

We will first consider a minimal language required to demonstrate the technique. This is the simple expression-based language from Chapter 5, but with a simple *loop* construct added, which just loops forever. Clearly this

is not a particularly useful addition to the language, but it is sufficient to demonstrate the method.

8.5.1 Source language

As before, we have values of any type, addition of natural numbers, and a conditional statement branching on a Boolean. However, this time we also have a `loop` construct, which loops forever. The type of this is `Exp T` for any `T`, as it never returns.

```
variable T : Set


data Exp : Set → Set where
  val : T → Exp T
  add : Exp ℕ → Exp ℕ → Exp ℕ
  if : Exp Bool → Exp T → Exp T → Exp T
  loop : Exp T
```

The first significant difference in the methodologies is that the `eval` function for defining the language semantics now returns a value in the partiality monad.

The content of the function remains the same for the existing cases, but the structure of it changes to become monadic.

The new `loop` case makes a recursive call to `eval loop` as expected, but this needs to be wrapped in a `later` constructor, as it is not structurally recursive.

```
eval : Exp T → Partial T i
eval (val x) = return x
eval (add x y) =
  do n ← eval x
```

```
      m ← eval y
      return (n + m)
eval (if b x y) =
  do b' ← eval b
     if b' then (eval x) else (eval y)
eval loop = later (eval loop)
```

8.5.2 Compiler specification

We use the same `Stack` indexed data type as in previous examples.

```
data Stack : List Set → Set where
  ε : Stack []
  _▷_ : T → Stack S → Stack (T :: S)
```

The types for `Code`, `comp`, and `compile` remain the same, with definitions being calculated as we go as usual.

```
data Code : List Set → List Set → Set₁ where
comp : Exp T → Code (T :: S) S' → Code S S'
compile : Exp T → Code S (T :: S)
```

The type for `exec` changes slightly. Rather than always returning a `Stack S'`, it now returns a value in the `Partial` monad, to match the semantics of our language. This means that the abstract machine can match the semantics of the `loop` construct, stepping forever without returning a value of type `Stack S'`.

```
exec : Code S S' → Stack S → Partial (Stack S') i
```

Our compiler correctness theorem has the same structure as previous examples, but takes on a different form. Firstly, equality is replaced by bisimilarity: the two sides need to correspond for any given number of simulation steps. Secondly, the two sides of the bisimilarity relation are in the `Partial` monad, so we can use monadic do-notation to describe the expression.

The specification for `comp`, using code continuations, is as follows:

$$\begin{array}{l} \texttt{(do v} \leftarrow \texttt{eval e} \\ \qquad \texttt{exec c (v} \triangleright \texttt{s))} \\ \cong_{\texttt{i}} \\ \texttt{exec (comp e c) s} \end{array} \tag{8.1}$$

This is conceptually the same as the previous specifications: evaluating an expression and running some code `c` with that expression on top of the stack is the same (now up to bisimilarity, rather than equality) as executing the result of compiling the expression with the code `c` as the continuation.

Finally, the full specification for `compile`, which does not take code continuations, is:

$$\begin{array}{l} \texttt{(do v} \leftarrow \texttt{eval e} \\ \qquad \texttt{return (v} \triangleright \texttt{s))} \\ \cong_{\texttt{i}} \\ \texttt{exec (compile e) s} \end{array} \tag{8.2}$$

8.5.3 Calculation

The calculation proceeds by coinduction, but this time it is over both the expression and the the number of steps in the bisimulation, `i`.

Case: `e = val x`

The first case we consider, as usual, is `e = val x`. This proceeds as normal, but we need an extra step to apply the definition of `>>=` for the `Partial` monad.

```
    do v ← eval (val x)
       exec c (v ▷ s)
≅i  { definition of eval }
    do v ← now x
       exec c (v ▷ s)
≅i  { definition of >>= }
    exec c (x ▷ s)
≅i  { define: exec (PUSH x c) s = exec c (x ▷ s) }
    exec (PUSH x c) s
```

This gives us our first compiler case, the same as for the convergent compiler.

```
comp (val x) c = PUSH x c
```

Case: **`e = add x y`**

The `add` case proceeds as normal, but we need to transform the expression according to the monad laws, and apply the bind operator, `>>=`, before we can define an equation for `exec` and apply the inductive hypotheses.

```
    do v ← eval (add x y)
       exec c (v ▷ s)
≅i  { definition of eval }
    do v ← do m ← eval x
              n ← eval y
              return (m + n)
       exec c (v ▷ s)
≅i  { monad laws }
    do m ← eval x
       n ← eval y
       v ← return (m + n)
       exec c (v ▷ s))
```

$\cong_i$ { definition of `>>=` }

```
do m ← eval x
   n ← eval y
   exec c ((m + n) ▷ s))
```

$\cong_i$ { define: `exec (ADD c) (n ▷ (m ▷ s)) = exec c ((m + n) ▷ s)` }

```
do m ← eval x
   n ← eval y
   exec (ADD c) (n ▷ (m ▷ s)))
```

$\cong_i$ { inductive hypothesis for `y` }

```
do m ← eval x
   exec (comp y (ADD c)) (m ▷ s))
```

$\cong_i$ { inductive hypothesis for `x` }

```
exec (comp x (comp y (ADD c))) s
```

Our second compiler case is again the same as for the convergent compiler:

```
comp (add x y) c = comp x (comp y (ADD c))
```

***Case:* `e = if b t f`**

The case for `if` is again similar, but requiring a rearrangement with the monad laws. We also use the fact that `>>=` distributes over `if`, so that the induction hypotheses can be applied separately for each branch of the `if`.

```
do v ← eval (if b t f)
   exec c (v ▷ s)
```

$\cong_i$ { definition of `eval` }

```
do v ← do b' ← eval b
          if b' then (eval t) else (eval f)
   exec c (v ▷ s)
```

$\cong_i$ { monad laws }

```
do b' ← eval b
   v ← if b' then (eval t) else (eval f)
```

```
      exec c (v ▷ s)
```

$\cong_{\mathrm{i}}$ { distributivity of `>>=` over `if` }

```
  do b' ← eval b
     if b' then (do v ← eval t
                    exec c (v ▷ s)) else (do v ← eval f
                                             exec c (v ▷ s))
```

$\cong_{\mathrm{i}}$ { coinduction hypothesis for `t` }

```
  do b' ← eval b
     if b' then (exec (comp t c) s) else (do v ← eval f
                                             exec c (v ▷ s))
```

$\cong_{\mathrm{i}}$ { coinduction hypothesis for `f` }

```
  do b' ← eval b
     if b' then (exec (comp t c) s) else (exec (comp f c) s)
```

$\cong_{\mathrm{i}}$ { define: `exec (IF t' f') (b' ▷ s) =`

```
                if b' then exec t' s else exec f' s }
  do b' ← eval b
     exec (IF (comp t c) (comp f c)) (b' ▷ s)
```

$\cong_{\mathrm{i}}$ { coinduction hypothesis for `b` }

```
  exec (comp b (IF (comp t c) (comp f c))) s
```

We have our third compiler case:

```
comp (if b t f) c = comp b (IF (comp t c) (comp f c))
```

Case: `e = loop`

The `loop` case is where we consider the divergent behaviour of the source program. This is where we need to perform coinduction over `i` as well as the expression.

We will first consider the base case, `i = zero`. We need to prove:

```
do v ← eval loop
```

```
    exec c (v ▷ s)
≅0
  exec (comp loop c) s
```

Recall the definition of i-Bisimilarity:

$$\mathtt{p} \cong_i \mathtt{q} \quad \Leftrightarrow \quad \forall \mathtt{v} : \mathtt{a}, j < i.\ \ \mathtt{p} \Downarrow_j \mathtt{v} \ \ \textit{iff} \ \ \mathtt{q} \Downarrow_j \mathtt{v}$$

In the case where $i = 0$, there are no $j < i$. Therefore, this is trivially true, as $a \cong_0 b$ forall a and b.

Now we conisder the case for `suc i`.

```
    do v ← eval loop
       exec c (v ▷ s)
≅i  { definition of eval }
    do v ← later (eval loop)
       exec c (v ▷ s))
≅i  { definition of >>= }
    later (do v ← eval loop
              exec c (v ▷ s))
```

This is the point that demonstrates that it is not enough to do coinduction just over the expression, as the expression we are applying the coinduction hypothesis to is not syntactically smaller than the original expression. However, as we are also performing coinduction over `i`, and because the expression is guarded by a call to `later`, we can apply our coinduction hypothesis here.

```
    later (do v ← eval loop
              exec c (v ▷ s))
≅i  { coinduction hypothesis }
```

```
later (exec (comp loop c) s)
```

We are now stuck. We can apply the usual technique of adding a new defining equation for `exec` here, but it results in a definition with a call to `comp` on the right-hand side. This is not desirable as we do not want to be doing compilation at runtime, but we can proceed with the calculation for now and fix this at the end.

```
  later (exec (comp loop c) s)
≅ᵢ  { define: exec LOOP s = comp loop c }
  exec LOOP s
```

We now have the definition for the `loop` case for our compiler:

```
comp loop s = LOOP
```

This definition allows us to fix our previous definition of `exec`, by substituting the right-hand side for `LOOP`:

```
exec LOOP s = LOOP
```

Finally, we can calculate a definition for `compile` using Equation (8.2), included here for convenience:

```
(do v ← eval e
    return (v ▷ s))
 ≅ᵢ
exec (compile e) s
```
(8.2 revisited)

Starting from the left-hand-side, we want to convert it to a form that allows Equation (8.1) to be applied:

```
do v ← eval e
```

```
    return (v ▷ s)
```

$\cong_{\mathrm{i}}$ { define: `exec HALT s = return s` }

```
do v ← eval e
   exec HALT (v ▷ s)
```

$\cong_{\mathrm{i}}$ { apply Equation (8.2) }

```
exec (comp e HALT) s
```

We have defined our final code constructor, `HALT`, and this is now in the form `exec c s`, so we have our defining equation for `compile`:

```
compile : Exp T → Code S (T :: S)
compile e = comp e HALT
```

Putting this all together, we have calculated the following definitions:

```
data Code : List Set → List Set → Set₁ where
  PUSH : T → Code (T :: S) S' → Code S S'
  ADD  : Code (ℕ :: S) S' → Code (ℕ :: ℕ :: S) S'
  IF   : Code S S' → Code S S' → Code (Bool :: S) S'
  LOOP : Code S S'
  HALT : Code S S


exec : Code S S' → Stack S → Partial (Stack S') i
exec (PUSH x c) s          = exec c (x ▷ s)
exec (ADD c) (m ▷ (n ▷ s)) = exec c ((n + m) ▷ s)
exec (IF c1 c2) (b ▷ s)    = if b then exec c1 s else exec c2 s
exec LOOP s                = later (exec LOOP s)
exec HALT s                = return s

comp : Exp T → Code (T :: S) S' → Code S S'
comp (val x) c    = PUSH x c
comp (add x y) c  = comp x (comp y (ADD c))
comp (if b x y) c = comp b (IF (comp x c) (comp y c))
```

```
comp loop c       = LOOP


compile : Exp T → Code S (T ∷ S)
compile e = comp e HALT
```

8.6 Simply-typed lambda calculus revisited

In the previous chapter, we calculated a compiler for the simply-typed lambda calculus. However, we noted that the mechanisation of the proof in Agda required an intermediate step of translating the calculations into a relational style to satisfy the termination checker.

This extra step can be avoided by using the partiality monad and coinduction, allowing the calculation to be mechanised directly in Agda. The code for this is provided in the appendix. This calculation relies on a version of the partiality monad code from Bahr and Hutton (2022), modified to mkae the partiality monad universe-polymorphic. This modified code is included in the supplemental material for the thesis.

8.7 Summary

In this chapter, we have seen that adapting the technique of coinductive compiler calculation to a dependently-typed setting is straigthforward. The two techniques work together well, allowing us to verify that compilers are correct for programs experiencing divergent behaviour, as well as convergent behaviour. Using the partiality monad allows us to further expand the set of languages we can calculate compilers for, as it allows the source language (and, therefore, the calculated abstract machine) to be non-terminating.

This also allows us to mechanise our proofs in a much more direct manner,

as the coinductive calculations can be written directly in Agda, eliminating the need to translate certain calculations to a relational style first, as was necessary in Chapter 7.

Part III

Conclusions

Chapter 9

Conclusion and Further Work

In this thesis we have explored techniques for calculating compilers using dependent types, and shown how to calculate compilers for a range of simple programming languages. Each chapter included a summary of what was achieved. We give some concluding remarks in this chapter, and discuss potential further developments of the techniques.

9.1 Conclusion

The compiler calculation methodology originally developed by Bahr and Hutton (2015) allows compilers to be calculated directly from the language specification, producing a proof of correctness for free, rather than having to write and prove the compiler correct separately. Additionally, it is straightforwrad to mechanise the calculations—all of the calculations shown in this thesis are mechanised in Agda, with the full code provided in Appendix A.

Moving to a dependently-typed setting, which has been the focus of this thesis, brings further benefits. The original work by Bahr and Hutton (2015) essentially didn't consider types at all, dealing only with integers.

The approach developed in this thesis allows us to consider multiple types, while keeping the source language semantics total. The previous work required all expressions to be one type (integers) to prevent the existence of malformed expressions, as there was no type checking involved. In this thesis, rather than providing a separate type-checking algorithm, typed expressions can be encoded using inductive families, indexing the expression with its type. Using inductive familiies in this way encodes the type information directly in the expression, which allows us to use it in the calculation process.

Use of dependent types also brings benefits at the level of the target langugae for the compiler, and how the resulting code is executed. In particular, the code now encodes a lot more information about how it is executed, such as the number and types of values on the stack, allowing the execution function for the target machine to be total. In the previous work, the execution function was not total, for example because the stack may underflow.

Making the source and target languages total also makes it easier to mechanise the calculations and resulting definitions, as many proof assistants require all functions to be total. In previous work, the calculations could not be directly translated to the proof assistant and required converting into a different form (such as a relational semantics) to ensure totality, but the approach presented here allows the calculations to be written in essentially the same manner as the paper form. Similar translations were required previously to make the target machine total.

9.2 Further work

Mechanisation

Currently our technique is a manual process that is applied by hand, or with the help of a proof assistant such as Agda or Coq. However, it is possible that much of the process could be automated to derive compilers in a mechanical manner from specifications of their correctness. Mechanisation options could range from a special-purpose library to assist in the process of equational reasoning (Mu et al., 2009), to providing support for the calculation process in a general-purpose equational reasoning system (Farmer et al., 2015), to a system that is custom-designed for our particular calculational approach Handley and Hutton (2018).

Nonetheless, any mechanical process would likely still involve some form of human interaction to make decisions that affect the compilation, such as whether certain terms should be evaluated at run-time or compile-time, or to select which particular design choice to pursue when more than one possibility arises during the compiler calculation process.

Source languages

To date, our new technique has been applied to idealised source and target languages. This approach has allowed us to develop the technique and demonstrate its utility, including being able to calculate a well-typed compiler for a language with exceptions, an example for which it had previously proved difficult to even write a well-typed compiler.

However, while these languages are very valuable for developing the methodology, they provide little practical value. A logical next step now is to apply the technique to more realistic languages, such as the core language of the

Glasgow Haskell Compiler, an explicitly-typed variant of the polymorphic lambda calculus (Sulzmann et al., 2007). This would drive developments in the methodology to handle the more advanced features of the core language, such as handling of polymorphism. Alternatively, we could consider other programming paradigms, such as object-oriented programming. For example, we could attempt to calculate a compiler for Featherweight Java, an object-oriented core language used in the study of object-oriented programming (Igarashi et al., 2001).

Target languages

In addition, the technique could also be adapted to target real hardware or virtual machines, building on Bahr and Hutton's (2020) recent work on register-based languages. However, considering realistic source and target languages also raises issues concerning totality, as many such languages are either not total, or would require non-trivial effort to verify totality within a proof assistant.

The current approach works by deriving the desired target language during the calculation process, but this approach would not be directly adaptable to targeting existing machines. The current approach is solving for three variables at the same time: the code type, the compiler, and the execution function. Fixing the target machine reduces this to one variable, the compiler itself. It remains to be seen how this will affect the calculations: on the one hand reducing the variables may make the calculation easier, but reducing the degrees of freedom may make the calculation harder. When we get stuck in the calculation the current approach makes progress by defining a new machine operation to solve the equation, whereas when targeting a fixed machine, that solution will need to be found in terms of the target machine's existing operations.

Calcluating type checkers

Finally, another possible topic for further research is whether the same concepts can be applied in other areas, such as type systems. For example, is it possible to calculate a type-checker, along with its correctness proof, starting from a set of typing rules? Can properties such as preservation and progress of types be automatically proven during the compilation process?

This would be beneficial as the technique demonstrated in this thesis requires the source program to already be encoded in an intrinsically-typed form. Real-world compilers would first need to type-check the user's program, to be able to either convert it to an intrinsically-typed form, or provide an error message if the program is not type-correct.

It could also further extend the range of source languages that can be handled by the technique, as it takes the more traditional approach of using a separate type-checking phase, rather than intrinsic typing.

Another approach would be to attempt to calculate the type system itself for a language. Given an untyped language with non-total execution semantics, can we calculate a type system that rules out cases that get stuck or do not terminate? For example, given the expression language with addition and conditional expressions, could we calculate the type system of integers and booleans that was used in Chapter 5?

Bibliography

Agda Developers. Agda language reference: Coinduction, 2024. URL https://agda.readthedocs.io/en/v2.6.20240714/language/coinduction.html.

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From Interpreter to Compiler and Virtual Machine: A Functional Derivation. Technical report, BRICS, 2003.

Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

Andrew Appel, Lennart Beringer, Adam Chlipala, Benjamin Pearce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. The science of deep specification, 2015. URL https://deepspec.org/.

Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming, Gothenburg*, 1999.

Patrick Bahr and Graham Hutton. Calculating Correct Compilers. *Journal of Functional Programming*, 25, September 2015.

Patrick Bahr and Graham Hutton. Calculating Correct Compilers II: Return of the Register Machines. *Journal of Functional Programming*, 30, August 2020.

Patrick Bahr and Graham Hutton. Monadic Compiler Calculation. *Proceedings of the ACM on Programming Languages*, 6(ICFP), August 2022.

Patrick Bahr and Graham Hutton. Calculating Compilers for Concurrency. *Proceedings of the ACM on Programming Languages*, 7(ICFP), August 2023.

Patrick Bahr and Graham Hutton. Beyond Trees: Calculating Graph-Based Compilers. *Proceedings of the ACM on Programming Languages*, 8(ICFP), August 2024.

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013. doi: 10.1017/S095679681300018X.

Edwin Brady and Kevin Hammond. A verified staged interpreter is a verified compiler. In *GPCE'06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, 2006.

Rod M. Burstall and Peter J. Landin. Programs and their proofs: an algebraic approach. volume 4. Edinburgh University Press, 1969.

Venanzio Capretta. General Recursion via Coinductive Types. *Logical Methods in Computer Science*, Volume 1, Issue 2, July 2005. doi: 10.2168/LMCS-1(2:1)2005. URL https://lmcs.episciences.org/2265.

Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI'07: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940. doi: 10.2307/2266170.

Nils Anders Danielsson. Operational semantics using the partiality monad. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 127–138, New York, NY, USA, 2012. Association for Computing Machinery. ISBN

9781450310543. doi: 10.1145/2364527.2364546. URL https://doi.org/10.1145/2364527.2364546.

Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2, nov 2003. ISSN 0163-5948. doi: 10.1145/966221.966235. URL https://doi.org/10.1145/966221.966235.

Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

W Diffie. Mechanical verification of a compiler correctness proof. A.I. Memo, Stanford University, 1972.

Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

Edsger Wybe Dijkstra. Notes on Structured Programming. EWD249, April 1970. URL https://www.cs.utexas.edu/~EWD/ewd02xx/EWD249.PDF.

Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

Conal Elliott. Compiling to Categories. In *Proceedings of the ACM on Programming Languages (ICFP)*, 2017.

Conal Elliott. Calculating compilers categorically. Draft, July 2018.

Andrew Farmer, Neil Sculthorpe, and Andy Gill. Reasoning with the Hermit: Tool Support for Equational Reasoning on GHC Core Programs. In *Haskell Symposium*, 2015.

Yoshihiko Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *HOSC*, 12(4), 1999.

Jeremy Gibbons. Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity. *The Art, Science, and Engineering of Programming*, 6(2), 2022.

Martin Handley and Graham Hutton. Improving Haskell. In *Proceedings of the Symposium on Trends in Functional Programming*, March 2018.

John Hannan and Frank Pfenning. Compiler verification in LF. In *LICS*, pages 407–418, 1992.

John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 410–423, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917693. doi: 10.1145/237721.240882. URL https://doi.org/10.1145/237721.240882.

Graham Hutton and Patrick Bahr. Compiling a 50-Year Journey. *Journal of Functional Programming*, 27, September 2017.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *TOPLAS*, 22(3), 2001.

Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

Donald Kaplan. Correctness of a compiler for algol-like programs. Technical Report Artifical Intelligence Memo No. 48, Stanford University, 1967.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The Verified CakeML Compiler Backend. *Journal of Functional Programming*, 29, 2019.

Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7), 2009.

Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3), 2018.

Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 2019.

Simon Marlow. Haskell 2010 Language Report, 2010.

Conor McBride. Ornamental algebras, algebraic ornaments. University of Strathclyde, 2011.

John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Proceedings of Symposia in Applied Mathematics*, 19, 1967.

James McKinna and Joel Wright. A type-correct, stack-safe, provably correct, expression compiler in Epigram. Manuscript, 2006.

Erik Meijer. *Calculating Compilers*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *FPCA*, 1991.

Robin Milner. Logic for computable functions description of a machine implementation. Technical report, Stanford University, 1972a.

Robin Milner. Implementation and applications of scott's logic for computable functions. *ACM sigplan notices*, 7(1):1–6, 1972b.

Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine intelligence*, 7(3):51–70, 1972.

E Moggi. Computational lambda-calculus and monads. In *Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–15. IEEE Computer Society, 1989.

Shin-cheng Mu, Hsiang-shang Ko, and Patrik Jansson. Algebra of Programming in Agda: Dependent Types for Relational Program Derivation. *Journal of Functional Programming*, 19(5), 2009.

George C Necula and Peter Lee. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices*, 33(5):333–344, 1998.

Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

James Painter. Semantic correctness of a compiler for an algol-like language. Technical Report Artifical Intelligence Memo No. 44, Stanford University, 1967.

Alberto Pardo, Emmanuel Gunther, Miguel Pagano, and Marcos Viera. An internalist approach to correct-by-construction compilers. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, 2018.

Mitchell Pickard and Graham Hutton. Calculating Dependently-Typed Compilers. *Proceedings of the ACM on Programming Languages*, 5 (ICFP), August 2021.

Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018.

John Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *ACM Annual Conference*, 1972.

Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. Intrinsically Typed Compilation with Nameless Labels. *Proceedings of the ACM on Programming Languages*, 5(POPL), January 2021.

Mike Spivey. A Functional Theory of Exceptions. *Science of Computer Programming*, 14(1):25–43, 1990.

Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with Type Equality Coercions. In *TLDI*, 2007.

Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 294–305, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931074. URL https://doi.org/10.1145/2931037.2931074.

Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8), aug 1984.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

Philip Wadler. The Concatenate Vanishes. University of Glasgow, 1989.

Philip Wadler. Propositions as types. *Communications of the ACM*, 58 (12):75–84, 2015.

Mitchell Wand. Deriving Target Code as a Representation of Continuation Semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982a.

Mitchell Wand. Semantics-Directed Machine Architecture. In *POPL*, 1982b.

Richard Weyhrauch and Robin Milner. Program semantics and correctness in a mechanized logic. In *Proc. 1st USA-Japan Computer Conf., Tokyo*, 1972.

Part IV

Appendices

Appendix A

Agda code

This appendix includes the code listings for the compilers calculated in this thesis. All code has been tested on Agda version 2.6.2.1.

A.1 Lemmas.agda

```
module Lemmas where


open import Data.Bool hiding (_≤_)
open import Data.Nat
open import Data.Nat.Properties
open import Data.Maybe
open import Relation.Binary.PropositionalEquality


postulate
  funext : ∀ {l₁ l₂} → {A : Set l₁}{B : A → Set l₂}
    → {f g : (x : A) → B x} → ((x : A) → f x ≡ g x) → f ≡ g


+-over-≤ : {m n o : ℕ} → n ≤ o → m + n ≤ m + o
+-over-≤ {zero} {n} {o} x = x
+-over-≤ {(suc n₁)} {n} {o} x = s≤s (+-over-≤ x)
```

```
s≤z-absurd : ∀ {l} {Whatever : Set l}{n : ℕ} → .(suc n ≤ zero) → Whatever
s≤z-absurd ()

m+sn≤i : {m n i : ℕ} → m + suc n ≤ i → suc (m + n) ≤ i
m+sn≤i {m} {n} {i} p rewrite +-suc m n = p

m+s≤z-absurd : ∀ {l} {Whatever : Set l}{m n : ℕ} → .(m + suc n ≤ zero)
  → Whatever
m+s≤z-absurd p = s≤z-absurd (m+sn≤i p)

sabc≡asbc : {a b c : ℕ} → a + suc (b + c) ≡ suc (a + b + c)
sabc≡asbc {zero} {b} {c} = refl
sabc≡asbc {suc a} {b} {c} = cong suc sabc≡asbc

x+sy : (x y z : ℕ) → (suc (suc (x + y)) ≤ suc z) → x + suc y ≤ z
x+sy x y z prf rewrite (+-suc x y) = ≤-pred prf

n+ssm≤si : {n m i : ℕ} → n + suc (suc m) ≤ suc i → n + suc m ≤ i
n+ssm≤si p = ≤-pred (m+sn≤i p)

cong₂-irr : ∀ {a b c} {A : Set a} {B : A → Set b} {C : Set c}
        (f : (a : A) → .(B a) → C) {x y}.{u v} → x ≡ y → f x u ≡ f y v
cong₂-irr f refl = refl

maybeDistributes : ∀ {l₁ l₂ l₃}{A : Set l₁}{B : Set l₂}{C : Set l₃} →
  (f : B → C)(a : A → B)(b : B)(m : Maybe A) →
  f (maybe′ (λ n → a n) b m) ≡ maybe′ (λ n → f (a n)) (f b) m
maybeDistributes f a b (just x) = refl
maybeDistributes f a b nothing = refl

maybeFlatten₃ : {A B C D : Set}
  → (d : D)(x : Maybe A)(y : Maybe B)(f : C → D)(g : A → B → C)
  → maybe′ f d (maybe′ (λ n → maybe′ (λ m → just (g n m)) nothing y) nothing x)
```

```
  ≡
    maybe′ (λ n → maybe′ (λ m → f (g n m)) d y) d x
maybeFlatten₃ d (just x) (just y) f g = refl
maybeFlatten₃ d (just x) nothing f g = refl
maybeFlatten₃ d nothing y f g = refl

maybeFlatten₂ : {A B : Set}
  (f : A → B)(b : B)(x : Maybe A)(y : Maybe A)
  → maybe′ f b (maybe′ just y x)
  ≡
    maybe′ f (maybe′ f b y) x
maybeFlatten₂ f b (just x) y = refl
maybeFlatten₂ f b nothing y = refl

if-distributes : ∀ {l₁ l₂} {A : Set l₁}{B : Set l₂}(f : A → B)
  (b : Bool)(x y : A)
  → f (if b then x else y) ≡ (if b then f x else f y)
if-distributes f false x y = refl
if-distributes f true x y = refl
```

A.2 no-continuations.agda

```
open import Data.Nat
open import Data.Bool hiding (T)
open import Data.List
open import Data.Product
open import Relation.Binary.PropositionalEquality
open ≡-Reasoning


open import Lemmas


variable
  T : Set
  S S' S'' : List Set



------------------------------------------------------------------------
-- Source language
------------------------------------------------------------------------

data Exp : Set → Set where
  val : T → Exp T
  add : Exp ℕ → Exp ℕ → Exp ℕ
  if : Exp Bool → Exp T → Exp T → Exp T

eval : Exp T → T
eval (val x)    = x
eval (add x y)  = eval x + eval y
eval (if b x y) = if eval b then eval x else eval y



------------------------------------------------------------------------
-- Target language
```

```
------------------------------------------------------------------------

data Stack : List Set → Set where
  ε : Stack []
  _▷_ : T → Stack S → Stack (T ∷ S)

infixr 20 _▷_

data Code : List Set → List Set → Set₁ where
  PUSH  : T → Code S (T ∷ S)
  ADD   : Code (ℕ ∷ ℕ ∷ S) (ℕ ∷ S)
  _+++_ : Code S S' → Code S' S'' → Code S S''
  IF    : Code S S' → Code S S' → Code (Bool ∷ S) S'

exec : Code S S' → Stack S → Stack S'
exec (PUSH x) s         = x ▷ s
exec (IF c1 c2) (b ▷ s) = if b then exec c1 s else exec c2 s
exec (c1 +++ c2) s      = exec c2 (exec c1 s)
exec ADD (m ▷ n ▷ s)    = (n + m) ▷ s


------------------------------------------------------------------------
-- Compiler
------------------------------------------------------------------------

compile : Exp T → Code S (T ∷ S)
compile (val x)    = PUSH x
compile (if b x y) = compile b +++ IF (compile x) (compile y)
compile (add x y)  = (compile x +++ compile y) +++ ADD


------------------------------------------------------------------------
-- Calculation
------------------------------------------------------------------------
```

```
correct : (e : Exp T) → (s : Stack S) → eval e ▷ s ≡ exec (compile e) s
correct (val x) s = begin
    eval (val x) ▷ s
  -- definition of eval
  ≡⟨ refl ⟩
    x ▷ s
  -- define exec (PUSH x) s = x ▷ s
  ≡⟨ refl ⟩
    exec (PUSH x) s
  ∎
correct (if b x y) s = begin
    eval (if b x y) ▷ s
  -- definition of eval
  ≡⟨ refl ⟩
    (if eval b then eval x else eval y) ▷ s
  -- distributivity
  ≡⟨ if-distributes (λ z → z ▷ s) (eval b) (eval x) (eval y) ⟩
    (if eval b then eval x ▷ s else eval y ▷ s)
  -- induction hypothesis for x
  ≡⟨ cong (λ z → if eval b then z else eval y ▷ s) (correct x s) ⟩
    (if eval b then exec (compile x) s else eval y ▷ s)
  -- induction hypothesis for y
  ≡⟨ cong (λ z → if eval b then exec (compile x) s else z) (correct y s) ⟩
    (if eval b then exec (compile x) s else exec (compile y) s)
  -- define: exec (IF c1 c2) (b ▷ s) = if b then exec c1 s else exec c2 s
  ≡⟨ refl ⟩
    exec (IF (compile x) (compile y)) (eval b ▷ s)
  -- define: exec (c1 +++ c2) = exec c2 (exec c1 s)
  ≡⟨ cong (exec (IF (compile x) (compile y))) (correct b s) ⟩
    exec (IF (compile x) (compile y)) (exec (compile b) s)
  -- define: exec (c1 +++ c2) = exec c2 (exec c1 s)
  ≡⟨ refl ⟩
    exec (compile b +++ IF (compile x) (compile y)) s
```

```
  ∎
correct (add x y) s = begin
    eval (add x y) ▷ s
  -- definition of eval
  ≡⟨ refl ⟩
    (eval x + eval y) ▷ s
  -- define exec ADD (m ▷ n ▷ s) = (n + m) ▷ s
  ≡⟨ refl ⟩
    exec ADD (eval y ▷ eval x ▷ s)
  -- induction hypothesis for x
  ≡⟨ cong (λ z → exec ADD (eval y ▷ z)) (correct x s) ⟩
    exec ADD (eval y ▷ exec (compile x) s)
  -- induction hypothesis for y
  ≡⟨ cong (exec ADD) (correct y (exec (compile x) s)) ⟩
    exec ADD (exec (compile y) (exec (compile x) s))
  -- definition of +++
  ≡⟨ refl ⟩
    exec ADD (exec (compile x +++ compile y) s)
  -- definition of +++
  ≡⟨ refl ⟩
    exec ((compile x +++ compile y) +++ ADD) s
  ∎
```

A.3 continuations.agda

```
open import Data.Nat
open import Data.Bool hiding (T)
open import Data.List
open import Data.Product
open import Relation.Binary.PropositionalEquality
open ≡-Reasoning


open import Lemmas


variable
  T : Set
  S S' S'' : List Set



------------------------------------------------------------------------
-- Source language
------------------------------------------------------------------------

data Exp : Set → Set where
  val : T → Exp T
  add : Exp ℕ → Exp ℕ → Exp ℕ
  if : Exp Bool → Exp T → Exp T → Exp T

eval : Exp T → T
eval (val x)    = x
eval (add x y)  = eval x + eval y
eval (if b x y) = if eval b then eval x else eval y



------------------------------------------------------------------------
-- Target language
```

```
------------------------------------------------------------------------

data Stack : List Set → Set where
  ε : Stack []
  _▷_ : T → Stack S → Stack (T ∷ S)

infixr 20 _▷_

data Code : List Set → List Set → Set₁ where
  PUSH : T → Code (T ∷ S) S' → Code S S'
  ADD : Code (ℕ ∷ S) S' → Code (ℕ ∷ ℕ ∷ S) S'
  IF : Code S S' → Code S S' → Code (Bool ∷ S) S'
  HALT : Code S S

exec : Code S S' → Stack S → Stack S'
exec (PUSH x c) s        = exec c (x ▷ s)
exec (IF c1 c2) (b ▷ s)  = if b then exec c1 s else exec c2 s
exec (ADD c) (m ▷ n ▷ s) = exec c ((n + m) ▷ s)
exec HALT s              = s



------------------------------------------------------------------------
-- Compiler
------------------------------------------------------------------------

comp : Exp T → Code (T ∷ S) S' → Code S S'
comp (val x) c    = PUSH x c
comp (if b x y) c = comp b (IF (comp x c) (comp y c))
comp (add x y) c  = comp x (comp y (ADD c))

compile : Exp T → Code S (T ∷ S)
compile e = comp e HALT
```

```
------------------------------------------------------------------------
-- Calculation
------------------------------------------------------------------------


correct : (c : Code (T ∷ S) S') → (e : Exp T) → (s : Stack S)
  → exec c (eval e ▷ s) ≡ exec (comp e c) s
correct c (val x) s = begin
    exec c (eval (val x) ▷ s)
  -- definition of eval
  ≡⟨ refl ⟩
    exec c (x ▷ s)
  -- define exec (PUSH x c) s = exec c (x ▷ s)
  ≡⟨ refl ⟩
    exec (PUSH x c) s
  ∎
correct c (if b x y) s = begin
    exec c (eval (if b x y) ▷ s)
  -- definition of eval
  ≡⟨ refl ⟩
    exec c ((if eval b then eval x else eval y) ▷ s)
  -- distributivity
  ≡⟨ cong (exec c) (if-distributes (λ z → z ▷ s) (eval b) (eval x) (eval y)) ⟩
    exec c (if eval b then eval x ▷ s else eval y ▷ s)
  -- distributivity
  ≡⟨ if-distributes (exec c) (eval b) (eval x ▷ s) (eval y ▷ s) ⟩
    (if eval b then exec c (eval x ▷ s) else exec c (eval y ▷ s))
  -- induction hypothesis for x
  ≡⟨ cong (λ z → if eval b then z else exec c (eval y ▷ s)) (correct c x s) ⟩
    (if eval b then exec (comp x c) s else exec c (eval y ▷ s))
  -- induction hypothesis for y
  ≡⟨ cong (λ z → if eval b then exec (comp x c) s else z) (correct c y s) ⟩
    (if eval b then exec (comp x c) s else exec (comp y c) s)
  --  define: exec (IF c1 c2) (b ▷ s) = if b then exec c1 s else exec c2 s
  ≡⟨ refl ⟩
```

```
    exec (IF (comp x c) (comp y c)) (eval b ▷ s)
  -- induction hypothesis for b
  ≡⟨ correct ((IF (comp x c) (comp y c))) b s ⟩
    exec (comp b (IF (comp x c) (comp y c))) s
  ∎
correct c (add x y) s = begin
    exec c (eval (add x y) ▷ s)
  -- definition of eval
  ≡⟨ refl ⟩
    exec c ((eval x + eval y) ▷ s)
  -- define: exec (ADD c) (m ▷ n ▷ s) = exec c ((n + m) ▷ s)
  ≡⟨ refl ⟩
    exec (ADD c) (eval y ▷ eval x ▷ s)
  -- induction hypothesis for y
  ≡⟨ correct (ADD c) y (eval x ▷ s) ⟩
    exec (comp y (ADD c)) (eval x ▷ s)
  -- induction hypothesis for x
  ≡⟨ correct (comp y (ADD c)) x s ⟩
    exec (comp x (comp y (ADD c))) s
  ∎


compile-correct : (e : Exp T) → (s : Stack S)
  → eval e ▷ s ≡ exec (compile e) s
compile-correct e s = begin
    eval e ▷ s
  ≡⟨ refl ⟩
    exec HALT (eval e ▷ s)
  ≡⟨ correct HALT e s ⟩
    exec (comp e HALT) s
  ∎
```

A.4 exceptions.agda

```
open import Data.Bool hiding (T; _≤_)
open import Data.Nat
open import Data.Nat.Properties
open import Data.List
open import Data.Maybe
open import Data.Product
open import Relation.Binary.PropositionalEquality
open ≡-Reasoning

open import Lemmas

------------------------------------------------------------------------
-- Source language
------------------------------------------------------------------------

data Ty : Set where
  nat : Ty
  han : List Ty → List Ty → Ty

variable
  T : Ty
  a b : Bool
  S S' S'' : List Ty

data Exp : Bool → Set where
  val : ℕ → Exp false
  add : Exp a → Exp b → Exp (a ∨ b)
  throw : Exp true
  catch : Exp a → Exp b → Exp (a ∧ b)

-- Evaluation semantics where expression can throw an uncaught exception
```

```
eval? : Exp b → Maybe ℕ
eval? (val n) = just n
eval? (add x y) = maybe′ (λ n → (maybe′ (λ m → just (n + m))
                    nothing (eval? y))) nothing (eval? x)
eval? throw = nothing
eval? (catch x h) = maybe′ (λ n → just n) (eval? h) (eval? x)

-- Evaluation semantics where expression cannot throw an uncaught exception
eval : b ≡ false → Exp b → ℕ
eval p (val n) = n
eval p (add {false} {false} x y)  = eval refl x + eval refl y
eval p (catch {false} {a} x h) = eval refl x
eval p (catch {true} {false} x h) = maybe′ (λ n → n) (eval refl h) (eval? x)

-- Proof that, in the case where e cannot throw, eval and eval? are equivalent
eval-correct : (p : b ≡ false) → (e : Exp b) → just (eval p e) ≡ eval? e
eval-correct p (val n) = refl
eval-correct p (add {false} {false} x y) rewrite sym (eval-correct refl x)
  | sym (eval-correct refl y) = refl
eval-correct p (catch {false} {a} x h) rewrite sym (eval-correct refl x) = refl
eval-correct p (catch {true} {false} x h) with eval? x
eval-correct p (catch {true} {false} x h) | just n  = refl
eval-correct p (catch {true} {false} x h) | nothing = eval-correct refl h



------------------------------------------------------------------------
-- Target language
------------------------------------------------------------------------

data Code : List Ty → List Ty → Set where
  PUSH : ℕ → Code (nat ∷ S) S' → Code S S'
  ADD : Code (nat ∷ S) S'  → Code (nat ∷ nat ∷ S) S'
  THROW : Code (S'' ++ han S S' ∷ S) S'
  MARK : (h : Code S S') → (c : Code (han S S' ∷ S) S') → Code S S'
```

```
  UNMARK : Code (T :: S) S'  → Code (T :: han S S' :: S) S'
  HALT : Code S S

El : Ty → Set
El nat = ℕ
El (han S S') = Code S S'

data Stack : List Ty → Set where
  ε : Stack []
  _▷_ : El T → Stack S → Stack (T :: S)

infixr 20 _▷_

-- The definitions for exec and fail calculated in the paper do not
-- pass the totality checker, as they are not structurally
-- recursive. We fix this problem by passing around an extra arguemnt,
-- representing the size of terms, and a proof that this size is
-- greater-than or equal to the actual size of the term. This is not a
-- strict equality relationship, because it decreases at different
-- rates depending on the code constructor being evaluated.

-- The sizeX functions calculate the sizes of code/values/stacks.
sizeCode : Code S S' → ℕ
sizeCode (PUSH x c) = suc (suc (sizeCode c))
sizeCode (ADD c) = sizeCode c
sizeCode THROW = zero
sizeCode (MARK h c) = suc (suc (sizeCode c + sizeCode h))
sizeCode (UNMARK c) = suc (sizeCode c)
sizeCode HALT = zero

sizeEl : El T → ℕ
sizeEl {nat} x = 0
sizeEl {han S S'} x = sizeCode x
```

```
sizeSta : Stack S → ℕ
sizeSta ε = suc zero
sizeSta (x ▷ s) = suc (sizeEl x + sizeSta s)

-- Running UNMARK on a stack does not make it bigger
unmark-smaller : {s : Stack S}(x : El T)(h : Code S S')
  → sizeSta (x ▷ s) ≤ sizeSta (x ▷ h ▷ s)
unmark-smaller {T = nat} x h = s≤s (≤-step (n≤m+n _ _))
unmark-smaller {T = han _ _} x h = s≤s (+-over-≤ (≤-step (n≤m+n _ _)))

-- exec' and fail' correspond to the definition of exec and fail in
-- the paper, but with additional arguments to aid totality checking.
-- The proof argument is marked as irrelevant, so its actual value
-- doesn't matter; it just has to exist.
exec' : (c : Code S S') → (s : Stack S)
  → (size : ℕ) → .(p : size ≥ sizeCode c + sizeSta s)
  → Stack S'

fail' : (s : Stack (S'' ++ han S S' ∷ S))
  → (size : ℕ) → .(p : size ≥ sizeSta s)
  → Stack S'
fail' {[]} (h' ▷ s) (suc size) p
  = exec' h' s size (≤-pred p)
fail' {_ ∷ _} (n ▷ s) (suc size) p
  = fail' s size (≤-trans (n≤m+n (sizeEl n) (sizeSta s)) (≤-pred p))

exec' (PUSH n c) s (suc size) p
  = exec' c (n ▷ s) size (≤-pred (≤-trans (s≤s
      (≤-reflexive (+-suc (sizeCode c) (sizeSta s)))) p))
exec' (ADD c) (m ▷ n ▷ s) (suc size) p
  = exec' c ((n + m) ▷ s) size (≤-pred (≤-trans (≤-reflexive
      (sym (+-suc (sizeCode c) (suc (sizeSta s))))) p))
exec' THROW s size p
  = fail' s size p
```

```
exec' (MARK h c) s (suc size) p
  = exec' c (h ▷ s)   size (≤-pred (≤-trans (s≤s (≤-reflexive sabc≡asbc)) p))
exec' (UNMARK c) (x ▷ h ▷ s) (suc size) p
  = exec' c (x ▷ s) size (≤-trans (+-over-≤ (unmark-smaller x h)) (≤-pred p))
exec' HALT s size p
  = s
exec' (ADD c) (m ▷ n ▷ s)   zero p
  rewrite (+-suc (sizeCode c) (suc (sizeSta s))) = s≤z-absurd p

-- exec and fail simply call exec' and fail' with the appropriate size
-- argument and proof
exec : (c : Code S S') → (s : Stack S) → Stack S'
exec c s = exec' c s (sizeCode c + sizeSta s) ≤-refl

fail : Stack (S'' ++ han S S' :: S) → Stack S'
fail s = fail' s (sizeSta s) ≤-refl



------------------------------------------------------------------------
-- Proofs of defining equations for exec and fail
------------------------------------------------------------------------

-- Due to the size arguemnt to exec', the definitions from the paper
-- do not automatically reduce. We have proven each of the defining
-- equations here, which are used in the calculation later.

execPUSH : ∀ n → (c : Code (nat :: S) S') → (s : Stack S)
  → exec (PUSH n c) s ≡ exec c (n ▷ s)
execPUSH n c s =
 begin
    exec' (PUSH n c) s (sizeCode (PUSH n c) + sizeSta s) (≤-refl)
  ≡⟨ cong₂-irr (exec' c (n ▷ s)) ((sym (+-suc (sizeCode c) (sizeSta s)))) ⟩
    exec' (PUSH n c) s (suc (sizeCode c + sizeSta (n ▷ s)))
      (s≤s (≤-reflexive (sym (+-suc (sizeCode c) (sizeSta s)))))
```

```
  ≡⟨ cong₂-irr (exec' c (n ▷ s)) (+-suc (sizeCode c) (sizeSta s)) ⟩
    exec' (PUSH n c) s (suc (suc (sizeCode c + sizeSta s)))
      (s≤s (s≤s ≤-refl))
  ≡⟨ cong₂-irr (exec' c (n ▷ s)) (sym (+-suc (sizeCode c) (sizeSta s))) ⟩
    exec' c (n ▷ s) (sizeCode c + sizeSta (n ▷ s)) ≤-refl
  ∎

execADD : ∀ n m → (c : Code (nat ∷ S) S') → (s : Stack S)
  → exec (ADD c) (m ▷ n ▷ s) ≡ exec c ((n + m) ▷ s)
execADD n m c s =
  begin
    exec (ADD c) (m ▷ n ▷ s)
  ≡⟨ refl ⟩
    exec' (ADD c) (m ▷ n ▷ s) (sizeCode (ADD c) + sizeSta (m ▷ n ▷ s)) ≤-refl
  ≡⟨ cong₂-irr (exec' (ADD c) (m ▷ n ▷ s))
       {v = ≤-reflexive (+-suc (sizeCode c) (suc (sizeSta s)))}
    (+-suc (sizeCode c) (suc (sizeSta s))) ⟩
    exec' (ADD c) (m ▷ n ▷ s) (suc (sizeCode c + (suc (sizeSta s))))
      (≤-reflexive (+-suc (sizeCode c) (suc (sizeSta s))))
  ≡⟨ cong₂-irr (exec' c ((n + m) ▷ s)) refl ⟩
    exec' c ((n + m) ▷ s) (sizeCode c + suc (sizeSta s)) ≤-refl
  ≡⟨ refl ⟩
    exec c ((n + m) ▷ s)
  ∎

execTHROW : (s : Stack (S'' ++ han S S' ∷ S))
  → exec THROW s ≡ fail s
execTHROW s = refl

-- Proofs that the size argument to exec and fail do not matter; the
-- proof arguemnt to these functions ensures that the values are
-- always correct
exec-size-irr : (size₁ size₂ : ℕ) → (c : Code S S') → (s : Stack S)
  → (p₁ : _)(p₂ : _) → exec' c s size₁ p₁ ≡ exec' c s size₂ p₂
```

```
fail-size-irr : (size₁ size₂ : ℕ) → (s : Stack (S'' ++ han S S' ∷ S))
  → (p₁ : _)(p₂ : _) → fail' s size₁ p₁ ≡ fail' s size₂ p₂
fail-size-irr {[]} zero size₂ (x ▷ s) () p₂
fail-size-irr {_ ∷ _} zero size₂ (x ▷ s) () p₂
fail-size-irr {[]} (suc size₁) zero (x ▷ s) p₁ ()
fail-size-irr {_ ∷ _} (suc size₁) zero (x ▷ s) p₁ ()
fail-size-irr {[]} (suc size₁) (suc size₂) (h ▷ s) p₁ p₂
  = exec-size-irr size₁ size₂ h s (≤-pred p₁) (≤-pred p₂)
fail-size-irr {_ ∷ _} (suc size₁) (suc size₂) (x ▷ s) p₁ p₂
  = fail-size-irr size₁ size₂ s
  (≤-pred (≤-trans (s≤s (n≤m+n (sizeEl x) (sizeSta s))) p₁))
  (≤-pred (≤-trans (s≤s (n≤m+n (sizeEl x) (sizeSta s))) p₂))

exec-size-irr (suc size₁) (suc size₂) (PUSH x c) s p₁ p₂
  = exec-size-irr size₁ size₂ c (x ▷ s)
  (x+sy (sizeCode c) (sizeSta s) size₁ p₁)
  (x+sy (sizeCode c) (sizeSta s) size₂ p₂)
exec-size-irr zero zero (ADD c) s p₁ p₂ = refl
exec-size-irr zero (suc size₂) (ADD c) (x ▷ s) p₁ p₂ = m+s≤z-absurd p₁
exec-size-irr (suc size₁) zero (ADD c) (x ▷ s) p₁ p₂ = m+s≤z-absurd p₂
exec-size-irr (suc size₁) (suc size₂) (ADD c) (m ▷ n ▷ s) p₁ p₂
  = exec-size-irr size₁ size₂ c ((n + m) ▷ s)
  (n+ssm≤si p₁) (n+ssm≤si p₂)
exec-size-irr zero zero THROW s p₁ p₂ = refl
exec-size-irr zero (suc size₂) (THROW {[]}) (x ▷ s) () p₂
exec-size-irr zero (suc size₂) (THROW {_ ∷ _}) (x ▷ s) () p₂
exec-size-irr (suc size₁) zero (THROW {[]}) (x ▷ s) p₁ ()
exec-size-irr (suc size₁) zero (THROW {_ ∷ _}) (x ▷ s) p₁ ()
exec-size-irr (suc size₁) (suc size₂) THROW s p₁ p₂
  = fail-size-irr (suc size₁) (suc size₂) s p₁ p₂
exec-size-irr (suc size₁) (suc size₂) (MARK h c) s p₁ p₂
  = exec-size-irr size₁ size₂ c (h ▷ s)
  (≤-pred (≤-trans (s≤s (≤-reflexive sabc≡asbc)) p₁))
```

```
  (≤-pred (≤-trans (s≤s (≤-reflexive sabc≡asbc)) p₂))
exec-size-irr (suc size₁) (suc size₂) (UNMARK c) (n ▷ h ▷ s) p₁ p₂
  = exec-size-irr size₁ size₂ c (n ▷ s)
  (≤-pred (≤-trans (s≤s (+-over-≤ (s≤s (+-over-≤ (≤-step
    (n≤m+n (sizeCode h) (sizeSta s)))))) p₁))
  (≤-pred (≤-trans (s≤s (+-over-≤ (s≤s (+-over-≤ (≤-step
    (n≤m+n (sizeCode h) (sizeSta s)))))) p₂))
exec-size-irr size₁ size₂ HALT s p₁ p₂ = refl

execUNMARK : (c : Code (nat ∷ S) S')(h : Code S S')(n : ℕ) → (s : Stack S)
  → exec (UNMARK c) (n ▷ h ▷ s) ≡ exec c (n ▷ s)
execUNMARK c h n s = begin
    exec (UNMARK c) (n ▷ h ▷ s)
  ≡⟨ refl ⟩
    exec' (UNMARK c) (n ▷ h ▷ s) (suc (sizeCode c)
      + suc (suc (sizeCode h + sizeSta s))) ≤-refl
  ≡⟨ exec-size-irr (sizeCode c + suc (suc (sizeCode h + sizeSta s)))
       (sizeCode c + suc (sizeSta s)) c (n ▷ s)
     (+-over-≤ (s≤s (n≤m+n (suc (sizeCode h)) (sizeSta s)))) ≤-refl ⟩
    exec' c (n ▷ s) (sizeCode c + suc (sizeSta s)) ≤-refl
  ≡⟨ refl ⟩
    exec c (n ▷ s)
  ∎

execMARK : (c : Code (han S S' ∷ S) S')(h : Code S S') → (s : Stack S)
  → exec (MARK h c) s ≡ exec c (h ▷ s)
execMARK c h s = begin
    exec (MARK h c) s
  ≡⟨ refl ⟩
    exec' (MARK h c) s (suc (suc (sizeCode c + sizeCode h + sizeSta s))) ≤-refl
  ≡⟨ cong₂-irr (exec' c (h ▷ s)) (sym sabc≡asbc) ⟩
    exec' c (h ▷ s) (sizeCode c + suc (sizeCode h + sizeSta s)) ≤-refl
  ≡⟨ refl ⟩
    exec c (h ▷ s)
```

```
  ∎

failHan : (s : Stack S)(h : Code S S')
  → fail {S'' = []} (h ▷ s) ≡ exec h s
failHan s h = refl



------------------------------------------------------------------------
-- Compilers
------------------------------------------------------------------------

comp? : Exp b → Code (nat ∷ (S'' ++ han S S' ∷ S)) S'
  → Code (S'' ++ han S S' ∷ S) S'
comp? (val n) c = PUSH n c
comp? (add x y) c = comp? x (comp? {S'' = nat ∷ _} y (ADD c))
comp? throw c = THROW
comp? (catch x h) c = MARK (comp? h c) (comp? {S'' = []} x (UNMARK c))

comp : b ≡ false → Exp b → Code (nat ∷ S) S' → Code S S'
comp p (val x) c = PUSH x c
comp p (add {false} {false} x y) c = comp refl x (comp refl y (ADD c))
comp p (catch {false} {_} x h) c = comp refl x c
comp p (catch {true} {false} x h) c
  = MARK (comp refl h c) (comp? {S'' = []} x (UNMARK c))

compile : b ≡ false → Exp b → Code S (nat ∷ S)
compile p e = comp p e HALT



------------------------------------------------------------------------
-- Calculations
------------------------------------------------------------------------

correct? :
```

```
  (e : Exp b)
  → (s : Stack (S'' ++ han S S' ∷ S))
  → (c : Code ((nat ∷ S'') ++ han S S' ∷ S) S')
  → maybe′ (λ n → exec c (n ▷ s)) (fail s) (eval? e) ≡ exec (comp? e c) s
correct? (val n) s c = begin
    maybe′ (λ n' → exec c (n' ▷ s)) (fail s) (eval? (val n))
  -- definition of eval
  ≡⟨ refl ⟩
    exec c (n ▷ s)
  -- definition of exec
  ≡⟨ sym (execPUSH n c s) ⟩
    exec (PUSH n c) s
  ∎
correct? throw s c = begin
    maybe′ (λ n → exec c (n ▷ s)) (fail s) (eval? throw)
  -- definition of eval
  ≡⟨ refl ⟩
    fail s
  -- define: exec THROW s = fail s
  ≡⟨ sym (execTHROW s) ⟩
    exec THROW s
  ∎
correct? {S'' = S''} (add x y) s c = begin
    maybe′ (λ n → exec c (n ▷ s)) (fail s) (eval? (add x y))
  -- definition of eval
  ≡⟨ maybeFlatten₃ (fail s) (eval? x) (eval? y) (λ n → exec c (n ▷ s)) _+_ ⟩
    maybe′ (λ n → maybe′ (λ m → exec c ((n + m) ▷ s)) (fail s) (eval? y))
      (fail s) (eval? x)
  -- definition of exec
  ≡⟨ cong (λ z → maybe′ z (fail s) (eval? x)) (funext λ n → cong
       (λ z → maybe′ z (fail s) (eval? y))
       (funext (λ m → sym (execADD n m c s)))) ⟩
    maybe′ (λ n → maybe′ (λ m → exec (ADD c) (m ▷ n ▷ s)) (fail s) (eval? y))
      (fail s) (eval? x)
```

```
  -- define: fail (n ▷ s) = fail s
  ≡⟨ cong (λ z → maybe′ (λ n → maybe′ (λ m → exec (ADD c) (m ▷ n ▷ s)) z
       (eval? y)) (fail s) (eval? x)) refl ⟩
    maybe′ (λ n → maybe′ (λ m → exec (ADD c) (m ▷ n ▷ s))
      (fail {S'' = nat ∷ _} (n ▷ s)) (eval? y)) (fail s) (eval? x)
  -- induction hypothesis for y
  ≡⟨ cong (λ z → maybe′ z (fail s) (eval? x))
       (funext (λ n → correct? {S'' = nat ∷ S''} y (n ▷ s) (ADD c))) ⟩
    maybe′ (λ n → exec (comp? y (ADD c)) (n ▷ s)) (fail s) (eval? x)
  -- induction hypothesis for x
  ≡⟨ correct? x s (comp? {S'' = nat ∷ S''} y (ADD c)) ⟩
    exec (comp? x (comp? {S'' = nat ∷ S''} y (ADD c))) s
  ∎
correct? {S'' = S''} (catch x h) s c = begin
    maybe′ (λ n → exec c (n ▷ s)) (fail s) (eval? (catch x h))
  -- definition of eval
  ≡⟨ maybeFlatten₂ (λ n → exec c (n ▷ s)) (fail s) (eval? x) (eval? h) ⟩
    maybe′ (λ n → exec c (n ▷ s)) (maybe′ (λ n → exec c (n ▷ s))
      (fail s) (eval? h)) (eval? x)
  -- induction hypothesis for h
  ≡⟨ cong (λ z → maybe′ (λ n → exec c (n ▷ s)) z (eval? x)) (correct? h s c) ⟩
    maybe′ (λ n → exec c (n ▷ s)) (exec (comp? h c) s) (eval? x)
  -- define: fail {S'' = han _ _ ∷ _} (h ▷ s) = exec h s
  ≡⟨ refl ⟩
    maybe′ (λ n → exec c (n ▷ s)) (fail {S'' = []}
      (comp? {S'' = S''} h c ▷ s)) (eval? x)
  -- definition of exec
  ≡⟨ cong (λ z → maybe′ z (fail {[]} (comp? {S'' = S''} h c ▷ s)) (eval? x))
       (funext (λ n → sym (execUNMARK c (comp? {S'' = S''} h c) n s))) ⟩
    maybe′ (λ n → exec (UNMARK c) (n ▷ comp? {S'' = S''} h c ▷ s))
      (fail {S'' = []} (comp? {S'' = S''} h c ▷ s)) (eval? x)
  -- induction hypothesis for x
  ≡⟨ correct? x (comp? h c ▷ s) (UNMARK c) ⟩
    exec (comp? {S'' = []} x (UNMARK c)) (comp? {S'' = S''} h c ▷ s)
```

```
  -- definition of exec
  ≡⟨ sym (execMARK (comp? {S'' = []} x (UNMARK c)) (comp? h c) s) ⟩
    exec (MARK (comp? {S'' = S''} h c) (comp? {S'' = []} x (UNMARK c))) s
  ∎

correct : (p : b ≡ false)(e : Exp b)(s : Stack S)(c : Code (nat ∷ S) S')
  → exec c ((eval p e) ▷ s) ≡ exec (comp p e c) s
correct p (val n) s c = begin
    exec c (eval p (val n) ▷ s)
  -- definition of eval
  ≡⟨ refl ⟩
    exec c (n ▷ s)
  -- define: exec (PUSH n c) s = exec c (n ▷ s)
  ≡⟨ sym (execPUSH n c s) ⟩
    exec (PUSH n c) s
  ∎
correct p (add {false} {false} x y) s c = begin
    exec c (eval p (add x y) ▷ s)
  -- definition of eval
  ≡⟨ refl ⟩
    exec c ((eval refl x + eval refl y) ▷ s)
  -- define: exec (ADD c) (m ▷ n ▷ s) = exec c (n + m ▷ s)
  ≡⟨ sym (execADD (eval refl x) (eval refl y) c s) ⟩
    exec (ADD c) (eval refl y ▷ eval refl x ▷ s)
  -- inductive hypothesis for y
  ≡⟨ correct refl y (eval refl x ▷ s) (ADD c) ⟩
    exec (comp refl y (ADD c)) (eval refl x ▷ s)
  -- inductive hypothesis for x
  ≡⟨ correct refl x s (comp refl y (ADD c)) ⟩
    exec (comp refl x (comp refl y (ADD c))) s
  ∎
correct p (catch {false} {_} x h) s c = begin
    exec c (eval p (catch x h) ▷ s)
  -- aplying eval
```

```
  ≡⟨ refl ⟩
    exec c (eval refl x ▷ s)
  -- inductive hypothesis for x
  ≡⟨ correct refl x s c ⟩
    exec (comp refl x c) s
  ∎
correct p (catch {true} {false} x h) s c = begin
    exec c (eval p (catch x h) ▷ s)
  -- definition of eval
  ≡⟨ refl ⟩
    exec c (maybe′ (λ n → n) (eval refl h) (eval? x) ▷ s)
  -- distributivity of case
  ≡⟨ maybeDistributes (λ n → exec c (n ▷ s)) (λ n → n)
       (eval refl h) (eval? x) ⟩
    maybe′ (λ n → exec c (n ▷ s)) (exec c (eval refl h ▷ s)) (eval? x)
  -- inductive hypothesis for h
  ≡⟨ cong (λ z → maybe′ (λ n → exec c (n ▷ s)) z (eval? x))
       (correct refl h s c) ⟩
    maybe′ (λ n → exec c (n ▷ s)) (exec (comp refl h c) s) (eval? x)
  -- define: fail {S'' = han _ _ ∷ _} (h ▷ s) = exec h s
  ≡⟨ cong (λ z → maybe′ (λ n → exec c (n ▷ s)) z (eval? x))
       (sym (failHan s (comp refl h c))) ⟩
    maybe′ (λ n → exec c (n ▷ s)) (fail {[]} (comp refl h c ▷ s)) (eval? x)
  -- define: exec (UNMARK c) (x ▷ h ▷ s) = exec c (x ▷ s)
  ≡⟨ cong (λ z → maybe′ z (fail {[]} (comp refl h c ▷ s)) (eval? x))
       (funext λ n → sym (execUNMARK c (comp refl h c) n s)) ⟩
    maybe′ (λ n → exec (UNMARK c) (n ▷ comp refl h c ▷ s))
       (fail {[]} (comp refl h c ▷ s)) (eval? x)
  -- specification (4) for x
  ≡⟨ correct? {S'' = []} x (comp refl h c ▷ s) (UNMARK c) ⟩
    exec (comp? x (UNMARK c)) (comp refl h c ▷ s)
  -- define: exec (MARK h c) s = exec c (h ▷ s)
  ≡⟨ sym (execMARK (comp? {S'' = []} x (UNMARK c)) (comp refl h c) s) ⟩
    exec (MARK (comp refl h c) (comp? {S'' = []} x (UNMARK c))) s
```

```
  ∎


compile-correct : (p : b ≡ false) → (e : Exp b) → (s : Stack S)
  → eval p e ▷ s ≡ exec (compile p e) s
compile-correct p e s = begin
    eval p e ▷ s
  ≡⟨ refl ⟩
    exec HALT (eval p e ▷ s)
  ≡⟨ correct p e s HALT ⟩
    exec (comp p e HALT) s
  ∎
```

A.5 stlc.agda

```
open import Data.Empty
open import Data.Fin hiding (lift)
open import Data.List hiding (lookup)
open import Data.Nat
open import Data.Product
open import Data.Vec
open import Relation.Binary.PropositionalEquality



------------------------------------------------------------------------
-- Types and variables
------------------------------------------------------------------------

-- whether a value is used in the setting of expressions or compiled code
data Setting : Set where
  E : Setting
  C : Setting

variable
  n : ℕ
  s : Setting
  T : Set

mutual
  StackTy : Set₁
  StackTy = List ExpTy

  EnvT : (n : ℕ) → Set₁
  EnvT n = Vec ExpTy n

  data ExpTy : Set₁ where
```

```
    ty : (T : Set) → ExpTy
    fn : (a : ExpTy) → (b : ExpTy) → ExpTy
    cloT : EnvT n → StackTy → StackTy → ExpTy

data Lift {l} : Set → Set l where
  lift : T → Lift T

variable
  A B : ExpTy
  S S' S'' S0 S1 S2 : StackTy
  envT envT' envT'' : EnvT n



------------------------------------------------------------------------
-- Source language
------------------------------------------------------------------------

data STLC : EnvT n → ExpTy → Set₁ where
  val : T → STLC envT (ty T)
  var : (i : Fin n) → STLC envT (lookup envT i)
  lam : (STLC (A Vec.∷ envT) B) → STLC envT (fn A B)
  app : STLC envT (fn A B) → STLC envT A → STLC envT B




------------------------------------------------------------------------
-- Target language
------------------------------------------------------------------------

data Code : EnvT n → StackTy → StackTy → Set₁ where
  HALT : Code envT S S
  LOOKUP : (i : Fin n) → Code envT (lookup envT i ∷ S) S' → Code envT S S'
  PUSH : (x : T) → Code envT (ty T ∷ S) S' → Code envT S S'
  ABS : ({S S' : StackTy}{n : ℕ}{envT' : EnvT n}
```

```
    → Code (A Vec.∷ envT) (cloT envT' (B ∷ S) S' ∷ S) S')
    → Code envT (fn A B ∷ S) S'
    → Code envT S S'
  APPLY : Code envT (B ∷ S) S' → Code envT (A ∷ fn A B ∷ S) S'
  RET : {envT' : EnvT n} → Code envT (A ∷ cloT envT' (A ∷ S) S' ∷ S) S'


------------------------------------------------------------------------
-- Compiler
------------------------------------------------------------------------

comp : STLC envT A → Code envT (A ∷ S) S' → Code envT S S'
comp (val x) c = PUSH x c
comp (var i) c = LOOKUP i c
comp {S = S} {S' = S'} (lam e) c = ABS (comp e RET) c
comp (app x y) c = comp x (comp y (APPLY c))


------------------------------------------------------------------------
-- Semantics
------------------------------------------------------------------------

mutual
  Val : Setting → ExpTy → Set₁
  Val s (ty T) = Lift T
  Val E (fn a b) =
    Σ ℕ (λ n → Σ (EnvT n) (λ envT → Env E envT × STLC (a Vec.∷ envT) b))
  Val C (fn a b) = {S S' : StackTy}{n : ℕ}{envT' : EnvT n} →
    Σ ℕ (λ n → Σ (EnvT n) (λ envT → Env C envT
        × Code (a Vec.∷ envT) (cloT envT' (b ∷ S) S' ∷ S) S'))
  Val E (cloT envT S S') = Lift ⊥
  Val C (cloT envT S S') = Clo envT S S'

  data Env : Setting → EnvT n → Set₁ where
```

```
    εₛ : Env s Vec.[]
    _▷ₛ_ : {A : ExpTy} → (a : Val s A) → Env s envT → Env s (A Vec.∷ envT)

  data Clo : EnvT n → StackTy → StackTy → Set₁ where
    clo : Env C envT → Code envT S S' → Clo envT S S'

variable
  env : Env C envT

get : (i : Fin n) → Env s envT → Val s (lookup envT i)
get zero (x ▷ₛ s) = x
get (suc i) (x ▷ₛ s) = get i s

{-# TERMINATING #-}
eval : STLC envT A → Env E envT → Val E A
eval (val x) env = lift x
eval (var i) env = get i env
eval {n}{envT} (lam e) env = n , envT , env , e
eval (app x y) env with eval x env
eval (app x y) env | (n , envT' , env' , e) = eval e (eval y env ▷ₛ env')



-- evaluation semantics for the simply-typed lambda calculus
data _⇓[_]_ : STLC envT A → Env E envT → Val E A → Set₁ where
  evalVal : {x : T}{env : Env E envT}
    → val x ⇓[ env ] lift x
  evalVar : {i : Fin n}{env : Env E envT}
    → var i ⇓[ env ] get i env
  evalLam : {n : ℕ}{envT : EnvT n}{x : STLC (A Data.Vec.∷ envT) B}{env : _}
    → lam x ⇓[ env ] (n , envT , env , x)
  evalApp : {env : Env E envT}{env' : Env E envT'}
    {x : STLC envT (fn A B)}{y : STLC envT A}{y' : _}
    {e : STLC (A Vec.∷ envT') B}{xy : _}
    → x ⇓[ env ] (n , envT' , env' , e)
```

```
    → y ⇓[ env ] y'
    → e ⇓[ y' ▷ₛ env' ] xy
    → app x y ⇓[ env ] xy

data Stack : StackTy → Set₁ where
  ε : Stack []
  _▷_ : Val C A → Stack S → Stack (A ∷ S)

variable
  sta : Stack S

mutual
  valExpToCode : Val E A → Val C A
  valExpToCode {ty T} x = x
  valExpToCode {fn A B} (n , envT , env , e)
    = n , envT , (envExpToCode env) , (comp e RET)
  valExpToCode {cloT x₁ x₂ x₃} (lift ())

  envExpToCode : Env E envT → Env C envT
  envExpToCode εₛ = εₛ
  envExpToCode (a ▷ₛ x) = (valExpToCode a) ▷ₛ (envExpToCode x)

valExpDistributes
  : (i : Fin n) → (env : Env E envT)
  → (valExpToCode (get i env) ≡ get i (envExpToCode env))
valExpDistributes zero (a ▷ₛ env) = refl
valExpDistributes (suc i) (a ▷ₛ env) = valExpDistributes i env

-- small-step execution semantics of code
data _,_,_==>_,_,_ : Code envT S S'' → Env C envT → Stack S
  → Code envT' S' S'' → Env C envT' → Stack S' → Set₁ where
  execHALT : HALT , env , sta ==> HALT , env , sta
  execLOOKUP : {i : Fin n}{c : Code envT (lookup envT i ∷ S) S'}
    → LOOKUP i c , env , sta ==> c , env , (get i env ▷ sta)
```

```
  execPUSH : {x : T}{c : Code envT (ty T ∷ S) S'}
    → PUSH x c , env , sta ==> c , env , (lift x ▷ sta)
  execABS : {x : {S S' : StackTy}{n : ℕ}{envT' : EnvT n}
        → Code (A Vec.∷ envT) (cloT envT' (B ∷ S) S' ∷ S) S'}
    {c : Code envT (fn A B ∷ S) S'}
    → ABS x c , env , sta ==> c , env , ((_ , (_ , (env , x))) ▷ sta)
  execAPPLY : {c : Code envT (B ∷ S) S'}{v : Val C A}{f : Val C (fn A B)}
    → {env : Env C envT}
    → {sta : Stack S}
    → {env' : Env C envT'}
    → {c' : Code (A Vec.∷ envT') (cloT envT (B ∷ S) S' ∷ S) S'}
    → f {S} {S'} ≡ (n , envT' , env' , c')
    → APPLY c , env , (v ▷ (f ▷ sta)) ==> c' , (v ▷ₛ env') , (clo env c ▷ sta)
  execRET : {S S' : StackTy}{sta : Stack S}{x : Val C A}{env' : Env C envT'}
    → {c' : Code envT' (A ∷ S) S'}
    → RET , env , (x ▷ (clo env' c' ▷ sta)) ==> c' , env' , (x ▷ sta)

-- big-step execution semantics of code
data _,_,_=>>_,_,_ : Code envT S S'' → Env C envT → Stack S
  → Code envT' S' S'' → Env C envT' → Stack S' → Set₁ where
  execRefl : {c : Code envT S S'}{env : Env C envT}{sta : Stack S}
    → c , env , sta =>> c , env , sta
  execTrans : {c : Code envT S0 S}{env : Env C envT}{sta : Stack S0}
    {c' : Code envT' S1 S}{env' : Env C envT'}{sta' : Stack S1}
    {c'' : Code envT'' S2 S}{env'' : Env C envT''}{sta'' : Stack S2}
    → c , env , sta ==> c' , env' , sta'
    → c' , env' , sta' =>> c'' , env'' , sta''
    → c , env , sta =>> c'' , env'' , sta''

execLift : {c : Code envT S S''}{env : Env C envT}{sta : Stack S}
    {c' : Code envT' S' S''}{env' : Env C envT'}{sta' : Stack S'}
    → c , env , sta ==> c' , env' , sta'
    → c , env , sta =>> c' , env' , sta'
execLift p = execTrans p execRefl
```

```
execTrans' : {c : Code envT S0 S}{env : Env C envT}{sta : Stack S0}
    {c' : Code envT' S1 S}{env' : Env C envT'}{sta' : Stack S1}
    {c'' : Code envT'' S2 S}{env'' : Env C envT''}{sta'' : Stack S2}
    → c' , env' , sta' =>> c'' , env'' , sta''
    → c , env , sta =>> c' , env' , sta'
    → c , env , sta =>> c'' , env'' , sta''
execTrans' y execRefl = y
execTrans' z (execTrans x y) = execTrans x (execTrans' z y)



------------------------------------------------------------------------
-- Calculation
------------------------------------------------------------------------

getConvEnv : (i : Fin n)(env : Env E envT)(sta : Stack S)
  → (valExpToCode (get i env) ▷ sta)
  ≡ (get i (envExpToCode env) ▷ sta)
getConvEnv zero (a ▷ₛ env) sta = refl
getConvEnv (suc i) (a ▷ₛ env) sta = getConvEnv i env sta

calculation :
  (e : STLC envT A)
  → {e' : _}
  → (env : Env E envT)
  → (sta : Stack S)
  → (c : Code envT (A :: S) S')
  → e ⇓[ env ] e'
  → comp e c , envExpToCode env , sta
  =>> c , envExpToCode env , (valExpToCode e' ▷ sta)
calculation (val i) env sta c evalVal = execTrans'
  -- exec c env (eval (val x) env ▷ sta)
  --   apply eval
  execRefl
```

```
  -- exec c env (lift x ▷ sta)
  --   define exec (PUSH x c) env sta = exec c env (lift x ▷ sta)
  (execLift execPUSH)
  -- exec (PUSH i c) env sta
calculation (var i) env sta c evalVar
  rewrite getConvEnv i env sta = execTrans'
  -- exec c env (eval (var i) env ▷ sta)
  --   apply eval
  execRefl
  -- exec c env (get i env ▷ sta)
  --   define exec (LOOKUP i c) env sta = exec c (get i env ▷ sta)
  (execLift execLOOKUP)
  -- exec (LOOKUP i c) env sta
calculation (lam e) env sta c evalLam = execTrans'
  -- exec c env (eval (lam e) env ▷ sta)
  --   apply eval
  execRefl
  -- exec c env (valExpToCode (n , envT , env , e) ▷ sta)
  --   apply valExpToCode
  -- exec c env (n , envT , envExpToCode env , comp e RET ▷ sta)
  --   define exec {n}{envT} (ABS x c) env sta
  --     = exec c env ((n , envT , env , x) ▷ sta)
  (execLift execABS)
  -- exec (ABS (comp e RET) c) env sta
calculation (app x y) env sta c
  (evalApp {env' = env'}{y' = y'}{e = e}{xy = xy} evalx evaly evale)
  with calculation x env sta (comp y (APPLY c)) evalx
  | calculation y env (_ ▷ sta) (APPLY c) evaly
  | calculation e (y' ▷ₛ env') (clo (envExpToCode env) c ▷ sta) RET evale
calculation (app x y) env sta c (evalApp evalx evaly evale)
  | ihX | ihY | ihE =
  -- exec c env (eval (app x y) env ▷ sta)
  --   apply eval
  -- exec c env (xy ▷ sta)
```

```
--   define exec RET env (x ▷ (clo env' c' ▷ sta)) = exec c' env' (x ▷ sta)
execTrans' (execLift execRET)
-- exec RET (valExpToCode y' ▷ₛ env') (xy ▷ clo env c ▷ sta)
--   induction hypothesis for e
(execTrans' ihE
-- exec (comp e RET) (valExpToCode y' ▷ₛ env') (clo env c ▷ sta)
--   define exec (APPLY c) env (v ▷ ((n , envT' , env' , c') ▷ sta))
--     = exec c' (v ▷ₛ env') (clo env c ▷ sta)
(execTrans' (execLift (execAPPLY refl))
-- exec (APPLY c) env
--   (valExpToCode y' ▷ₛ ((n , envT' , env' , comp e RET) ▷ sta))
--   induction hypothesis for y
(execTrans' ihY
-- exec (comp y (APPLY c)) (envExpToCode env)
--   (valExpToCode y' ▷ₛ (n , envT' , env' , comp e RET) ▷ sta)
--   induction hypothesis for x
ihX
-- exec (comp x (comp y (APPLY c))) (envExpToCode env) sta
)))
```

A.6 loop.agda

```
-- Now or Never: Type-Safe Compilers for Partial Languages
{-# OPTIONS --copatterns --sized-types #-}

module loop where

open import PartialUPoly
open import Data.Bool hiding (T)
open import Data.Nat
open import Data.List

variable
  i : Size
  T : Set
  S S' S'' : List Set

data Exp : Set → Set where
  val : T → Exp T
  add : Exp ℕ → Exp ℕ → Exp ℕ
  if : Exp Bool → Exp T → Exp T → Exp T
  loop : Exp T

mutual
  eval : Exp T → Partial T i
  eval (val x) = return x
  eval (add x y) =
    do n ← eval x
       m ← eval y
       return (n + m)
  eval (if b x y) =
    do b' ← eval b
       if b' then (eval x) else (eval y)
```

```
  eval loop = later (∞eval loop)



  ∞eval : ∀ {i} → Exp T → ∞Partial T i
  force (∞eval x) = eval x

data Stack : List Set → Set where
  ε : Stack []
  _▷_ : T → Stack S → Stack (T ∷ S)

data Code : List Set → List Set → Set₁ where
  PUSH : T → Code (T ∷ S) S' → Code S S'
  ADD : Code (ℕ ∷ S) S' → Code (ℕ ∷ ℕ ∷ S) S'
  IF : Code S S' → Code S S' → Code (Bool ∷ S) S'
  LOOP : Code S S'
  HALT : Code S S

∞exec : Code S S' → Stack S → ∞Partial (Stack S') i

exec : Code S S' → Stack S → Partial (Stack S') i
exec (PUSH x c) s = exec c (x ▷ s)
exec (ADD c) (m ▷ (n ▷ s)) = exec c ((n + m) ▷ s)
exec (IF c1 c2) (b ▷ s) = if b then exec c1 s else exec c2 s
exec LOOP s = later (∞exec LOOP s)
exec HALT s = return s

force (∞exec c s) = exec c s

comp : Exp T → Code (T ∷ S) S' → Code S S'
comp (val x) c = PUSH x c
comp (add x y) c = comp x (comp y (ADD c))
comp (if b x y) c = comp b (IF (comp x c) (comp y c))
comp loop c = LOOP
```

```
compile : Exp T → Code S (T ∷ S)
compile e = comp e HALT


if-then-cong-false : ∀ b {A n} {x y y' : Partial {lzero} A ∞} (eq : y ~[ n ] y')
  → (if b then x else y) ~[ n ] (if b then x else y')
if-then-cong-false true eq = ~irefl
if-then-cong-false false eq =  eq


correct : ∀ i → (e : Exp T) → (s : Stack S) → (c : Code (T ∷ S) S')
  → (do v ← eval e
        exec c (v ▷ s))
    ~[ i ]
    exec (comp e c) s
correct i (val x) s c =
  (eval (val x) >>= λ v → exec c (v ▷ s))
  ~⟨⟩
  (now x >>= λ v → exec c (v ▷ s))
  ~⟨⟩
  exec c (x ▷ s)
  ~⟨⟩
  exec (PUSH x c) s
  ∎
correct i (add x y) s c =
  (do v ← eval (add x y)
      exec c (v ▷ s))
  ~⟨⟩
  (do v ← do m ← eval x
             n ← eval y
             return (m + n)
      exec c (v ▷ s))
  ~⟨ bind-assoc (eval x) ⟩
  (do m ← eval x
      v ← do n ← eval y
             return (m + n)
```

```
      exec c (v ▷ s))
  ~⟨ bind-cong-r (eval x) (λ n → bind-assoc (eval y)) ⟩
  (do m ← eval x
      n ← eval y
      v ← return (m + n)
      exec c (v ▷ s))
  ~⟨⟩
  (do m ← eval x
      n ← eval y
      exec c ((m + n) ▷ s))
  ~⟨⟩
  (do m ← eval x
      n ← eval y
      exec (ADD c) (n ▷ (m ▷ s)))
  ~⟨ bind-cong-r (eval x) (λ m → correct i y ((m ▷ s)) ((ADD c))) ⟩
  (do m ← eval x
      exec (comp y (ADD c)) (m ▷ s))
  ~⟨ correct i x s (comp y (ADD c)) ⟩
  exec (comp x (comp y (ADD c))) s
  ∎
correct i (if b t f) s c =
  (do v ← eval (if b t f)
      exec c (v ▷ s))
  ~⟨⟩
  (do v ← do b' ← eval b
             if b' then (eval t) else (eval f)
      exec c (v ▷ s))
  ~⟨ bind-assoc (eval b) ⟩
  (do b' ← eval b
      v ← if b' then (eval t) else (eval f)
      exec c (v ▷ s))
  ~⟨⟩
  (do b' ← eval b
      v ← if b' then (eval t) else (eval f)
```

```
      exec c (v ▷ s))
  ~⟨  bind-cong-r (eval b) (λ b' → if-bind b') ⟩
  (do b' ← eval b
      if b' then (do v ← eval t
                     exec c (v ▷ s)) else (do v ← eval f
                                              exec c (v ▷ s)))
  ~⟨ bind-cong-r (eval b) (λ b' → if-then-cong b' (correct i t s c)) ⟩
  (do b' ← eval b
      if b' then (exec (comp t c) s) else (do v ← eval f
                                              exec c (v ▷ s)))
  ~⟨ bind-cong-r (eval b) (λ b' → if-then-cong-false b' (correct i f s c)) ⟩
  (do b' ← eval b
      if b' then (exec (comp t c) s) else (exec (comp f c) s))
  ~⟨⟩
  (do b' ← eval b
      exec (IF (comp t c) (comp f c)) (b' ▷ s))
  ~⟨ correct i b s (IF (comp t c) (comp f c)) ⟩
  exec (comp b (IF (comp t c) (comp f c))) s
  ∎
correct (suc j) loop s c =
  (eval loop >>= λ v → exec c (v ▷ s))
  ~⟨⟩
  (later (∞eval loop) >>= λ v → exec c (v ▷ s))
  ~⟨⟩
  (later (∞eval loop ∞>>= λ v → exec c (v ▷ s)))
  ~⟨ ~ilater (correct j loop s c) ⟩
  (later (∞exec (comp loop c) s))
  ~⟨⟩
  exec LOOP s
  ∎
correct zero loop s c = ~izero
```

A.7 stlc-coinduction.agda

```
{-# OPTIONS --copatterns --sized-types #-}


module stlc-coinduction where


open import PartialUPoly

open import Data.List hiding (lookup)

open import Data.Fin hiding (_+_)

open import Data.Nat

open import Data.Product

open import Data.Vec hiding (_>>=_)

open import Data.Bool hiding (T)

open import Relation.Binary.PropositionalEquality


variable

  i : Size

  T : Set

  n : ℕ



--------------------------------------------------------------------------------

-- Source language

--------------------------------------------------------------------------------


data Ty : Set₁ where

  Base : Set → Ty

  _↠_  : Ty → Ty → Ty

  Cont : Vec Ty n → List Ty → List Ty → Ty


variable

  S S' : List Ty

  xs : Vec Ty n
```

```
  envT : Vec Ty n
  a b x t : Ty

data Exp : Ty → Vec Ty n → Set₁ where
  Val : T → Exp (Base T) []
  Var : (i : Fin n) → Exp (lookup envT i) envT
  Lam : Exp b (a :: envT) → Exp (a ↠ b) envT
  App : Exp (a ↠ b) envT → Exp a envT → Exp b envT
  Add : Exp (Base ℕ) envT → Exp (Base ℕ) envT → Exp (Base ℕ) envT
  If  : Exp (Base Bool) envT → Exp a envT → Exp a envT → Exp a envT

data Env : Vec Ty n → Set₁
data Value : Ty → Set₁

data Env where
  ε : Env []
  _▷_ : Value x → Env xs → Env (x :: xs)

data Value where
  V : T → Value (Base T)
  Clo : {n : ℕ}{envT : Vec Ty n} → Exp b (a :: envT) → Env envT → Value (a ↠ b)

getVal : Value (Base T) → T
getVal {T} (V x) = x

getClo : Value (a ↠ b) →
  Σ ℕ (λ n → Σ (Vec Ty n) (λ envT → Exp b (a :: envT) × Env envT))
getClo (Clo {n = n}{envT = envT} f' env') = n , envT , (f' , env')

lookupE : Env envT → (n : Fin n) → Value (lookup envT n)
lookupE (x ▷ xs) zero = x
lookupE (x ▷ xs) (suc n) = lookupE xs n

mutual
```

```
  eval : Exp t envT → Env envT → Partial (Value t) i
  eval (Var v) env = return (lookupE env v)
  eval (Val x) env = return (V x)
  eval (Lam b) env = return (Clo b env)
  eval (App f x) env
    = do clo ← eval f env
         let _ , _ , f' , env' = getClo clo
         x' ← eval x env
         later (∞eval f' (x' ▷ env'))
  eval (Add x y) env
    = do m ← eval x env
         n ← eval y env
         let m' = getVal m
         let n' = getVal n
         return (V (m' + n'))
  eval (If b t f) env
    = do b' ← eval b env
         let b' = getVal b'
         if b' then eval t env else eval f env

  ∞eval : ∀ {i} → Exp t envT → Env envT → ∞Partial (Value t) i
  force (∞eval e env) = eval e env



--------------------------------------------------------------------------------
-- Target language
--------------------------------------------------------------------------------

data Code : Vec Ty n → List Ty → List Ty → Set₁ where
  PUSH : T → Code envT (Base T :: S) S' → Code envT S S'
  LOOKUP
    : (i : Fin n)
    → Code envT (lookup envT i :: S) S'
    → Code envT S S'
```

```
  ABS
    : ({S S' : List Ty}{n' : ℕ}{envT' : Vec Ty n'}
        → Code (a :: envT) (Cont envT' (b :: S) S' :: S) S')
    → Code envT (a ↠ b :: S) S'
    → Code envT S S'
  RET : {n' : ℕ}{envT' : Vec Ty n'}
    → Code envT (a :: Cont envT' (a :: S) S' :: S) S'
  APP
    : Code envT (b :: S) S'
    → Code envT (a :: a ↠ b :: S) S'
  FIX  : {a : Ty}
    → Code envT (((a ↠ a) ↠ a) :: S) S'
    → Code envT S S'
  ADD
    : Code envT (Base ℕ :: S) S'
    → Code envT (Base ℕ :: Base ℕ :: S) S'
  IF
    : Code envT S S'
    → Code envT S S'
    → Code envT (Base Bool :: S) S'


mutual
  data Env' : Vec Ty n → Set₁ where
    ε : Env' []
    _▷_ : Value' x → Env' xs → Env' (x :: xs)


  data Value' : Ty → Set₁ where
    V' : T → Value' (Base T)
    Clo'
      : ({S S' : List Ty}{n' : ℕ}{envT' : Vec Ty n'}
          → Code (a :: envT) (Cont envT' (b :: S) S' :: S) S')
      → Env' envT
      → Value' (a ↠ b)
    Restore : {S S' : List Ty}{envT : Vec Ty n}
```

```
      → Code envT S S'
      → Env' envT
      → Value' (Cont envT S S')

lookupE' : Env' envT → (i : Fin n) → Value' (lookup envT i)
lookupE' (x ▷ env) zero = x
lookupE' (x ▷ env) (suc i) = lookupE' env i

data Stack : List Ty → Set₁ where
  ε : Stack []
  _▷_ : Value' t → Stack S → Stack (t ∷ S)

mutual
  {-# TERMINATING #-}
  exec : Code envT S S' → Stack S → Env' envT → Partial (Stack S') i
  exec (PUSH x c) s env
    = exec c (V' x ▷ s) env
  exec (LOOKUP i c) s env
    = exec c (lookupE' env i ▷ s) env
  exec (ADD c) (V' n ▷ (V' m ▷ s)) env
    = exec c (V' (m + n) ▷ s) env
  exec (IF t f) (V' b' ▷ s) env
    = if b' then exec t s env else exec f s env
  exec (ABS f c) s env
    = exec c ((Clo' f env) ▷ s) env
  exec (APP c) (x ▷ (Clo' f env' ▷ s)) env
    = later (∞exec f ((Restore c env) ▷ s) (x ▷ env'))
  exec RET (v ▷ (Restore f env' ▷ s)) env
    = exec f (v ▷ s) env'
  exec (FIX c) s env
    = exec c ((Clo' (LOOKUP zero (FIX (LOOKUP zero (APP (APP RET))))) env)
                ▷ s) env

  ∞exec : Code envT S S' → Stack S → Env' envT → ∞Partial (Stack S') i
```

```
  force (∞exec c s env) = exec c s env


comp : Exp t envT → Code envT (t :: S) S' → Code envT S S'
comp (Val x) c    = PUSH x c
comp (Var i) c    = LOOKUP i c
comp (Lam x) c    = ABS (comp x RET) c
comp (App f x) c  = comp f (comp x (APP c))
comp (Add x y) c  = comp x (comp y (ADD c))
comp (If b t f) c = comp b (IF (comp t c) (comp f c))



-------------------------------------------------------------------------------
-- Calculation
-------------------------------------------------------------------------------


mutual
  convE : Env envT → Env' envT
  convE ε = ε
  convE (x ▷ env) = (convV x) ▷ (convE env)


  convV : Value a → Value' a
  convV (V x) = V' x
  convV (Clo b env) = Clo' (comp b RET) (convE env)


convVLookup : (env : Env envT) → (v : Fin n)
  → convV (lookupE env v) ≡ lookupE' (convE env) v
convVLookup (x ▷ env) zero = refl
convVLookup (x ▷ env) (suc v) = convVLookup env v


Clo≡ : (clo : Value (a ↠ b))
  → let _ , _ , f , env = getClo clo in Clo f env ≡ clo
Clo≡ (Clo f env) = refl


~irewrite : ∀ {i} → {A : Set ℓ}{a b : Partial A ∞} → a ≡ b → a ~[ i ] b
```

```
~irewrite p rewrite p = ~irefl


if-then-cong' : ∀ b {ℓ n}{A : Set ℓ} {x y x' y' : Partial A ∞}
  (eqT : x ~[ n ] x') → (eqF : y ~[ n ] y')
  → (if b then x else y) ~[ n ] (if b then x' else y')
if-then-cong' false eqT eqF = eqF
if-then-cong' true  eqT eqF = eqT


spec : ∀ {S S'} i (env : Env envT) → (e : Exp t envT) → (s : Stack S)
  → (c : Code envT (t :: S) S')
  → (do v ← eval e env
        exec c (convV v ▷ s) (convE env))
    ~[ i ]
    exec (comp e c) s (convE env)
spec zero env e s c = ~izero
spec i env (Val x) s c =
  (do v ← eval (Val x) env
      exec c (convV v ▷ s) (convE env))
  ~⟨⟩
  (do v ← return (V x)
      exec c (convV v ▷ s) (convE env))
  ~⟨⟩
  exec c (V' x ▷ s) (convE env)
  ~⟨⟩
  exec (PUSH x c) s (convE env)
  ∎
spec i env (Var n) s c =
  (do v ← eval (Var n) env
      exec c (convV v ▷ s) (convE env))
  ~⟨⟩
  (do v ← return (lookupE env n)
      exec c (convV v ▷ s) (convE env))
  ~⟨⟩
  exec c (convV (lookupE env n) ▷ s) (convE env)
```

```
  ~⟨ ~irewrite (cong (λ x → exec c (x ▷ s) (convE env)) (convVLookup env n)) ⟩
  exec c (lookupE' (convE env) n ▷ s) (convE env)
  ~⟨⟩
  exec (LOOKUP n c) s (convE env)
  ∎
spec i env (Lam b) s c =
  (do v ← eval (Lam b) env
      exec c (convV v ▷ s) (convE env))
  ~⟨⟩
  (do v ← return (Clo b env)
      exec c (convV v ▷ s) (convE env))
  ~⟨⟩
  exec c (convV (Clo b env) ▷ s) (convE env)
  ~⟨⟩
  exec (ABS (comp b RET) c) s (convE env)
  ∎
spec (suc i) env (App f x) s c =
  (do v ← eval (App f x) env
      exec c (convV v ▷ s) (convE env))
  ~⟨⟩
  (do v ← do clo ← eval f env
             let _ , _ , f' , env' = getClo clo
             x' ← eval x env
             later (∞eval f' (x' ▷ env'))
      exec c (convV v ▷ s) (convE env))
  ~⟨ bind-assoc (eval f env) ⟩
  (do clo ← eval f env
      v ← do let _ , _ , f' , env' = getClo clo
             x' ← eval x env
             later (∞eval f' (x' ▷ env'))
      exec c (convV v ▷ s) (convE env))
  ~⟨⟩
  (do clo ← eval f env
      let _ , _ , f' , env' = getClo clo
```

```
    v ← do x' ← eval x env
           later (∞eval f' (x' ▷ env'))
    exec c (convV v ▷ s) (convE env))
~⟨ bind-cong-r (eval f env) (λ v → bind-assoc (eval x env)) ⟩
(do clo ← eval f env
    let _ , _ , f' , env' = getClo clo
    x' ← eval x env
    v ← later (∞eval f' (x' ▷ env'))
    exec c (convV v ▷ s) (convE env))
~⟨⟩
(do clo ← eval f env
    let _ , _ , f' , env' = getClo clo
    x' ← eval x env
    v ← later (∞eval f' (x' ▷ env'))
    exec RET (convV v ▷ (Restore c (convE env) ▷ s)) (convE (x' ▷ env')))
~⟨⟩
(do clo ← eval f env
    let _ , _ , f' , env' = getClo clo
    x' ← eval x env
    later (∞eval f' (x' ▷ env') ∞>>= λ v →
      exec RET (convV v ▷ (Restore c (convE env) ▷ s)) (convE (x' ▷ env'))))
~⟨ bind-cong-r (eval f env) (λ clo → bind-cong-r (eval x env)
     (let _ , _ , f' , env' = getClo clo in
     λ x' → ~ilater (spec i (x' ▷ env') f' (Restore c (convE env) ▷ s) RET))) ⟩
(do clo ← eval f env
    let _ , _ , f' , env' = getClo clo
    x' ← eval x env
    later (∞exec (comp f' RET) (Restore c (convE env) ▷ s) (convE (x' ▷ env'))))
~⟨⟩
(do clo ← eval f env
    let _ , _ , f' , env' = getClo clo
    x' ← eval x env
    exec (APP c) (convV x' ▷ (Clo' (comp f' RET) (convE env') ▷ s)) (convE env))
~⟨ bind-cong-r (eval f env) (λ clo
```

```
      → let _ , _ , f' , env' = getClo clo in spec (suc i) env x _ _) ⟩
  (do clo ← eval f env
      let _ , _ , f' , env' = getClo clo
      exec (comp x (APP c)) (Clo' (comp f' RET) (convE env') ▷ s) (convE env))
  ~⟨⟩
  (do clo ← eval f env
      let _ , _ , f' , env' = getClo clo
      exec (comp x (APP c)) (convV (Clo f' env') ▷ s) (convE env))
  ~⟨ bind-cong-r (eval f env) (λ clo
      → let _ , _ , f' , env' = getClo clo in
      ~irewrite (cong (λ v → exec (comp x (APP c)) (convV v ▷ s)
                    (convE env)) (Clo≡ clo))) ⟩
  (do clo ← eval f env
      exec (comp x (APP c)) (convV clo ▷ s) (convE env))
  ~⟨ spec (suc i) env f s (comp x (APP c)) ⟩
  exec (comp f (comp x (APP c))) s (convE env)
  ∎
spec i env (Add x y) s c =
  (do v ← eval (Add x y) env
      exec c (convV v ▷ s) (convE env))
  --~⟨ add-lemma' i {!!} {!!} ⟩
  ~⟨ bind-assoc (eval x env) ⟩
  (do m ← eval x env
      v ← do n ← eval y env
             let m' = getVal m
             let n' = getVal n
             return (V (m' + n'))
      exec c (convV v ▷ s) (convE env))
  ~⟨ bind-cong-r (eval x env) (λ m → bind-assoc (eval y env)) ⟩
  (do m ← eval x env
      n ← eval y env
      let m' = getVal m
      let n' = getVal n
      v ← return (V (m' + n'))
```

```
    exec c (convV v ▷ s) (convE env))
~⟨⟩
(do m ← eval x env
    n ← eval y env
    let m' = getVal m
    let n' = getVal n
    exec c (convV (V (m' + n')) ▷ s) (convE env))
~⟨⟩
(do m ← eval x env
    let m' = getVal m
    n ← eval y env
    let n' = getVal n
    exec c (convV (V (m' + n')) ▷ s) (convE env))
~⟨⟩
(do m ← eval x env
    let m' = getVal m
    n ← eval y env
    let n' = getVal n
    exec c (convV (V (m' + n')) ▷ s) (convE env))
~⟨⟩ -- define ADD
(do m ← eval x env
    let m' = getVal m
    n ← eval y env
    let n' = getVal n
    exec (ADD c) (convV (V n') ▷ (convV (V m') ▷ s)) (convE env))
~⟨ bind-cong-r (eval x env) (λ { (V m) → bind-cong-r (eval y env)
                                          (λ { (V n) → ~irefl })}) ⟩
(do m ← eval x env
    n ← eval y env
    exec (ADD c) (convV n ▷ (convV m ▷ s)) (convE env))
~⟨ bind-cong-r (eval x env) (λ m → spec i env y (convV m ▷ s) (ADD c)) ⟩
(do m ← eval x env
    exec (comp y (ADD c)) (convV m ▷ s) (convE env))
~⟨ spec i env x s (comp y (ADD c)) ⟩
```

```
  exec (comp x (comp y (ADD c))) s (convE env)
  ∎
spec i env (If b t f) s c =
  (do v ← eval (If b t f) env
      exec c (convV v ▷ s) (convE env))
  ~⟨⟩
  (do v ← do b' ← eval b env
             let b' = getVal b'
             if b' then eval t env else eval f env
      exec c (convV v ▷ s) (convE env))
  ~⟨⟩
  (do v ← do b' ← eval b env
             let b' = getVal b'
             if b' then eval t env else eval f env
      exec c (convV v ▷ s) (convE env))
  ~⟨ bind-assoc (eval b env) ⟩
  (do b' ← eval b env
      let b' = getVal b'
      v ← if b' then eval t env else eval f env
      exec c (convV v ▷ s) (convE env))
  ~⟨ bind-cong-r (eval b env) (λ b' → if-bind (getVal b')) ⟩
  (do b' ← eval b env
      let b' = getVal b'
      if b' then (do v ← eval t env
                     exec c (convV v ▷ s) (convE env))
            else (do v ← eval f env
                     exec c (convV v ▷ s) (convE env)))
  ~⟨ bind-cong-r (eval b env) (λ b' → let b' = getVal b' in
       if-then-cong' b' (spec i env t s c) (spec i env f s c)) ⟩
  (do b' ← eval b env
      let b' = getVal b'
      if b' then exec (comp t c) s (convE env)
            else exec (comp f c) s (convE env))
  ~⟨⟩
```

```
(do b' ← eval b env
    let b' = getVal b'
    exec (IF (comp t c) (comp f c)) (convV (V b') ▷ s) (convE env))
~⟨ bind-cong-r (eval b env) (λ { (V b') → ~irefl } ) ⟩
(do b' ← eval b env
    exec (IF (comp t c) (comp f c)) (convV b' ▷ s) (convE env))
~⟨ spec i env b s _ ⟩
exec (comp b (IF (comp t c) (comp f c))) s (convE env)
∎
```