Calculating Dependently-Typed Compilers (Functional Pearl)

MITCHELL PICKARD, University of Nottingham, UK
GRAHAM HUTTON, University of Nottingham, UK

Compilers are difficult to write, and difficult to get right. Bahr and Hutton recently developed a new technique for calculating compilers directly from specifications of their correctness, which ensures that the resulting compilers are correct-by-construction. To date, however, this technique has only been applicable to source languages that are untyped. In this article, we show that moving to a dependently-typed setting allows us to naturally support typed source languages, ensure that all compilation components are type-safe, and make the resulting calculations easier to mechanically check using a proof assistant.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → *Type theory*; **Logic and verification**.

Additional Key Words and Phrases: program calculation, dependent types

ACM Reference Format:
Mitchell Pickard and Graham Hutton. 2021. Calculating Dependently-Typed Compilers (Functional Pearl). *Proc. ACM Program. Lang.* 5, ICFP, Article 82 (August 2021), 27 pages. https://doi.org/10.1145/3473587

1 INTRODUCTION

Bahr and Hutton [2015] recently presented a new approach to developing compilers whose correctness is guaranteed by the manner in which they are constructed. The basic approach is as follows. We begin by defining the syntax of the source language, and a semantics for this language. Our 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. The approach builds upon earlier work by Wand [1982], Meijer [1992] and Ager *et al.* [2003] and was originally developed for stack-based target languages, but has also been extended to register-based languages [Bahr and Hutton 2020; Hutton and Bahr 2017].

Compiler calculations in the above approach are 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

Authors' addresses: Mitchell Pickard, School of Computer Science, University of Nottingham, UK, mitchell.pickard@nottingham.ac.uk; Graham Hutton, School of Computer Science, University of Nottingham, UK, graham.hutton@nottingham.ac.uk.

2475-1421/2021/8-ART82
https://doi.org/10.1145/3473587

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 article, 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.

We introduce our approach using the simple expression language used by McKinna and Wright in their development of a type-correct, stack-safe, provably correct expression compiler [2006]. We show how our approach can be used to calculate their well-typed compiler, with the correctness proof and all the required compilation machinery falling naturally out of the calculation process. As a more sophisticated example, we then calculate a well-typed compiler for a language with exceptions. In the past it proved difficult even to write such a compiler, so it is pleasing that our new methodology allowed us to calculate a compiler in a systematic manner. To further demonstrate the generality of the technique, we have also calculated a well-typed compiler for the simply-typed lambda calculus, which is included in the online supplementary material.

The aim of this article is to set out the basic methodology for calculating dependently-typed compilers, and establish its feasibility. In particular, we are not seeking to develop verified real-world compilers, but rather to establish a principled approach to *discovering* basic compilation techniques in a systematic manner. Despite the fact that dependently-typed compilers and compiler calculation are both long-standing research topics, these ideas have never been combined before. The article is targeted at readers with experience of a functional language such as Haskell, but does not assume specialist knowledge of Agda, dependent types or compiler calculation, and is written in a tutorial style to make the ideas accessible to a broader audience. The supplementary material for the article includes verified versions of all the calculations in Agda.

2 EXPRESSION LANGUAGE

In this section we introduce our compiler calculation technique by means of a simple example, and show how it benefits from the use of dependent types. We consider an expression language with natural numbers and Boolean values as basic types that is not type-safe in a simply-typed setting, but can be made type-safe using dependent types and inductive families as shown by McKinna and Wright [2006]. We begin by defining the language and its semantics, then specify what it means for the compiler to be correct, and finally show how the compiler can be calculated.

For readers familiar with Haskell but not Agda, we remark on a few syntactic differences. First of all, the symbols `:` and `::` are interchanged, so that `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. Thirdly, we can define binary operators such as `_⊗_`, which can be used by inserting the arguments in place of the underscores, as in `a ⊗ b`. Finally, Agda supports Unicode, so all Unicode symbols that are used, such as the function arrow →, are valid Agda code.

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. The type declaration Exp : Set means that Exp is a type whose elements are not other types, which corresponds to the familiar notion of defining a new datatype in Haskell.

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 paper, 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 datatypes (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 simply return a value of this type. This approach automatically ensures 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.

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 simply 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)
```

The `variable` declaration allows us to assume that certain variables are always of type `List Set`, i.e. a list of types. A stack is then 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 Bahr and Hutton's setting [2015], code is an algebraic datatype (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 simple equation:

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

2.3 Compiler Calculation

To calculate the compiler, we proceed from Equation (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 introduce new constructors into the `Code` type, along with their interpretation by the execution function `exec`.

Case: `e = val x`.

We begin with the right-hand side of Equation (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 simply 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` simply 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 (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 (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
```

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 did 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 Bahr and Hutton's original setting [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:

```
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 functions such as cong to apply an equality in a particular context. All of the calculations in the article have been mechanised in Agda, and are available online as supplementary material.

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.

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)                (2)
```

That is, compiling an expression and executing the resulting code gives the same result as executing the additional code 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 (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 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 (2) to be applied:

```
    eval e ▷ s
=    { define: exec HALT s = s }
    exec HALT (eval e ▷ s)
=    { equation (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
```

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. If desired, however, we can transform the compiler to produce linear code by modifying `IF` to take a code pointer as its first argument rather than code itself [Bahr and Hutton 2015; Rouvoet et al. 2021].

4 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.

4.1 Source Language

We consider a simple expression language similar to that used by Bahr and Hutton [2015], with the addition of a Boolean index that specifies if an 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)
```

As previously, the constructors `val` and `add` allow expressions to be built up from natural numbers using an addition operator. Informally, the new constructor `throw` results in an exception being thrown, while the expression `catch x h` behaves as the expression `x` unless that throws an exception, in which case the catch behaves as the *handler* expression `h`.

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. However, it is straightforward to index the expression type over the return type as in the previous example.

4.2 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. In the former case, 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 addition cannot throw an exception, neither can its two 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.

4.3 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 our language, 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 did 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 converts the constructor `nat` into the type ℕ:

```
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:

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

That is, a stack is indexed by a list of types, and is either empty or formed by pushing a value of type `T` onto a stack of type `S` to give a stack of type `T :: S`. 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 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)` just requires adding a proof `p : b ≡ false` that the expression cannot throw an exception:

$$\texttt{exec (comp p e c) s = exec c (eval p e ▷ s)} \qquad (3)$$

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

4.4 Compiler Calculation

As previously, we calculate the compiler from Equation (3) by induction on the expression `e`. In each case, we aim to transform `exec c (eval p e ▷ s)` into the form `exec c' s` for some code `c'`, and 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 (3) 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 (3). 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)                (4)
    nothing → fail s
```

To continue the calculation, we must transform the term that we are manipulating into a form that matches Equation (4). 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 (4), the `nothing` branch must have the form `fail s'` for any 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 (4), 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 (4) 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 (4) for x }
     exec (comp? x (UNMARK c)) (comp refl h c ▷ s)
```

The final step is to bring the stack into the correct form to match the left-hand side of Equation (3). 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))
```

4.5 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 (4) 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'
```

In summary, 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'
```

4.6 Compiler Calculation II

We proceed from Equation (4) 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 case for throw, 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, 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 (4). 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 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. We can determine whether the handler is on the top of the stack by pattern matching on the implicit argument `S''`, which is the list of types of elements prior to the handler:

```
   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 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 first defining equation for `fail`:

```
     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 function compile and the HALT constructor can be calculated as in Section 3.

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

4.7 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.

4.8 Reflection

Our compiler for expressions that may throw an uncaught exception, comp?, is essentially the same as that developed by Bahr and Hutton [2015]. However, 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.

5 RELATED WORK

A review of the main historical developments in the area of compiler calculation is provided by Bahr and Hutton [2015]. In this section we survey some of the main related work in the use of dependent types for compiler verification and calculation.

The compiler we developed in Section 2.3 is essentially the same as that presented by McKinna and Wright [2006]. However, we have shown how each case can be calculated from the correctness specification, rather than having to write the compiler and separately prove its correctness. Bahr and Hutton [2015] calculated several compilers for untyped languages, including simple arithmetic expressions and a language with exceptions as we have in this article. However, our use of dependent types allows us to be more precise about these languages, such as by using a type index to specify whether a given expression can throw an exception or not. Alongside a compiler for expressions which may throw an exception, we also calculated a compiler for expressions which are statically known to not throw an (uncaught) exception, allowing them to be treated as pure.

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.

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, just as the languages in this article are, ensuring that at each stage only well-formed and well-typed programs can be represented. 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.

Brady and Hammond [2006] show another use of 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.

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 (such as cartesian closed categories) 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 article. He shows how to calculate a compiler targeting a stack machine supporting primitive functions and simple stack operations using the laws of cartesian categories.

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 article 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.

6 CONCLUSION AND FURTHER WORK

In this article we have shown how Bahr and Hutton's approach to calculating compilers for stack-based machines can be adapted to a dependently-typed setting. In this final section, we summarise the benefits of the approach, and discuss how it might be mechanised and generalised.

Benefits. Because our methodology builds on [Bahr and Hutton 2015], it inherits the same underlying benefits. First of all, our approach is based on the idea of calculating compilers *directly* from high-level specifications of their correctness, rather than indirectly by applying a series of transformations to the semantics for the source language. Secondly, it only requires *simple* equational reasoning techniques, and avoids the need for more sophisticated concepts such as continuations and defunctionalisation. Thirdly, it provides a principled approach to *discover* new compilation techniques, and to consider alternative design choices during the calculation process. Finally, it is readily amenable to mechanical *formalisation* using a proof assistant.

Moving to a dependently-typed setting then brings further benefits, in particular by allowing us to statically guarantee that only *well-formed* source programs and target code can be constructed. Moreover, because the virtual machine stack is now also guaranteed to be well-formed, the execution function for the machine becomes *total*, which simplifies the process of mechanically verifying the resulting calculations as many proof assistants require that all functions are total.

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.

Generalisation. 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. 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]. As a first step in this direction, we have calculated a dependently-typed compiler for the simply-typed lambda calculus, which is available in the online supplementary material.

In addition, the technique could also be adapted to target real hardware or virtual machines, building 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. 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?

ACKNOWLEDGEMENTS

We would like to thank Arjen Rouvoet and the ICFP reviewers for useful comments and suggestions on this article. This work was funded by the EPSRC grant EP/P00587X/1, *Mind the Gap: Unified Reasoning About Program Correctness and Efficiency.*

REFERENCES

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. *From Interpreter to Compiler and Virtual Machine: A Functional Derivation.* Technical Report RS-03-14. BRICS, Department of Computer Science, University of Aarhus.

Andrew Appel, Lennart Beringer, Adam Chlipala, Benjamin Pearce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2015. The Science of Deep Specification. https://deepspec.org/

Patrick Bahr and Graham Hutton. 2015. Calculating Correct Compilers. *Journal of Functional Programming* 25 (Sept. 2015).

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

Edwin Brady and Kevin Hammond. 2006. A Verified Staged Interpreter is a Verified Compiler. In *GPCE'06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering.*

Adam Chlipala. 2007. 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.*

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

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

Conal Elliott. 2018. Calculating Compilers Categorically. (2018). https://tinyurl.com/5de9xfm2

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

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

Graham Hutton and Patrick Bahr. 2017. Compiling a 50-Year Journey. *Journal of Functional Programming* 27 (Sept. 2017).

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

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

Simon Marlow (Ed.). 2010. *Haskell Language Report.* Available on the web from: https://www.haskell.org/definition/haskell2010.pdf.

Conor McBride. 2011. Ornamental Algebras, Algebraic Ornaments. (2011). University of Strathclyde.

James Mckinna and Joel Wright. 2006. A Type-Correct, Stack-Safe, Provably Correct, Expression Compiler in Epigram. (2006).

Erik Meijer. 1992. *Calculating Compilers*. Ph.D. Dissertation. Katholieke Universiteit Nijmegen.

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

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

Alberto Pardo, Emmanuel Gunther, Miguel Pagano, and Marcos Viera. 2018. An Internalist Approach to Correct-by-Construction Compilers. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18)*.

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

Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. 2021. Intrinsically Typed Compilation with Nameless Labels. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 22 (Jan. 2021).

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

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

Philip Wadler. 1989. The Concatenate Vanishes. (1989). University of Glasgow.

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