JFP **35**, e20, 27 pages, 2025.
doi:10.1017/S0956796825100087

Towards type-directed compiler calculation

WOUTER SWIERSTRA
Utrecht University, Netherlands
(*e-mail:* `w.s.swierstra@uu.nl`)

Abstract

This paper explores a principled approach to calculating abstract machines and associated compilers, starting from an intrinsically typed interpreter. After deriving a compiler for a simple expression language in some detail, the first steps of this calculation are repeated to derive an optimizing evaluator for the simply typed lambda calculus.

1 Introduction

A compiler translates high-level programs to machine instructions. These instructions may be drawn from some fixed instruction set for a particular piece of hardware, but they may equally well be instructions for a theoretical model of such hardware, sometimes referred to as an *abstract machine*. To verify that a compiler preserves the semantics of the source language is no small task (Leroy, 2009). Recently, there has been a renewed interest in *calculating* compilers from their specification (Bahr & Hutton, 2015, 2020, 2022; Pickard & Hutton, 2021). Such calculations guarantee that the compiler is correct, without necessitating further proof *post hoc*.

These calculations, however, are not entirely mechanical. In particular, when deriving a stack-based evaluator from one written in direct style (the types of), the stack machine operations are invented during calculation, rather than *calculated* from their specification. This paper aims to explore a recurring pattern in these calculations, showing how a compiler may be derived by first calculating a stack-based evaluator in a type-directed fashion. The complete derivation of a verified compiler proceeds in several steps:

- We will start by defining an intrinsically typed evaluator. This paper covers two such evaluators: one for a simple arithmetic language (Section 2.1) and the other for the simply typed lambda calculus (Section 3).
- Next, we will formulate the *types* of the target stack machine, together with the correctness property that the stack machine should satisfy (Section 2.2). This correctness property forms the specification of our desired stack machine.
- A straightforward calculation, starting from this specification, yields a stack-based interpreter, both for arithmetic expressions (Section 2.3) and the lambda calculus (Section 4).

- To derive a compiler for the language of arithmetic expressions, the stack-based interpreter is first written in *continuation passing style* (Section 2.4). From this interpreter, we will read off a linear instruction set corresponding to our target language. The compiler itself maps each language construct to its corresponding instruction; the derived execution of the target language interprets each instruction to the corresponding stack machine operation (Section 2.5).
- Alternatively, defunctionalizing the CPS transformed stack-based interpreter yields a tail-recursive abstract machine in the style of Ager *et al.* (2003) (Section 2.6).

Crucially, *the types drive the calculations*. For the lambda calculus, we do not replay the same series of steps as for our arithmetic language. Instead, we will focus on deriving *optimizations* by modifying the types used to represent variable binding (Section 5). The calculations presented here are inspired by those by Meijer (1992); each step in our calculations has been fully formalized in the interactive proof assistant and dependently typed programming language Agda (Norell, 2007).

2 A simple expression language

Before we can start calculating a compiler, we need to fix the source language and its semantics. We will begin by writing an intrinsically typed tagless interpreter in the style of Augustsson & Carlsson (1999). From this interpreter, we will derive the compiler for a typed stack machine language, as proposed by McKinna & Wright (2006). This result is not surprising, but it will showcase the style of calculation that we will use throughout the remainder of this article.

2.1 An intrinsically typed interpreter

A well typed interpreter needs a language of types. For the sake of simplicity, we will consider a simple universe with natural numbers, booleans, and functions:

```
data U : Set where
  bool : U
  nat  : U
  _⇒_ : U → U → U
```

Any element of this universe can be mapped to its corresponding Agda type. We will sometimes refer to this as the *denotation* or *interpretation* of our (object) types:

```
⟦_⟧ : U → Set
⟦ bool ⟧   = Bool
⟦ nat ⟧    = ℕ
⟦ a ⇒ b ⟧ = ⟦ a ⟧ → ⟦ b ⟧
```

With our types in place, we can now define the expression language that is the object of our study.

```
data Expr : U → Set where
  val  : ℕ → Expr nat
  plus : Expr (nat ⇒ nat ⇒ nat)
  if   : Expr (bool ⇒ a ⇒ a ⇒ a)
  app  : Expr (a ⇒ b) → Expr a → Expr b
```

Our expressions consist of (natural number) values, additions, conditional statements, and application. We could add numerous other operations – such as Boolean constants, comparisons, other arithmetic operations, and so forth – none of which would require any novel insight when performing the upcoming calculations. We have chosen to keep the language small, yet capturing several different language constructs. In contrast to other compiler calculations (Bahr & Hutton, 2015), we have chosen to include different types, including functions and a polymorphic language construct. Note that, although we have a single node for applications – rather than make each operator store its recursive arguments – this is not an essential design choice, but allows for parts of the calculation to be reused when considering the lambda calculus. The source language is chosen to exhibit various programming language features, rather than be particularly expressive. There is, for example, no way to write a boolean valued expression with only this choice of constructors.

We can reap the rewards of our well typed syntax when defining an interpreter. The type of our eval function maps expressions to their corresponding values:

```
eval : Expr a → ⟦ a ⟧
eval (val x)       = x
eval if            = if_then_else_
eval plus          = _+_
eval (app t₁ t₂)   = (eval t₁) (eval t₂)
```

The tagless interpreter relies on the static type structure: each operator of the expression language is mapped to its Agda counterpart; the app constructor is mapped to Agda's application. The types structure imposed on our expression language ensures this definition type checks.

2.2 Stack-based interpreters

With this interpreter in place, we can now start working towards our calculation. The stack machine that we will calculate will use the same types – but we will assign them a different meaning. We aim to derive a stack-based evaluator where a function's arguments are stored on a stack. To type such stacks, we extend the meaning of our types to work over lists of types, storing an element of the corresponding type:

```
⟦_⟧⋆ : List U → Set
⟦ [] ⟧⋆       = ⊤
⟦ a :: as ⟧⋆  = ⟦ a ⟧ × ⟦ as ⟧⋆
```

Now we can define an alternative meaning for our types, mapping types to stack-based functions. Each such stack-based function expects its arguments pushed on the top of the stack, replacing them with the result of the function call:

$$
\begin{aligned}
&[\![\_]\!]_s \;:\; \mathsf{U} \to \mathsf{Set}\\
&[\![\, \mathsf{a} \,]\!]_s \;=\; \forall \{\sigma \;:\; \mathsf{List\,U}\} \to \mathsf{args\,a}\,\sigma \to \mathsf{result\,a}\,\sigma
\end{aligned}
$$

This definition uses two auxiliary functions, args and result, that compute the shape of the argument stack and resulting stack, respectively:

$$
\begin{aligned}
&\mathsf{args} \;:\; \mathsf{U} \to \mathsf{List\,U} \to \mathsf{Set}\\
&\mathsf{args}\,(\mathsf{a} \Rightarrow \mathsf{b})\,\sigma \;=\; [\![\, \mathsf{a} \,]\!]_s \times \mathsf{args\,b}\,\sigma\\
&\mathsf{args\,nat}\;\;\sigma \;=\; [\![\, \sigma \,]\!]^{\star}\\
&\mathsf{args\,bool}\,\sigma \;=\; [\![\, \sigma \,]\!]^{\star}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{result} \;:\; \mathsf{U} \to \mathsf{List\,U} \to \mathsf{Set}\\
&\mathsf{result\,bool}\;\;\sigma \;=\; \mathsf{Bool} \times [\![\, \sigma \,]\!]^{\star}\\
&\mathsf{result\,nat}\;\;\sigma \;=\; \mathbb{N} \times [\![\, \sigma \,]\!]^{\star}\\
&\mathsf{result}\,(\mathsf{a} \Rightarrow \mathsf{b})\,\sigma \;=\; \mathsf{result\,b}\,\sigma
\end{aligned}
$$

At this point, an example is in order. Previously, we interpreted our object type nat as Agda's natural numbers, ℕ. With the stack-based interpretation of types, however, this becomes a *function* from stacks to stacks:

$$
[\![\, \mathsf{nat} \,]\!]_s \;=\; \forall \{\sigma\} \to [\![\, \sigma \,]\!]^{\star} \to \mathbb{N} \times [\![\, \sigma \,]\!]^{\star}
$$

Rather than return a natural number directly, we now expect an input stack and produce an output stack with a natural number on it. More generally, a function type $\mathsf{a} \Rightarrow \mathsf{b}$ becomes stack transformer, taking the arguments a on the top of the input stack, producing an output stack storing a value of type b on its top. In contrast to most stack-based semantics, the values stored on the stack will be functions themselves, rather than a natural number. We will revisit the higher-order nature of the values stored on the stack in the discussion towards the end of this paper.

Equivalence of type denotations. The two denotations we have defined for our types are equivalent. We can define the conversion in either direction by induction on the type structure:

$$
\begin{aligned}
&\alpha \;:\; \forall\,\mathsf{a} \to [\![\, \mathsf{a} \,]\!] \to [\![\, \mathsf{a} \,]\!]_s\\
&\gamma \;:\; \forall\,\mathsf{a} \to [\![\, \mathsf{a} \,]\!]_s \to [\![\, \mathsf{a} \,]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\alpha\,\mathsf{bool}\;\;\mathsf{x}\,\xi \;=\; (\mathsf{x}\,,\xi)\\
&\alpha\,\mathsf{nat}\;\;\mathsf{x}\,\xi \;=\; (\mathsf{x}\,,\xi)\\
&\alpha\,(\mathsf{a} \Rightarrow \mathsf{b})\,\mathsf{f}\,(\mathsf{x}\,,\xi) \;=\; \alpha\,\mathsf{b}\,(\mathsf{f}\,(\gamma\,\mathsf{a}\,\mathsf{x}))\,\xi
\end{aligned}
$$

$$
\begin{aligned}
&\gamma\,\mathsf{bool}\;\;\mathsf{f} \;=\; \mathsf{proj}_1\,(\mathsf{f}\,\mathsf{tt})\\
&\gamma\,\mathsf{nat}\;\;\mathsf{f} \;=\; \mathsf{proj}_1\,(\mathsf{f}\,\mathsf{tt})\\
&\gamma\,(\mathsf{a} \Rightarrow \mathsf{b})\,\mathsf{f} \;=\; \lambda\,\mathsf{x} \to \gamma\,\mathsf{b}\,(\lambda\,\xi \to \mathsf{f}\,(\alpha\,\mathsf{a}\,\mathsf{x}\,,\xi))
\end{aligned}
$$

At base types, nat and bool, the conversions are straightforward. The conversion to stack-based types pushes the argument value on the top of the stack. In the opposite direction, the stack-based semantics is run on an empty stack, tt, before projecting out the desired value from the top of resulting stack. The case for function types is more involved. Let us look at both cases in more detail:

- The α function converts a function in direct style to its stack-based equivalent. If we ignore the conversions α and γ on the right-hand side of the definition, we see that top of the stack x, is fed into f, before recursing and feeding any remaining stack arguments into the result.
- Conversely, the γ function converts a stack-based function to one in direct style. Once again, we begin by ignoring the conversions α and γ that occur in the function definition. The lambda bound argument x is pushed on to the stack ξ; the resulting stack is then passed to the function f. The two conversions ensure that the top of the stack stores a values of type $[\![\, \mathsf{a} \,]\!]_{\mathsf{s}}$ and the lambda's body returns a value of type $[\![\, \mathsf{b} \,]\!]$ as required.

These functions combine two ideas that can be found in the literature. The names are borrowed from Meijer (1992), who refers to these functions as the abstraction and concretization functions respectively. In his thesis, he shows how these functions form a generalized form of curry and uncurry, recursively rearranging a function's arguments in the appropriate fashion. The curry and uncurry functions typically have the following type:

```
uncurry : (a → b → c) → (a × b) → c
curry   : ((a × b) → c) → (a → b → c)
```

The α and γ functions defined above, however, are recursive: any (additional) arguments are recursively (un)curried. Although the function types we have encountered so far (arising from addition and conditionals in our source language) are first order, this will no longer be the case once we add lambda abstractions.

The types of the α and γ functions from Meijer's thesis have been generalized, inspired by recent work on categorical compiler calculation by Elliott (2020). Where Meijer's original definition did not account for (arbitrary) stacks, Elliott states that 'the essence of stack computation' is captured by the following type:

$$\mathsf{a} \to \mathsf{b} \;\cong\; \forall \sigma \to (\mathsf{a} \times \sigma) \to (\mathsf{b} \times \sigma)$$

By repeatedly uncurrying and accounting for the ambient stack, we arrive at the definition of α and γ above.

Finally, we assert that these two functions are mutual inverses:

$$\begin{aligned}
\alpha\gamma\text{-inv} &: \forall\, \mathsf{a} \to (\mathsf{x} : [\![\, \mathsf{a} \,]\!]) \;\to \mathsf{x} \equiv \gamma\; \mathsf{a}\; (\alpha\; \mathsf{a}\; \mathsf{x}) \\
\gamma\alpha\text{-inv} &: \forall\, \mathsf{a} \to (\mathsf{x} : [\![\, \mathsf{a} \,]\!]_{\mathsf{s}}) \to \alpha\; \mathsf{a}\; (\gamma\; \mathsf{a}\; \mathsf{x}) \equiv \mathsf{x}
\end{aligned}$$

Proving these equalities directly in Agda, however, is not possible. In particular, they rely on both function extensionality and parametricity, both of which are admissible but not provable in Agda. We can complete this proof, provided we add these as postulates. In what follows, we will leave the arguments to both α and γ implicit, as these are (usually) inferred by Agda automatically.

2.3 Calculating a stack-based interpreter

The problem of finding a stack-based interpreter for evaluating our language boils down to finding a function mapping expressions to their ‘stack machine’ semantics. That is, we would like to uncover a function with the following type:

```
stack-eval : Expr a → ⟦ a ⟧s
```

Of course, not any such function will do. In particular, it should satisfy the following property:

```
correctness : ∀ (e : Expr a) (ξ : args a σ) → α (eval e) ξ ≡ stack-eval e ξ
```

This specification suffices to determine the definition of our next interpreter. The calculation proceeds by induction on the expression, unfolding the definitions of eval and α. We will present each case individually; the derivations we provide have been checked by Agda – but we have replaced the proofs terms by comments explaining each step. In the final step of each calculation, we can read off the definition of the stack-eval function for that particular case.

Constant values. The case for values is entirely straightforward:

```
correctness (val x) ξ = begin
  α (eval (val x)) ξ
    ≡⟨ definition of eval ⟩
  α x ξ
    ≡⟨ definition of α ⟩
  x , ξ
    ≡⟨ use as definition ⟩
  stack-eval (val x) ξ
  □
```

Unsurprisingly, the stack-based evaluation of a value simply pushes the value on top of the stack. After each of our calculations, we define a separate function handling that constructor – which we will reuse later when calculating the corresponding compiler. In this case, the calculation above gives rise to the following function that pushes its argument on to the input stack:

```
push : ℕ → ⟦ nat ⟧s
push x ξ = (x , ξ)
```

Addition. The case for addition is not much harder.

```
correctness plus (x , y , ξ) = begin
  α (eval plus) (x , y , ξ)
    ≡⟨ definition of eval ⟩
  α (_+_) (x , y , ξ)
    ≡⟨ definition of α ⟩
  γ x + γ y , ξ
    ≡⟨ use as definition ⟩
  stack-eval (plus) (x , y , ξ)
  □
```

One advantage of defining addition as a leaf in the abstract tree of our expression language, rather than a node with two sub-expressions, is that the calculation of addition does not require induction, but follows trivially: we only need to unfold the definitions of eval and α. The corresponding operation for our stack-based interpreter adds the numbers on the top of the stack:

```
add : ⟦ nat ⇒ nat ⇒ nat ⟧s
add (x , y , ξ) = (γ x + γ y , ξ)
```

Conditionals. The case for conditional statements is a bit more complicated. Here Agda sometimes struggles to instantiate the (implicitly quantified) type of the two conditional branches. Furthermore, the calculation uses two lemmas to apply a function and pass an argument to both branches.

```
correctness if (c , t , e , ξ) = begin
  α (eval if) (c , t , e , ξ)
    ≡⟨ definition of eval ⟩
  α (if_then_else_) (c , t , e , ξ)
    ≡⟨ distribution over if ⟩
  (if γ c then α (γ t) else α (γ e)) ξ
    ≡⟨ by γα-inv ⟩
  (if γ c then t else e) ξ
    ≡⟨ distribution over if ⟩
  (if γ c then t ξ else e ξ)
    ≡⟨ use as definition ⟩
  stack-eval if (c , t , e , ξ)
  □
```

From the last line, we can read off the conditional operation for our target interpreter:

```
cond : ⟦ bool ⇒ a ⇒ a ⇒ a ⟧s
cond (c , t , e , ξ) = if γ c then t ξ else e ξ
```

Application. The case for applications is arguably the most interesting: it is here that we need to use our induction hypotheses.

```
correctness (app t₁ t₂) ξ = begin
  α (eval (app t₁ t₂)) ξ
    ≡⟨ definition of eval ⟩
  α ((eval t₁) (eval t₂)) ξ
    ≡⟨ by αγ-inv ⟩
  α ((eval t₁) (γ (α (eval t₂)))) ξ
    ≡⟨ induction hypothesis ⟩
  α ((eval t₁) (γ (stack-eval t₂))) ξ
    ≡⟨ definition of α ⟩
  α (eval t₁) (stack-eval t₂ , ξ)
    ≡⟨ induction hypothesis ⟩
  (stack-eval t₁) (stack-eval t₂ , ξ)
    ≡⟨ use as definition ⟩
  stack-eval (app t₁ t₂) ξ
  □
```

The derivation proceeds by unfolding the definition of eval. There is one creative step: to apply our induction hypothesis, we introduce a (spurious) function application of $\gamma \circ \alpha$ on the second argument of the application constructor. The remainder of the calculation then proceeds immediately. Generalizing the last line of our calculation, we read off the following application operation for stack-based computations:

$$\begin{array}{l} \mathsf{apply} : [\![\mathsf{a} \Rightarrow \mathsf{b}]\!]_{\mathsf{s}} \to [\![\mathsf{a}]\!]_{\mathsf{s}} \to [\![\mathsf{b}]\!]_{\mathsf{s}} \\ \mathsf{apply}\, \mathsf{f}\, \mathsf{x}\, \xi = \mathsf{f}\, (\mathsf{x}, \xi) \end{array}$$

This exemplifies stack-based computations, where function application amounts to pushing an argument on to a stack, before executing the function itself.

Collecting all the definitions above yields the following stack-based evaluator:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{stack\text{-}eval} : \mathsf{Expr}\ \mathsf{a} \to [\![\mathsf{a}]\!]_{\mathsf{s}}} \\ \mathsf{stack\text{-}eval}\, (\mathsf{val}\, \mathsf{x}) & = & \mathsf{push}\, \mathsf{x} \\ \mathsf{stack\text{-}eval}\, \mathsf{plus} & = & \mathsf{add} \\ \mathsf{stack\text{-}eval}\, \mathsf{if} & = & \mathsf{cond} \\ \mathsf{stack\text{-}eval}\, (\mathsf{app}\, \mathsf{t}_1\, \mathsf{t}_2) & = & \mathsf{apply}\, (\mathsf{stack\text{-}eval}\, \mathsf{t}_1)\, (\mathsf{stack\text{-}eval}\, \mathsf{t}_2) \end{array}$$

This is a slight variation of the stack machine given by McKinna & Wright (2006). There are several minor differences: their machine also uses a no-op operation; the conditional operation presented here is not recursive – but rather the app constructor is the only way to assemble larger expressions.

2.4 *Transforming to continuation passing style*

Although we have calculated a stack-based interpreter, we have not yet derived a *compiler*. To do so requires two further steps. First, we calculate a new version of our interpreter, written in continuation passing style, thereby fixing the order of evaluation of application nodes. In the next section, we will derive our compiler from this version. These two steps will produce a *linear* sequence of instructions for a stack machine, in contrast to the stack-based evaluator from the previous section.

The type of the CPS-transformed version of our stack-based interpreter should take an extra continuation as its argument:

$$\mathsf{stack\text{-}cps} : \mathsf{Expr}\ \mathsf{a} \to ([\![\mathsf{a}]\!]_{\mathsf{s}} \to \mathsf{C}) \to \mathsf{C}$$

Once again, not all such functions will do. In particular, the next version of our interpreter should satisfy the following specification:

$$\mathsf{stack\text{-}cps\text{-}correct} : (\mathsf{e} : \mathsf{Expr}\ \mathsf{a})\, (\mathsf{k} : [\![\mathsf{a}]\!]_{\mathsf{s}} \to \mathsf{C}) \to \mathsf{k}\, (\mathsf{stack\text{-}eval}\, \mathsf{e}) \equiv \mathsf{stack\text{-}cps}\, \mathsf{e}\, \mathsf{k}$$

The calculation itself of this interpreter is largely mechanical. We illustrate the calculation using two illustrative cases: constants and applications. In the case for constants, the ‘calculation’ merely unfolds the definition of stack-eval:

$$\begin{array}{l} \mathsf{stack\text{-}cps\text{-}correct}\, (\mathsf{val}\, \mathsf{x})\, \mathsf{k} = \\ \quad \mathsf{k}\, (\mathsf{stack\text{-}eval}\, (\mathsf{val}\, \mathsf{x})) \\ \qquad \equiv\langle\ \text{definition of stack-eval}\ \rangle \\ \quad \mathsf{k}\, (\mathsf{push}\, \mathsf{x}) \\ \qquad \equiv\langle\ \text{use as definition}\ \rangle \\ \quad \mathsf{stack\text{-}cps}\, (\mathsf{val}\, \mathsf{x})\, \mathsf{k} \\ \quad \square \end{array}$$

Just as we saw previously, it will be useful to introduce a separate auxiliary function for each case of the CPS-transformed stack-based interpreter. In the case for constants, this function applies the continuation to the corresponding stack operator:

$$\begin{aligned}&\mathsf{push\text{-}c} : \mathbb{N} \to (\mathsf{k} : [\![\mathsf{nat}]\!]_s \to \mathsf{C}) \to \mathsf{C}\\ &\mathsf{push\text{-}c\ n\ k} = \mathsf{k\ (push\ n)}\end{aligned}$$

Similar immediate calculations for addition and conditionals immediately give rise to two other functions for our next interpreter:

$$\begin{aligned}&\mathsf{add\text{-}c} : ([\![(\mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat})]\!]_s \to \mathsf{C}) \to \mathsf{C}\\ &\mathsf{add\text{-}c\ k} = \mathsf{k\ add}\end{aligned}$$

$$\begin{aligned}&\mathsf{cond\text{-}c} : ([\![\mathsf{bool} \Rightarrow \mathsf{a} \Rightarrow \mathsf{a} \Rightarrow \mathsf{a}]\!]_s \to \mathsf{C}) \to \mathsf{C}\\ &\mathsf{cond\text{-}c\ k} = \mathsf{k\ cond}\end{aligned}$$

The more interesting case is that for applications. After unfolding the definition of stack-eval, it is not immediately obvious how to apply our induction hypotheses. By introducing an additional step swapping the arguments of apply, we lift out the stack-eval e_1, bringing the expression into the desired form.

$$\begin{aligned}&\mathsf{stack\text{-}cps\text{-}correct\ (app\ e_1\ e_2)\ k} =\\ &\quad \mathsf{k\ (stack\text{-}eval\ (app\ e_1\ e_2))}\\ &\qquad \equiv\langle\ \text{definition of stack-eval}\ \rangle\\ &\quad \mathsf{k\ (apply\ (stack\text{-}eval\ e_1)\ (stack\text{-}eval\ e_2))}\\ &\qquad \equiv\langle\ \text{definition of swap and function composition}\ \rangle\\ &\quad \mathsf{(k \circ swap\ apply\ (stack\text{-}eval\ e_2))\ (stack\text{-}eval\ e_1)}\\ &\qquad \equiv\langle\ \text{induction hypothesis}\ \rangle\\ &\quad \mathsf{stack\text{-}cps\ e_1\ ((k \circ swap\ apply)\ (stack\text{-}eval\ e_2))}\\ &\qquad \equiv\langle\ \text{induction hypothesis}\ \rangle\\ &\quad \mathsf{stack\text{-}cps\ e_1\ (stack\text{-}cps\ e_2\ (\lambda\, x\, f \to k\ (apply\ f\ x)))}\\ &\qquad \equiv\langle\ \text{use as definition}\ \rangle\\ &\quad \mathsf{stack\text{-}cps\ (app\ e_1\ e_2)\ k}\\ &\quad \square\end{aligned}$$

The rest of the calculation follows directly. We read off the required auxiliary function from the last step:

$$\begin{aligned}&\mathsf{apply\text{-}c} : (\mathsf{k} : [\![\mathsf{b}]\!]_s \to \mathsf{C}) \to ([\![\mathsf{a}]\!]_s \to [\![\mathsf{a} \Rightarrow \mathsf{b}]\!]_s \to \mathsf{C})\\ &\mathsf{apply\text{-}c\ k\ x\ f} = \mathsf{k\ (apply\ f\ x)}\end{aligned}$$

In this calculation, we fix the evaluation order. Here we have opted to first evaluate e_1 and then evaluate e_2; alternatively, we could have chosen the reverse order, first applying the induction hypothesis for e_2 before using that for e_1, changing the order in which the arguments to app are evaluated. The observation that CPS-transforming fixes the evaluation order dates back to Reynolds (1972).

To complete the definition of our stack-based evaluator in continuation passing style, we collect all the derived functions in a single definition.

```
stack-cps : Expr a → (⟦ a ⟧s → C) → C
stack-cps (val n)       = push-c
stack-cps plus          = add-c
stack-cps if            = cond-c
stack-cps (app e₁ e₂)   = stack-cps e₁ ∘ stack-cps e₂ ∘ apply-c
```

We can recover our original stack-based evaluator by passing the identity function as the initial continuation.

```
stack-cps-eval : Expr a → ⟦ a ⟧s
stack-cps-eval e = stack-cps e (λ x → x)
```

The next two subsections describe different further transformations on our stack-based interpreter in continuation passing style. In the next section, we will derive a compiler, essentially introducing an intermediate data structure representing the code for the stack-cps evaluator. The final section defunctionalizes this evaluator, producing a tail-recursive abstract machine (Reynolds, 1972; Ager *et al.*, 2003; McBride, 2008).

2.5 Calculating a compiler

To recover the *compiler* proposed by McKinna and Wright, we create an intermediate datatype for our instruction set. This reifies the instructions for the evaluator presented in the previous section as a separate datatype. The resulting transformation is a form of 'reforestation', inverse to deforestation (Wadler, 1988), that introduces an intermediate datatype between the syntax and stack machine semantics. In this final step, we introduce a data type for our instruction set:

```
Code : U → Set
```

The constructors of this data type are read off from our previous evaluator; we will calculate a new function that executes these instructions:

```
exec : Code a → ⟦ a ⟧s
```

The definition of exec will be calculated from the following specification:

```
compiler-correctness : ∀ (e : Expr a) (c : Code (a ⇒ b)) (ξ : args b σ) →
  exec (compile-acc e c) ξ ≡ exec c (stack-eval e , ξ)
```

Here the code c plays the role of the continuation that we saw previously. Intuitively, this property states that executing compiled code should produce the same result as evaluating an expression directly using our stack-based evaluator.

The Code data type has one constructor for each of the functions passed to the stack-cps evaluator from the previous section:

```
data Code : U → Set where
  PUSH  : ℕ → Code (nat ⇒ c) → Code c
  ADD   : Code ((nat ⇒ nat ⇒ nat) ⇒ c) → Code c
  COND  : Code ((bool ⇒ a ⇒ a ⇒ a) ⇒ c) → Code c
  APP   : Code (b ⇒ c) → Code (a ⇒ (a ⇒ b) ⇒ c)
  HALT  : Code (c ⇒ c)
```

The type of each constructor is determined by the corresponding branch in the machine-cps function. Compare, for example, the type of push-c function with the corresponding PUSH constructor:

$$\mathsf{push\text{-}c} \;:\; \mathbb{N} \to (\mathsf{k} \;:\; [\![\, \mathsf{nat} \,]\!]_{\mathsf{s}} \to \mathsf{C}) \to \mathsf{C}$$

The continuation argument has become the remaining code to execute. A more complicated example is that for applications, giving rise to the APP constructor:

$$\mathsf{apply\text{-}c} \;:\; (\mathsf{k} \;:\; [\![\, \mathsf{b} \,]\!]_{\mathsf{s}} \to \mathsf{C}) \to ([\![\, \mathsf{a} \,]\!]_{\mathsf{s}} \to [\![\, \mathsf{a} \Rightarrow \mathsf{b} \,]\!]_{\mathsf{s}} \to \mathsf{C})$$

Once again, the continuation argument determines the type of the remaining code; any additional arguments, such as the values of type a and $\mathsf{a} \Rightarrow \mathsf{b}$, will now be found on the stack rather than passed explicitly. Finally, the HALT instruction corresponds to the identity function used as the initial continuation to kick off stack-cps.

The compilation itself simply maps each language construct to the corresponding stack machine operation, using an additional Code argument rather than the continuation. The only interesting case is that for applications, where the codes of both sub-expressions are compiled one after the other.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathsf{compile\text{-}acc} \;:\; \mathsf{Expr\;a} \to \mathsf{Code\;(a \Rightarrow b)} \to \mathsf{Code\;b}} \\
\mathsf{compile\text{-}acc\;(val\;x)} & = & \mathsf{PUSH\;x} \\
\mathsf{compile\text{-}acc\;plus} & = & \mathsf{ADD} \\
\mathsf{compile\text{-}acc\;if} & = & \mathsf{COND} \\
\mathsf{compile\text{-}acc\;(app\;e_1\;e_2)} & = & \mathsf{compile\text{-}acc\;e_1} \circ \mathsf{compile\text{-}acc\;e_2} \circ \mathsf{APP}
\end{array}$$

Once again, the top-level compiler starts with an initially empty code continuation:

$$\begin{array}{l}
\mathsf{compile} \;:\; \mathsf{Expr\;a} \to \mathsf{Code\;a} \\
\mathsf{compile\;e} \;=\; \mathsf{compile\text{-}acc\;e\;HALT}
\end{array}$$

To prove that our compiler is correct, we calculate the execution semantics of our code. There is little creativity involved at this point: each target instruction should call the corresponding operation that we have derived previously. The desired type of the corresponding exec maps code to stack-machine operations:

$$\mathsf{exec} \;:\; \mathsf{Code\;a} \to [\![\, \mathsf{a} \,]\!]_{\mathsf{s}}$$

The specification given at the beginning of this section is more general than necessary. Rather than reason about the top-level compile function directly, we formulate a more general property in terms of compile-acc. All that remains is to calculate the exec function that satisfies this specification. To illustrate the calculation, we cover the two cases for constants and application.

The case for constants follows immediately after expanding definitions:

$$\begin{array}{l}
\mathsf{compiler\text{-}correctness\;(val\;x)\;c}\;\xi \;=\; \\
\quad \mathsf{exec\;(compile\text{-}acc\;(val\;x)\;c)}\;\xi \\
\qquad \equiv\langle\; \mathsf{definition\;of\;compile\text{-}acc} \;\rangle \\
\quad \mathsf{exec\;(PUSH\;x\;c)}\;\xi \\
\qquad \equiv\langle\; \text{use as definition} \;\rangle \\
\quad \mathsf{exec\;c\;(push\;x\;,}\;\xi) \\
\qquad \equiv\langle\; \mathsf{definition\;of\;stack\text{-}eval} \;\rangle \\
\quad \mathsf{exec\;c\;(stack\text{-}eval\;(val\;x)\;,}\;\xi) \\
\quad \square
\end{array}$$

Note that we expand definitions on both sides of the equation: unfolding the definition stack-eval on the right and and compile-acc on the left.

The case for applications is not much harder. After unfolding definitions and applying our induction hypotheses, the desired definition reveals itself immediately:

```
compiler-correctness (app e₁ e₂) c ξ  =
  exec (compile-acc (app e₁ e₂) c) ξ
    ≡⟨ definition of compile-acc ⟩
  exec (compile-acc e₁ (compile-acc e₂ (APP c))) ξ
    ≡⟨ induction hypothesis ⟩
  exec (compile-acc e₂ (APP c)) (stack-eval e₁ , ξ)
    ≡⟨ induction hypothesis ⟩
  exec (APP c) (stack-eval e₂ , stack-eval e₁ , ξ)
    ≡⟨ use as definition ⟩
  exec c (apply (stack-eval e₁) (stack-eval e₂) , ξ)
    ≡⟨ definition of stack-eval ⟩
  exec c (stack-eval (app e₁ e₂) , ξ)
  □
```

Assembling these calculations into a single definition shows how executing code maps every constructor to the operation derived for our stack machine:

```
exec : Code a → ⟦ a ⟧s
exec (PUSH x c) ξ       = exec c (push x , ξ)
exec (ADD c) ξ          = exec c (add , ξ)
exec (COND c) ξ         = exec c (cond , ξ)
exec (APP c) (x , f , ξ) = exec c (apply f x , ξ)
exec HALT (x , ξ)       = x ξ
```

Finally, instantiating the code argument to HALT, lets us formulate our main correctness result, relating compiled code to the original direct evaluator from Section 2.1:

```
main-compiler-correctness : (e : Expr a) (ξ : args a σ) →
  exec (compile e) ξ ≡ α (eval e) ξ
```

This concludes the derivation of the correct compiler in several steps. Starting from an intrinsically typed interpreter, via a direct stack-based interpreter and one in continuation passing style, to our final compiler.

Higher-order stacks. One drawback of this construction is that the stacks store *higher-order values* – as opposed to the first-order stacks that appear in typical compiler calculations. The reason for this is twofold. First, the expression language may contain partial applications (of additions or conditionals). These partial applications must be stored on the stack during evaluation; as a result, the stack contains functional values. Second, the arguments of a function in our ‘stack semantics’ are themselves – recall the following clause of our stack semantics:

```
args (a ⇒ b) σ  = ⟦ a ⟧s × args b σ
```

To restrict ourselves to first-order data on the stack, we would need to replace $[\![\mathsf{a}]\!]_\mathsf{s}$ in this definition with a (first-order) representation of the argument type. In the expression language, we have seen so far, partial applications may be defunctionalized to their first-order representation; in the lambda calculus presented in the next section, we would need to represent such higher-order arguments by the corresponding closure. As different languages require different first-order data representations, we have used the definition above – at the expense of having a machine that stores higher-order values on the stack.

Calculating compilers. This section has derived the semantics of the virtual machine, namely the exec function, rather than the *compiler* we set out to calculate. Arguably, the compiler is the least interesting part of our calculation: the target language's constructors and their types are determined by functions calculated in the previous section. The compiler itself merely maps each language construct to its corresponding target operation. To also calculate the compiler, we need to specify its intended behaviour. To do so, we begin by defining the function that maps a series of instructions to their corresponding semantics, that is, the composition of the corresponding CPS-transformed function from the previous section:

```
toCPS : Code b → (⟦ b ⟧s → A) → A
```

We now specify the intended behaviour of the compiler in terms of the stack-based CPS-transformed evaluator derived in the previous section:

```
compiler-spec : (e : Expr a) (c : Code (a ⇒ b)) (k : ⟦ b ⟧s → A) →
  toCPS (compile-acc e c) k ≡ stack-cps e (toCPS c ∘ apply-c k)
```

In particular, combining the specifications from all the previous sections establishes the familiar correctness statement:

```
toCPS (compile e) id ξ ≡ α (eval e) ξ
```

2.6 A tail recursive machine

The previous section demonstrated how to derive a compiler from the stack-based interpreter in continuation passing style. This introduced a separate data type for code, where each constructor is a placeholders for a stack machine operation. In this section, we demonstrate how *defunctionalizing* the continuations leads to a tail recursive abstract machine – a transformation that has been explored extensively in the literature (Reynolds, 1972; Ager *et al.*, 2003; McBride, 2008), albeit rarely in the intrinsically typed setting. Even though we have already derived our correct compiler, this establishes the connection with similar techniques commonly found in the literature. This section serves to highlight the differences between the calculation of compiler and abstract machines.

Defunctionalizing the intrinsically typed stack-based evaluator in continuation passing style, defined in Section 2.4, is a bit of a puzzle. Even if similar derivations have been done before in the simply typed setting, ensuring the type indices line up nicely in this development requires some thought. To derive our final, tail recursive machine requires three definitions:

```
data K : U → Set
continue : K (a ⇒ b) → ⟦ a ⟧s → ⟦ b ⟧s
exec-cps : Expr a → K (a ⇒ b) → ⟦ b ⟧s
```

The datatype K represents our defunctionalized continuations; the function continue applies such a defunctionalized continuation to an argument value; finally, the exec-cps function corresponds to our tail recursive abstract machine. Once these types are in place, the definition of the remaining ingredients follows readily enough.

The CPS-transformed machine uses three continuations: the identity function and one for each argument of an application. Correspondingly, the datatype representing the defunctionalized continuations has three constructors:

```
data K where
  ARG  : Expr a → K (b ⇒ c) → K ((a ⇒ b) ⇒ c)
  FUN  : ⟦ a ⇒ b ⟧s → K (b ⇒ c) → K (a ⇒ c)
  STOP : K (a ⇒ a)
```

The names of the constructors are inspired by existing work on deriving abstract machines by Ager *et al.* (2003). The ARG constructor stores an (unevaluated) function argument; the FUN constructor applies a function to the top of the stack. Last but not least, the STOP constructor corresponds to the identity function, used as the initial continuation.

To assign meaning to these defunctionalized continuations, we define the continue function – mapping each value in K to the appropriate function:

```
continue STOP      v = v
continue (ARG e k) f = exec-cps e (FUN f k)
continue (FUN f k) v = continue k (apply f v)
```

Each of these cases corresponds to one of the continuations used previously. The STOP constructor halts the recursion and returns the value that has been computed. The ARG constructor evaluates the second argument of an application, once the first has been fully evaluated. The FUN constructor applies the functional value stored to the value on the top of the stack.

To complete the definition of our abstract machine, we define a tail-recursive function that uses our K data type, rather than the continuations used previously.

```
exec-cps (val x)       k = continue k (push x)
exec-cps plus          k = continue k (add)
exec-cps if            k = continue k cond
exec-cps (app e₁ e₂) k = exec-cps e₁ (ARG e₂ k)

abstract-machine   : Expr a → ⟦ a ⟧s
abstract-machine e = exec-cps e STOP
```

In the base cases, we reuse the instructions from the stack-based evaluator that we calculated previously and call the continue function to apply the continuation. In the case for applications, we store the second expression in our defunctionalized continuation and continue evaluating the first argument. The top-level abstract machine executes an expression, starting with an empty continuation.

Unfortunately, these definitions do not pass Agda's termination check. The problematic call is the ARG branch of the continue function: here we recurse over the expression e with a larger defunctionalized continuation. Although this expression e was originally a structurally smaller sub-expression, arising from the last clause of the exec-cps function, Agda's termination checker is unable to see that the recursion will terminate. Using a suitable well-founded relation, we could prove termination (Tomé Cortiñas & Swierstra, 2018), but we will refrain from doing so here.

Under the assumption that these functions terminate, we can prove following correctness result, relating the stack machine we calculated with its CPS-transformed and defunctionalized tail recursive variant:

```
abstract-machine-correct : (e : Expr a) (k : K (a ⇒ b)) (ξ : args b σ) →
  exec-cps e k ξ ≡ continue k (stack-eval e) ξ
```

The proof follows immediately by induction on the argument expression. Fixing the definitions of K and continue, we should be able to calculate the definition of exec-cps, just as we calculated the execution of code in the previous section.

3 Lambda calculus

The previous section calculated a compiler for a first-order language. Can we use the same techniques for a higher-order language? This section explores how to extend our specification and the initial steps of calculations to the simply typed lambda calculus. To redo our calculation, we first need to define an intrinsically typed source language:

```
data Term : List U → U → Set where
  lam : Term (a :: Γ) b → Term Γ (a ⇒ b)
  app : Term Γ (a ⇒ b) → Term Γ a → Term Γ b
  var : Ref Γ a → Term Γ a
  val : ℕ → Term Γ nat
```

The Term data type is a well-known representation of the simply typed lambda terms as an indexed family. We will typically use the variable Γ for contexts – or lists of types – and a and b for the types themselves. The applications and constants are familiar; this definition introduces lambdas and variables. Each lam constructor adds a new variable to the context. We can refer to these variables using the *well-typed De Bruijn* indices:

```
data Ref : List U → U → Set where
  top : Ref (a :: Γ) a
  pop : Ref Γ a → Ref (b :: Γ) a
```

You might read Ref Γ σ as a (proof-relevant) witness showing that the type σ occurs in the context Γ. The definition is analogous to simple De Bruijn indices, where top and pop play the roles of zero and successor, respectively.

Evaluating lambda terms requires an environment. Such environments consist of a heterogeneous list of values, indexed by a context describing the type of values it stores.

```
data Env (el : U → Set) : List U → Set where
  nil  : Env el []
  _·_ : el a → Env el Γ → Env el (a :: Γ)
```

As we will use the same environment to store different interpretations of our type universe, we add an additional parameter, el : U → Set. The key advantage of indexing both our references and environments with the same context is that we can now define the following total and type-safe lookup function:

```
lookup : Ref Γ a → Env el Γ → el a
lookup top     (x · env) = x
lookup (pop i) (x · env) = lookup i env
```

With these definitions in place, we complete the familiar definition of the evaluator for the simply typed lambda calculus:

```
⟦Env⟧ = Env ⟦_⟧
eval : Term Γ a → (⟦Env⟧ Γ → ⟦ a ⟧)
eval (lam t)       env = λ v → eval t (v · env)
eval (app t₁ t₂) env = (eval t₁ env) (eval t₂ env)
eval (var x)       env = lookup x env
eval (val c)       env = c
```

We introduce the type synonym ⟦Env⟧ for environments storing values. This eval function maps the lam constructor to Agda's lambdas; the app constructor is mapped to Agda's application. By passing an environment storing values for all the term's variables, the evaluation of variables becomes a call to the lookup function. This environment is extended as we go under a lambda.

This completes our brief review of the evaluator for the simply typed lambda calculus – but how can we *calculate* a stack-based interpreter?

4 A stack-based evaluator for the lambda calculus

To calculate a stack-based evaluator, we begin by introducing a variant of our environments using the 'stack semantics' for our types, rather than the 'direct semantics' used in our evaluator:

```
⟦Env⟧ₛ : List U → Set
⟦Env⟧ₛ = Env ⟦_⟧ₛ
```

We can use this variation of environments to specify the problem of deriving our stack-based evaluator. Our aim is to define an evaluator with the following type:

```
stack-eval : Term Γ a → (⟦Env⟧ₛ Γ → ⟦ a ⟧ₛ)
```

The type is almost the same as the type of our evaluator, only it interprets contexts and types using our stack semantics. To ensure that our stack-based evaluator is correct, it should satisfy the following property:

```
correctness : ∀ (t : Term Γ a) (env : ⟦Env⟧ Γ) (ξ : args a σ) →
  α (eval t env) ξ ≡ stack-eval t (mapEnv α env) ξ
```

In words, evaluating a term and executing it using the (currently undefined) stack-based evaluator should produce the same result. Note that the types of our object language, their stack-based denotation, and α and γ functions are all the same as we saw previously in the calculation of our expression compiler.

The only difference with our previous specification is the presence of the additional environment – passed to the evaluation function on the left-hand side of the equality. The stack machine we have set out to calculate, however, takes a different type environment as its argument, namely one of of type $[\![\mathsf{Env}]\!]_s\ \Gamma$ rather than $[\![\mathsf{Env}]\!]\ \Gamma$. We cannot reuse our environment directly, but can use our abstraction function α to map from one to the other:

```
mapEnv : {el₁ el₂ : U → Set} → (∀ {a} → el₁ a → el₂ a) → Env el₁ Γ → Env el₂ Γ
mapEnv f nil         = nil
mapEnv f (x · env)   = f x · mapEnv f env
```

To uncover the definition of the stack-eval function, we perform induction on the argument term. In each case, we rewrite the left-hand side of the equation above, until it no longer refers to the eval function. In this way, we can read off the required definition for stack-eval for each of the four constructors of the Term data type. Two of the cases are almost identical to those from our previous calculation, namely, those for applications and constants: the only difference being the additional environment argument. For the sake of brevity, we will only present the two new cases for variables and lambda abstractions.

Variables. To handle variables, we relate the two environments on either side of our specification, env and map α env. To do so, we rely on the natural property relating map and lookup, stating that for all functions f, environments env and references i we have:

```
f (lookup i env) = lookup i (map f env)
```

Using this property, our calculation proceeds as follows.

```
correctness (var i) env ξ =
  α (eval (var i) env) ξ
    ≡⟨ definition of eval ⟩
  α (lookup i env) ξ
    ≡⟨ lookup-map property ⟩
  lookup i (mapEnv α env) ξ
    ≡⟨ use as definition ⟩
  stack-eval (var i) (mapEnv α env) ξ
  □
```

In the final step, we can read off the required case for variables:

```
stack-eval (var i) = lookup i
```

Unsurprisingly, variables lookup the corresponding value from the environment.

Abstraction. The last remaining constructor is that for lambda abstraction. Even though lambda abstractions introduce higher-order functions into our object language, the techniques we have seen so far suffice to calculate the corresponding stack machine operations.

$$
\begin{array}{l}
\mathsf{correctness}\,(\mathsf{lam}\ \mathsf{t})\ \mathsf{env}\ (\mathsf{arg}\ ,\xi) =\\
\quad \alpha\ (\mathsf{eval}\ (\mathsf{lam}\ \mathsf{t})\ \mathsf{env})\ (\mathsf{arg}\ ,\xi)\\
\qquad \equiv\langle\ \text{definition of eval}\ \rangle\\
\quad \alpha\ (\lambda\,\mathsf{x} \to \mathsf{eval}\ \mathsf{t}\ (\mathsf{x}\cdot\mathsf{env}))\ (\mathsf{arg}\ ,\xi)\\
\qquad \equiv\langle\ \text{definition of } \alpha \text{ for functions}\ \rangle\\
\quad \alpha\ (\mathsf{eval}\ \mathsf{t}\ (\gamma\ \mathsf{arg}\cdot\mathsf{env}))\ \xi\\
\qquad \equiv\langle\ \text{induction hypothesis}\ \rangle\\
\quad \mathsf{stack\text{-}eval}\ \mathsf{t}\ (\mathsf{mapEnv}\ \alpha\ (\gamma\ \mathsf{arg}\cdot\mathsf{env}))\ \xi\\
\qquad \equiv\langle\ \text{definition of mapEnv}\ \rangle\\
\quad \mathsf{stack\text{-}eval}\ \mathsf{t}\ (\alpha\ (\gamma\ \ \mathsf{arg})\cdot\mathsf{mapEnv}\ \alpha\ \mathsf{env})\ \xi\\
\qquad \equiv\langle\ \text{by } \gamma\alpha\text{-inv}\ \rangle\\
\quad \mathsf{stack\text{-}eval}\ \mathsf{t}\ (\mathsf{arg}\cdot\mathsf{mapEnv}\ \alpha\ \mathsf{env})\ \xi\\
\qquad \equiv\langle\ \text{use as definition}\ \rangle\\
\quad \mathsf{stack\text{-}eval}\ (\mathsf{lam}\ \mathsf{t})\ (\mathsf{mapEnv}\ \alpha\ \mathsf{env})\ (\mathsf{arg}\ ,\xi)\\
\quad \square
\end{array}
$$

The derivation for lambda expressions is almost embarrassingly simple. After unfolding the definitions of eval and α, we can immediately use our induction hypothesis. After this step, the remaining calculation unfolds definitions and applies the property relating α and γ to rewrite the term into the desired form.

We can now collect all these clauses in single definition, describing a stack-based evaluator for lambda terms. Before doing so, however, we define auxiliary operations for each of the definitions we have calculated:

$$
\begin{array}{l}
\mathsf{apply} : ([\![\mathsf{Env}]\!]_s\ \Gamma \to [\![\ \mathsf{a} \Rightarrow \mathsf{b}\ ]\!]_s) \to ([\![\mathsf{Env}]\!]_s\ \Gamma \to [\![\ \mathsf{a}\ ]\!]_s) \to ([\![\mathsf{Env}]\!]_s\ \Gamma \to [\![\ \mathsf{b}\ ]\!]_s)\\
\mathsf{apply}\ \mathsf{f}\ \mathsf{arg}\ \mathsf{env}\ \xi\ = \mathsf{f}\ \mathsf{env}\ (\mathsf{arg}\ \mathsf{env}\ ,\xi)\\
\mathsf{enter} : ([\![\mathsf{Env}]\!]_s\ (\mathsf{a} :: \Gamma) \to [\![\ \mathsf{b}\ ]\!]_s) \to ([\![\mathsf{Env}]\!]_s\ \Gamma \to [\![\ \mathsf{a} \Rightarrow \mathsf{b}\ ]\!]_s)\\
\mathsf{enter}\ \mathsf{f}\ \mathsf{env}\ (\mathsf{arg}\ ,\xi)\ = \mathsf{f}\ (\mathsf{arg}\cdot\mathsf{env})\ \xi\\
\mathsf{push} : \mathbb{N} \to [\![\mathsf{Env}]\!]_s\ \Gamma \to [\![\ \mathsf{nat}\ ]\!]_s\\
\mathsf{push}\ \mathsf{c}\ \mathsf{env}\ \xi\ = (\mathsf{c}\ ,\xi)
\end{array}
$$

Finally, we define our evaluator by mapping each constructor to the operation we have calculated:

$$
\begin{array}{lcl}
\mathsf{stack\text{-}eval}\ (\mathsf{val}\ \mathsf{c}) & = & \mathsf{push}\ \mathsf{c}\\
\mathsf{stack\text{-}eval}\ (\mathsf{lam}\ \mathsf{t}) & = & \mathsf{enter}\ (\mathsf{stack\text{-}eval}\ \mathsf{t})\\
\mathsf{stack\text{-}eval}\ (\mathsf{app}\ \mathsf{t}_1\ \mathsf{t}_2) & = & \mathsf{apply}\ (\mathsf{stack\text{-}eval}\ \mathsf{t}_1)\ (\mathsf{stack\text{-}eval}\ \mathsf{t}_2)\\
\mathsf{stack\text{-}eval}\ (\mathsf{var}\ \mathsf{x}) & = & \mathsf{lookup}\ \mathsf{x}
\end{array}
$$

To define the corresponding compiler or tail-recursive abstract machine, we would need to write the stack-based evaluator in continuation passing style, before reforesting or defunctionalizing, respectively. Doing so would require a careful treatment of the context information, that changes as we go under binders. Instead of doing so, we change tack and revisit the original work by Meijer (1992) and calculate the compiler optimizations suggested therein.

5 Optimizing environments

Meijer (1992) suggests two optimizations for the stack-based evaluator derived so far:

- We *always* extend the environment when entering the body of a lambda, even if the corresponding variable is unused.
- In the case for applications, the *entire* environment is duplicated and passed to both arguments, even if only a fraction of the environment is needed.

The challenge is to calculate these optimizations, showing they follow naturally from our definitions so far. To achieve this, we need to shift to a different representation of variables and binding: rather than maintain a context of all the variables that *may* be used, we keep track of a subset of this context, tracking all the variables that *have* been used. Before we can make this more precise, we need to introduce several auxiliary definitions for sublists and the functions to manipulate them.

We could keep track of the variables a term uses in a separate context, i.e., a list of types. In the case for applications, we will need to compute their union, combining the variables used on either side of the application; but the union of lists is rather messy to define, especially as we are using positions in each list to variables. Fortunately, we are not interested in *arbitrary* contexts, but rather in *subsets of our context* Γ. We can introduce a separate data type to describe such subsets.

```
data Subset : List U → Set where
  stop : Subset []
  drop : Subset Γ → Subset (a :: Γ)
  keep : Subset Γ → Subset (a :: Γ)
```

The type Subset Γ tells us for each element of Γ whether it should be included in the subset or not. We map such subsets to their corresponding context easily enough.

```
⌊_⌋ : Subset Γ  →  List U
⌊ stop ⌋             = []
⌊ drop Δ ⌋           = ⌊ Δ ⌋
⌊ keep {a = a} Δ ⌋ = a :: ⌊ Δ ⌋
```

We will typically use the variable Δ to refer to such subsets.

To determine which variables to include in an environment, we would like to compute the subset of variables that occur in a given term. That is, we would like to define a function with the following type:

```
variables : Term Γ a  →  Subset Γ
```

Before we can complete this definition, however, we will need to define several auxiliary functions for creating and manipulating subsets.

To compute the union of two subsets, $\Delta_1 \cup \Delta_2$, we simply keep each element that occurs in either Δ_1 or in Δ_2:

```
_∪_ : Subset Γ → Subset Γ → Subset Γ
stop    ∪ stop    = stop
drop xs ∪ drop ys = drop (xs ∪ ys)
drop xs ∪ keep ys = keep (xs ∪ ys)
keep xs ∪ drop ys = keep (xs ∪ ys)
keep xs ∪ keep ys = keep (xs ∪ ys)
```

If we were to work with contexts directly, it is less clear how to compute their union: given a pair of contexts, we would need to determine which variables are shared by both.

The empty subset, ∅, is the unit of this union operator.

```
∅ : {Γ : List U} → Subset Γ
∅ {[]}     = stop
∅ {x :: Γ} = drop ∅
```

We can create a nonempty subset from any given reference in Γ:

```
singleton : Ref Γ a → Subset Γ
singleton top     = keep ∅
singleton (pop p) = drop (singleton p)
```

Here we keep the variable the reference refers to, but drop all others.

Finally, we will need one last operation, used as we go under a binder. The tail operation that returns the shrinks the subset, limiting it to the remaining context:

```
tail : Subset (a :: Γ) → Subset Γ
tail (drop s) = s
tail (keep s) = s
```

Using these four operations, we can easily compute the variables that occur in a given term. Each of the four operations on subsets defined above is used to handle each of the four cases in our term language.

```
variables (lam t)       = tail (variables t)
variables (app t_1 t_2) = (variables t_1) ∪ (variables t_2)
variables (var i)       = singleton i
variables (val c)       = ∅
```

While this definition lets us compute the variables used by a single term, when calculating our optimizing stack-based evaluator, we need to know the variables that occur in *every* subterm. Instead of repeatedly calling variables, we choose to define a variation of the Term data type, where each constructor tracks the set of variables used in its subterms:

```
data CDB (Γ : List U) : Subset Γ → U → Set where
  lam : CDB (a :: Γ) Δ b → CDB Γ (tail Δ) (a ⇒ b)
  app : CDB Γ Δ_1 (a ⇒ b) → CDB Γ Δ_2 a → CDB Γ (Δ_1 ∪ Δ_2) b
  var : (i : Ref Γ a) → CDB Γ (singleton i) a
  val : ℕ → CDB Γ ∅ nat
```

McBride (2018) has dubbed this particular variable representation, modeling the variables used in a term as a subset of all variables in scope, as the *co-de Bruijn* representation; hence the name, CDB.

Note that we can always forget the information about variable occurrences and recover our original Term:

```
forget : CDB Γ Δ a → Term Γ a
forget (lam cdb)           = lam (forget cdb)
forget (app cdb₁ cdb₂)     = app (forget cdb₁) (forget cdb₂)
forget (var i)             = var i
forget (val x)             = val x
```

Note that the inverse operation, producing a pair of subset Δ and co-de Bruijn term of the corresponding type, is not much harder to define using the functions we have defined at the beginning of this section. This shift in variable representation, while useful, is not overly demanding.

We are almost ready to start the calculation of the optimizing version of our stack-based evaluator. Before we can do so, we need one last operation on environments. Given any environment over the context Γ and a subset of Γ, we can compute a new environment, restricting it to contain only those values that occur in the argument subset:

```
restrict : (Δ : Subset Γ) → ⟦Env⟧ₛ Γ → ⟦Env⟧ₛ ⌊ Δ ⌋
restrict stop nil          = nil
restrict (drop Δ) (v · η)  = restrict Δ η
restrict (keep Δ) (v · η)  = v · restrict Δ η
```

This definition is entirely determined by its type signature.

We will now show how to find a variation of our previous stack-based evaluator that operates on CDB terms, but only requires an environment with values for all the variables that actually occur in the term:

```
stack-eval-opt : CDB Γ Δ a → ⟦Env⟧ₛ ⌊ Δ ⌋ → ⟦ a ⟧ₛ
```

This optimizing evaluator should satisfy the following correctness property.

```
correctness-opt : (t : CDB Γ Δ a) → (env : ⟦Env⟧ₛ Γ) (ξ : args a σ) →
  stack-eval (forget t) env ξ ≡ stack-eval-opt t (restrict Δ env) ξ
```

Once again, we will start our derivations on the left-hand side of the equation, rewriting until we have eliminated any reference to the stack-eval function and can read off the desired definition of stack-eval-opt. The cases for constants are unchanged. The case for variables follows naturally, as we only have a single possible value to return. The only non-trivial cases are those for applications and lambda abstractions.

Abstraction – unused variable. The case for abstractions now distinguishes two further cases, depending on whether or not the bound variable occurs or not. When the variable does occur, and our subset starts with a keep constructor, we can unfold definitions and apply our induction hypothesis, thereby fixing the behaviour of stack-eval-opt. The case when the freshly bound variable goes unused (and our subset starts with a drop constructor), however, is more interesting. In that case, we know that the body of the lambda does not use the bound variable. As a result, the environment we pass in must *not* be extended with its value. This intuition is made precise by the following calculation, considering the case where a lambda binds an unused variable:

$$\begin{aligned}
&\mathsf{correctness\text{-}opt}\ (\mathsf{lam}\ \{\Delta = \mathsf{drop}\ \Delta\}\ \mathsf{t})\ \mathsf{env}\ (\mathsf{arg}\ ,\ \xi) = \\
&\quad \mathsf{stack\text{-}eval}\ (\mathsf{forget}\ (\mathsf{lam}\ \{\Delta = \mathsf{drop}\ \Delta\}\ \mathsf{t}))\ \mathsf{env}\ (\mathsf{arg}\ ,\ \xi) \\
&\qquad \equiv\langle\ \text{definition of } \mathsf{forget}\ \rangle \\
&\quad \mathsf{stack\text{-}eval}\ (\mathsf{lam}\ (\mathsf{forget}\ \{\Delta = \mathsf{drop}\ \Delta\}\ \mathsf{t}))\ \mathsf{env}\ (\mathsf{arg}\ ,\ \xi) \\
&\qquad \equiv\langle\ \text{definition of } \mathsf{stack\text{-}eval}\ \rangle \\
&\quad \mathsf{stack\text{-}eval}\ (\mathsf{forget}\ \{\Delta = \mathsf{drop}\ \Delta\}\ \mathsf{t})\ (\mathsf{arg} \cdot \mathsf{env})\ \xi \\
&\qquad \equiv\langle\ \text{induction hypothesis}\ \rangle \\
&\quad \mathsf{stack\text{-}eval\text{-}opt}\ \mathsf{t}\ (\mathsf{restrict}\ (\mathsf{drop}\ \Delta)\ (\mathsf{arg} \cdot \mathsf{env}))\ \xi \\
&\qquad \equiv\langle\ \text{definition of } \mathsf{restrict}\ \rangle \\
&\quad \mathsf{stack\text{-}eval\text{-}opt}\ \mathsf{t}\ (\mathsf{restrict}\ \Delta\ \mathsf{env})\ \xi \\
&\qquad \equiv\langle\ \text{use as definition}\ \rangle \\
&\quad \mathsf{stack\text{-}eval\text{-}opt}\ (\mathsf{lam}\ \{\Delta = \mathsf{drop}\ \Delta\}\ \mathsf{t})\ (\mathsf{restrict}\ \Delta\ \mathsf{env})\ (\mathsf{arg}\ ,\ \xi) \\
&\quad \square
\end{aligned}$$

We combine this function with the case for lambda abstractions as given by the function enter in previous section, leading to the following definition of enter-opt. The second argument of enter-opt corresponds to evaluating the body of the lambda – generalizing stack-eval-opt t in the calculation above. Furthermore, the definition generalizes over any environment – rather than writing the argument environment in the form restrict Δ env explicitly.

$$\begin{aligned}
&\mathsf{enter\text{-}opt} : (\Delta : \mathsf{Subset}\ (\mathsf{a} :: \Gamma)) \rightarrow (\llbracket \mathsf{Env} \rrbracket_{\mathsf{s}}\ \lfloor \Delta \rfloor \rightarrow \llbracket \mathsf{b} \rrbracket_{\mathsf{s}}) \rightarrow \\
&\quad (\llbracket \mathsf{Env} \rrbracket_{\mathsf{s}}\ \lfloor \mathsf{tail}\ \Delta \rfloor \rightarrow \llbracket \mathsf{a} \Rightarrow \mathsf{b} \rrbracket_{\mathsf{s}}) \\
&\mathsf{enter\text{-}opt}\ (\mathsf{keep}\ \Delta)\ \mathsf{f}\ \mathsf{env}\ (\mathsf{arg}\ ,\ \xi) = \mathsf{f}\ (\mathsf{arg} \cdot \mathsf{env})\ \xi \\
&\mathsf{enter\text{-}opt}\ (\mathsf{drop}\ \Delta)\ \mathsf{f}\ \mathsf{env}\ (\mathsf{arg}\ ,\ \xi) = \mathsf{f}\ \mathsf{env}\ \xi
\end{aligned}$$

The first case, associated with the keep constructor, is exactly the same as the enter function we saw previously. The case for the drop constructor discards the argument at the top of the stack, before entering the function body, just as we saw in the calculation above.

Application. In the case for applications, we can perform a similar calculation, unfolding our definitions and applying the induction hypotheses. As we shall see, the challenge here, however, is to pass the right environment to the two recursive calls:

$$\begin{aligned}
&\mathsf{correctness\text{-}opt}\ (\mathsf{app}\ \mathsf{t}_1\ \mathsf{t}_2)\ \mathsf{env}\ \xi = \\
&\quad \mathsf{stack\text{-}eval}\ (\mathsf{forget}\ (\mathsf{app}\ \mathsf{t}_1\ \mathsf{t}_2))\ \mathsf{env}\ \xi \\
&\qquad \equiv\langle\ \text{definition of } \mathsf{forget}\ \rangle \\
&\quad \mathsf{stack\text{-}eval}\ (\mathsf{app}\ (\mathsf{forget}\ \mathsf{t}_1)\ (\mathsf{forget}\ \mathsf{t}_2))\ \mathsf{env}\ \xi \\
&\qquad \equiv\langle\ \text{definition of } \mathsf{stack\text{-}eval}\ \rangle \\
&\quad \mathsf{stack\text{-}eval}\ (\mathsf{forget}\ \mathsf{t}_1)\ \mathsf{env}\ (\mathsf{stack\text{-}eval}\ (\mathsf{forget}\ \mathsf{t}_2)\ \mathsf{env}\ ,\ \xi) \\
&\qquad \equiv\langle\ \text{induction hypothesis}\ \rangle \\
&\quad \mathsf{stack\text{-}eval}\ (\mathsf{forget}\ \mathsf{t}_1)\ \mathsf{env}\ (\mathsf{stack\text{-}eval\text{-}opt}\ \mathsf{t}_2\ (\mathsf{restrict}\ \Delta_2\ \mathsf{env})\ ,\ \xi) \\
&\qquad \equiv\langle\ \text{induction hypothesis}\ \rangle \\
&\quad \mathsf{stack\text{-}eval\text{-}opt}\ \mathsf{t}_1\ (\mathsf{restrict}\ \Delta_1\ \mathsf{env})\ (\mathsf{stack\text{-}eval\text{-}opt}\ \mathsf{t}_2\ (\mathsf{restrict}\ \Delta_2\ \mathsf{env})\ ,\ \xi) \\
&\qquad \equiv\langle\ \text{projection lemmas}\ \rangle \\
&\quad \mathsf{stack\text{-}eval\text{-}opt}\ \mathsf{t}_1\ (\pi_1\ \Delta_1\ \Delta_2\ (\mathsf{restrict}\ (\Delta_1 \cup \Delta_2)\ \mathsf{env})) \\
&\qquad (\mathsf{stack\text{-}eval\text{-}opt}\ \mathsf{t}_2\ (\pi_2\ \Delta_1\ \Delta_2\ (\mathsf{restrict}\ (\Delta_1 \cup \Delta_2)\ \mathsf{env}))\ ,\ \xi) \\
&\qquad \equiv\langle\ \text{use as definition}\ \rangle \\
&\quad \mathsf{stack\text{-}eval\text{-}opt}\ (\mathsf{app}\ \mathsf{t}_1\ \mathsf{t}_2)\ (\mathsf{restrict}\ (\Delta_1 \cup \Delta_2)\ \mathsf{env})\ \xi \\
&\quad \square
\end{aligned}$$

We can almost read off the required definition – but the two recursive calls require two different environments. To construct the appropriate environment, we define a pair of auxiliary functions, projecting the desired elements out of a larger environment:

$$\pi_1 : (\Delta_1\ \Delta_2 : \mathsf{Subset}\ \Gamma) \to [\![\mathsf{Env}]\!]_s \lfloor \Delta_1 \cup \Delta_2 \rfloor \to [\![\mathsf{Env}]\!]_s \lfloor \Delta_1 \rfloor$$
$$\pi_2 : (\Delta_1\ \Delta_2 : \mathsf{Subset}\ \Gamma) \to [\![\mathsf{Env}]\!]_s \lfloor \Delta_1 \cup \Delta_2 \rfloor \to [\![\mathsf{Env}]\!]_s \lfloor \Delta_2 \rfloor$$

Using these definitions, we can define an optimizing version of apply as follows:

$$\begin{aligned}&\mathsf{apply\text{-}opt} : (\Delta_1\ \Delta_2 : \mathsf{Subset}\ \Gamma) \to ([\![\mathsf{Env}]\!]_s \lfloor \Delta_1 \rfloor \to [\![\,a \Rightarrow b\,]\!]_s) \to\\ &\quad ([\![\mathsf{Env}]\!]_s \lfloor \Delta_2 \rfloor \to [\![\,a\,]\!]_s) \to ([\![\mathsf{Env}]\!]_s \lfloor \Delta_1 \cup \Delta_2 \rfloor \to [\![\,b\,]\!]_s)\\ &\mathsf{apply\text{-}opt}\ \Delta_1\ \Delta_2\ f\ x\ env\ \xi\ = f\ (\pi_1\ \Delta_1\ \Delta_2\ env)\ (x\ (\pi_2\ \Delta_1\ \Delta_2\ env)\ ,\ \xi)\end{aligned}$$

Here our types guarantee that our evaluator is still type safe by construction. In particular, these types guarantee that the values required are always available in the appropriate environment.

We do still need to show that π_1 and π_2 compute the same environment as restrict, but these two lemmas are entirely straightforward to prove:

$$\begin{aligned}&\pi_1\text{-lemma} : (\Delta_1\ \Delta_2 : \mathsf{Subset}\ \Gamma) \to (env : [\![\mathsf{Env}]\!]_s\ \Gamma) \to\\ &\quad (\mathsf{restrict}\ \Delta_1\ env) \equiv \pi_1\ \Delta_1\ \Delta_2\ (\mathsf{restrict}\ (\Delta_1 \cup \Delta_2)\ env)\\ &\pi_2\text{-lemma} : (\Delta_1\ \Delta_2 : \mathsf{Subset}\ \Gamma) \to (env : [\![\mathsf{Env}]\!]_s\ \Gamma) \to\\ &\quad (\mathsf{restrict}\ \Delta_2\ env) \equiv \pi_2\ \Delta_1\ \Delta_2\ (\mathsf{restrict}\ (\Delta_1 \cup \Delta_2)\ env)\end{aligned}$$

Finally, we use these two definitions – enter-opt and push-opt – to define an optimizing stack-based evaluator. Of the two remaining cases, return and lookup, the first can remain unchanged. The lookup function used for variables, however, must be adapted slightly to use the new variable and environment representation – even if its behaviour stays the same.

$$\begin{aligned}&\mathsf{stack\text{-}eval\text{-}opt} : \mathsf{CDB}\ \Gamma\ \Delta\ a \to [\![\mathsf{Env}]\!]_s \lfloor \Delta \rfloor \to [\![\,a\,]\!]_s\\ &\mathsf{stack\text{-}eval\text{-}opt}\ (\mathsf{val}\ x) && = \mathsf{push}\ x\\ &\mathsf{stack\text{-}eval\text{-}opt}\ (\mathsf{var}\ t) && = \mathsf{lookup}\ t\\ &\mathsf{stack\text{-}eval\text{-}opt}\ (\mathsf{lam}\ t) && = \mathsf{enter\text{-}opt}\ \ \Delta\ (\mathsf{stack\text{-}eval\text{-}opt}\ t)\\ &\mathsf{stack\text{-}eval\text{-}opt}\ (\mathsf{app}\ t_1\ t_2) && = \mathsf{apply\text{-}opt}\ \ \Delta_1\ \Delta_2\ (\mathsf{stack\text{-}eval\text{-}opt}\ t_1)\ (\mathsf{stack\text{-}eval\text{-}opt}\ t_2)\end{aligned}$$

Here we have hidden a few implicit arguments, introduced in the patterns, from the typeset code. The correctness of this definition follows immediately from the calculations we have given above.

6 Discussion

There is a substantial body of work on verifying and calculating compilers and abstract machines (Bahr & Hutton, 2015, 2020, 2022; Pickard & Hutton, 2021). Just as in this article, these calculations start from an evaluator for the source language. Each of these calculations define a stack-based interpreter with the same behaviour as the evaluator, but the motivation for *how* this stack-based interpreter is calculated is usually given by the surrounding text. The most closely related work by Pickard & Hutton (2021) calculates a compiler for an arithmetic expressions and exceptions in an intrinsically typed fashion. In this calculation, the (indexed) types for stacks, code, and stack-based interpreters are not derived, but rather drawn from existing work by McKinna & Wright (2006); there is no general pattern to systematically derive these types. The purpose of this paper is to demonstrate how these types follow systematically using the α-γ conversion functions. Although the definition of the stack-based denotation of our object types and these conversion functions require some creativity – these form a part of our *specification*. They can be re-used to map other languages and interpreters to their stack-based implementation.

The work by Pickard & Hutton (2021) does not address higher-order functions; it is not immediately obvious how to extend their stack types to facilitate this. Although we have not defined a compiler for the lambda calculus, our specification is general enough to specify and calculate a stack-based evaluator for the simply typed lambda calculus. Calculating the corresponding compiler is less straightforward: the variable context changes as we traverse under a binder. Adapting the derivation of the target language and its semantics accordingly is still an open problem. The existing work on handling variable binding by Allais *et al.* (2018) may offer a viable solution, shifting to the Kripke function space in our compiler derivation to extend the context accordingly as every time we go under a binder.

There is a further limitation of the approach in this paper. The same α-γ conversion functions are used for both the lambda calculus and the language of arithmetic expressions. When determining the type of the arguments a function expects on the stack, the args function used in the definition of our ‘stack semantics’ for types, $[\![\_]\!]_s$, is defined as follows.

```
args : U → List U → Set
args (a ⇒ b) σ  = ⟦ a ⟧ₛ × args b σ
```

In particular, the value stored on the top of the stack is itself a ‘stack value’ rather than a ‘direct value’. As a result, base types, such as natural numbers or booleans, are represented as *functions* operating on an arbitrary (possibly empty) stack rather than first-order data. In the calculations throughout the paper, occasionally an additional application of the γ is used to convert these functions back to natural numbers and booleans. This is a symptom of the choice to define the conversion functions independently of both the source language and the direct semantics. Alternatively, one could replace the type $[\![a]\!]_s$ with the direct semantics $[\![a]\!]$, but this choice would also store functions on the stack. A better approach would be to define a first-order value representation:

```
val : U → Set
val nat      = ℕ
val bool     = Bool
val (a ⇒ b)  = Σ (List U) (λ Γ → Term (a :: Γ) b × Env val Γ)
```

Using this definition, we ensure that all the data on the stack is indeed first-order. Any arguments to higher-order functions are stored as closures on the stack. The corresponding calculations would produce more efficient machines, without some of the spurious conversions between data representations – at the expense of having to use the type of the source language, Term, in the definition of the stacks’ type.

The current work, however, has not yet considered effectful interpreters, such as those that use exceptions (Pickard & Hutton, 2021) or other monadic effects Bahr & Hutton (2022) and Garby *et al.* (2024). The specification of the stack-based interpreter can be readily extended to these settings. Recall the very first specification of the stack-based evaluator for arithmetic expressions boils down to the following equation:

$$\forall\,(\mathsf{e} : \mathsf{Expr}\ \mathsf{a}) \to \alpha\ (\mathsf{eval}\ \mathsf{e}) \equiv \mathsf{machine}\ \mathsf{e}$$

If the evaluator at the start of the derivation may throw an exception or use some other monadic effect, the obvious reformulation of the specification reads:

$$\forall\,(\mathsf{e} : \mathsf{Expr}\ \mathsf{a}) \to \mathsf{map}\ \alpha\ (\mathsf{eval}\ \mathsf{e}) \equiv \mathsf{machine}\ \mathsf{e}$$

where the conversion α is mapped over the result of evaluation. This ensures that the resulting machine produces a monadic result in line with the original evaluator. Although the CPS transformation may no longer be necessary to fix the order of evaluation in this setting, it does help calculate a linear sequence of instructions, as opposed to the syntax trees that the stack-based evaluator takes as its input.

The most closely related work in spirit is that by Meijer (1992) and Elliott (2020). Where Meijer's work suggests using the abstraction and concretization functions to determine the type of the resulting machine, the types are slightly different from those presented here. Using the notation from this paper, Meijer defines:

$$\begin{aligned}&[\![\_]\!]_{\mathsf{s}} : \mathsf{U} \to \mathsf{Set}\\&[\![\,\mathsf{a}\,]\!]_{\mathsf{s}} = \mathsf{args}\ \mathsf{a}\ \top \to \mathbb{N}\end{aligned}$$

That is, this definition fixes the return type to be natural numbers, the only base type in the object language Meijer considers. Inspired by the observation made by Elliott (2020), we tease the stack semantics for our types into two parts: the argument types are on the input stack; these are replaced by the function's result type in the output stack.

Gibbons (2002, §4) also showed how to compute a compiler using uncurrying. The calculation, however, is difficult to formalize as there is no typing discipline guaranteeing the absence of stack underflows. More recently, Gibbons (2022) has explored how generalized composition operators, initially suggested by Wand (1982), capture CPS transformed functions that pass intermediate results from one computation to the next. Gibbons's work, like ours, makes use of *dependent types* to assign static types to these composition operators. Just as our stack semantics splits a (function) type into its arguments and return type, Gibbons's Arrow type collects a function's arguments in a list of types. This type is then used to define the generalized composition operators used throughout the derivation.

The defunctionalized CPS transformation is a standard technique for turning structurally recursive computations into tail recursive abstract machines, dating back to (Reynolds, 1972). Since then it has been popularized by Danvy and his many co-authors (Ager *et al.*, 2003; Danvy, 2004, 2008; Danvy & Nielsen, 2001). More recently, Huang & Yallop (2023) have studied how to defunctionalize dependently typed higher-order functions. A similar pattern of type indices appears when typing Danvy and Nielsen's (2004) refocusing transformations (Swierstra, 2012).

Acknowledgements. I would like to thank Guillaume Allais, Roger Bosman, Jacco Krijnen, Lawrence Chonavel, Mitchell Pickard, Graham Hutton, and the anonymous reviewers for their feedback on earlier versions of this paper. Cas van der Rest was an invaluable sparring partner who contributed to a much improved revision of this work. The thoughtful and constructive feedback from the anonymous reviewers helped to improve this paper significantly.

Conflicts of Interest. None.

References

Ager, M. S., Biernacki, D., Danvy, O. & Midtgaard, J. (2003) A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming,* pp. 8–19.

Allais, G., Atkey, R., Chapman, J., McBride, C. & McKinna, J. (2018) A type and scope safe universe of syntaxes with binding: their semantics and proofs. In *Proceedings of the ACM on Programming Languages* **2**(ICFP), 1–30.

Augustsson, L. & Carlsson, M. (1999) An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming, Gothenburg*.

Bahr, P. & Hutton, G. (2015) Calculating correct compilers. *J. Funct. Program.* **25**(e14), 1–44.

Bahr, P. & Hutton, G. (2020) Calculating correct compilers ii: Return of the register machines. *J. Funct. Program.* **30**, e25.

Bahr, P. & Hutton, G. (2022) Monadic compiler calculation. *Proc. ACM Program. Lang.* **6**(ICFP), 93:1–93:29.

Danvy, O. (2004) A rational deconstruction of landin's secd machine. In *Symposium on Implementation and Application of Functional Languages,* pp. 52–71. Springer.

Danvy, O. (2008) From reduction-based to reduction-free normalization. In *International School on Advanced Functional Programming*, pp. 66–164. Springer.

Danvy, O. & Nielsen, L. R. (2001) Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pp. 162–174.

Danvy, O. & Nielsen, L. R. (2004) Refocusing in reduction semantics. *BRICS Rep. Ser.* **11**(26).

Elliott, C. (2020) *Calculating Compilers Categorically*. Unpublished draft.

Garby, Z., Hutton, G. & Bahr, P. (2024) Calculating compilers effectively (Functional Pearl). In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium*, pp. 109–119.

Gibbons, J. (2002) Calculating functional programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction: International Summer School and Workshop Oxford, UK, April 10–14, 2000 Revised Lectures*, pp. 151–203. Springer.

Gibbons, J. (2022) Continuation-passing style, defunctionalization, accumulations, and associativity. *Art Sci. Eng. Program.* **6**. https://doi.org/10.22152/programming-journal.org/2022/6/7

Huang, Y. & Yallop, J. (2023) Defunctionalization with dependent types. *Proc. ACM Program. Lang.* **7**(PLDI), 516–538.

Leroy, X. (2009) Formal verification of a realistic compiler. *Commun. ACM* **52**(7), 107–115.

McBride, C. (2008) Clowns to the left of me, jokers to the right: Dissecting data structures. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08, pp. 287–295.

McBride, C. (2018) Everybody's got to be somewhere. *Electron. Proc. Theor. Comput. Sci.* **275**(July), 53–69.

McKinna, J. & Wright, J. (2006) *A Type-Correct, Stack-Safe, Provably Correct Expression Compiler*. Accepted for publication in the Journal of Functional Programming.

Meijer, E. (1992) *Calculating Compilers*. PhD thesis, Radboud Universiteit Nijmegen.

Norell, U. (2007) *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology.

Pickard, M. & Hutton, G. (2021) Calculating Dependently-Typed Compilers. *Proc. ACM Program. Lang.* **5**(ICFP). https://doi.org/10.1145/3473587

Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference-Volume 2*, pp. 717–740.

Swierstra, W. (2012) From mathematics to abstract machine: A formal derivation of an executable Krivine machine. In Proceedings Fourth Workshop on *Mathematically Structured Functional Programming,* Tallinn, Estonia, 25 March 2012, Chapman, J. & Levy, P. B. (eds), Electronic Proceedings in Theoretical Computer Science, vol. 76, pp. 163–177. Open Publishing Association.

Tomé Cortiñas, C. & Swierstra, W. (2018) From algebra to abstract machine: A verified generic construction. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2018, pp. 78–90.

Wadler, P. (1988) Deforestation: Transforming programs to eliminate trees. *European Symposium on Programming* pp. 344–358. Springer.

Wand, M. (1982) Deriving target code as a representation of continuation semantics. *ACM Trans. Program. Lang. Syst.* **4**(3), 496–517.