Under consideration for publication in the Journal of Functional Programming

Calculating Correct Compilers

PATRICK BAHR
Department of Computer Science, University of Copenhagen, Denmark

GRAHAM HUTTON
School of Computer Science, University of Nottingham, UK

Abstract

In this article we present a new approach to the problem of calculating compilers. In particular, we develop a simple but general technique that allows us to derive correct compilers from high-level semantics by systematic calculation, with all the required compilation machinery falling naturally out of the calculation process. Our approach is based upon the use of standard equational reasoning techniques, and has been applied to calculate compilers for a wide range of language features and their combination, including arithmetic expressions, exceptions, local and global state, various forms of lambda calculi, bounded and unbounded loops, non-determinism, and interrupts. All the calculations have been mechanically verified using the Coq proof assistant.

1 Introduction

The desire to *calculate compilers* has been a key objective in the field of program transformation since its earliest days. Starting from a high-level semantics for a source language, the aim is to transform the semantics into a *compiler* that translates source programs into a lower-level *target language*, together with a *virtual machine* that executes the resulting target programs. There are two important advantages of this approach. Firstly, the definitions for the compiler, target language and virtual machine are *systematically derived* during the transformation process, rather than having to be manually defined by the user. And secondly, the resulting compiler and virtual machine do not usually require subsequent proofs of correctness, as they are *correct by construction* (Backhouse, 2003).

The idea of calculating compilers in this manner has been explored by a number of authors; for example, see Wand (1982); Meijer (1992); Ager *et al.* (2003b). However, it has traditionally been viewed as an advanced topic that requires considerable knowledge and experience with concepts such as continuations and defunctionalisation (Reynolds, 1972). In this article, we show that compilers can in fact be calculated in a simple and straightforward manner, without the need for such additional machinery, using standard equational reasoning techniques. Our new approach builds upon previous work in the area, and focuses specifically on compilers that target stack-based virtual machines.

We develop our approach in two stages, in which we first introduce the basic ideas in a series of steps, and then show how the steps can be combined. More specifically, we start by defining a semantics for the source language in the form of an evaluation function. Then we calculate more general versions of the evaluation function that incorporate first a stack

and then also a continuation, which in turn make the manipulation of arguments values and control flow explicit. Using equational specifications that capture the intended behaviour of the generalised functions, we calculate their definitions by *constructive induction* (Backhouse, 2003), using the desire the apply the induction hypotheses as the driving force for the calculation process. We then apply defunctionalisation to factorise the generalised evaluation function into a compiler and a virtual machine. Finally, we simplify the process by combining the separate transformations into a single step, which starts directly from an equational specification for the correctness of the compiler, and avoids the need to use continuations and defunctionalisation during the calculation process.

The techniques that we use are all well-known. Our contribution is to show how they can be applied in a novel manner, in combination with the use of *partial specifications* to avoid predetermining implementation decisions, to give a new approach to calculating compilers that is both simple and generally applicable. Our approach has been used to calculate compilers for a wide range of language features and their combination, including arithmetic expressions, exceptions, local and global state, various forms of lambda calculi, bounded and unbounded loops, non-determinism and interrupts. We illustrate the utility of our technique using a series of such examples of increasing complexity, starting with a simple expression language to introduce the basic ideas.

All the programs and calculations in the article are written in Haskell, but we only use the basic concepts of recursive types, recursive functions, and inductive proofs. Whereas in many articles calculations are often omitted or compressed for reasons of brevity, in this article they are the central focus, so are presented in detail. All the calculations have also been mechanically verified using the Coq proof assistant, and the proof scripts are available online as supplementary material for the article.

2 Arithmetic Expressions

To introduce our technique, we begin by considering a simple language of arithmetic expressions comprising integer values and an addition operator:

$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

We calculate a compiler for this language in a series of steps, starting with the definition of a semantics for the language, to which we then apply a number of transformations. We then simplify the process by combining the separate transformation steps, which results in a simple but powerful new approach to calculating compilers.

2.1 Step 1 – Define the semantics

The semantics for our expression language is most naturally given by defining a function that simply evaluates an expression to an integer value:

$$\begin{aligned}
&\mathit{eval} && :: \mathit{Expr} \to \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ n) && = n\\
&\mathit{eval}\ (\mathit{Add}\ x\ y) && = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}$$

Note that the definition for *eval* is *compositional* (Schmidt, 1986), in the sense that the semantics of addition is given purely in terms of the semantics of its two argument expressions. With a view to using simple inductive proof methods, we will typically aim to define our semantics in such a compositional manner.

2.2 Step 2 – Transform into a stack transformer

The next step is to transform the evaluation function into a version that utilises a stack, in order to make the manipulation of argument values explicit. In particular, rather than returning a single value of type *Int*, we seek to derive a more general evaluation function, $eval_S$, that takes a stack of integers as an additional argument, and returns a modified stack given by pushing the value of the expression onto the top of the stack. More precisely, if we represent a stack as a list of integers (where the head is the top element)

type $Stack = [Int]$

then we seek to derive a function

$eval_S :: Expr \to Stack \to Stack$

such that:

$$eval_S\ e\ s \quad = \quad eval\ e : s \tag{1}$$

Rather than first defining the function $eval_S$ and then separately proving by induction that it satisfies the above equation, we aim to *calculate* a definition for $eval_S$ that satisfies the equation by *constructive induction* (Backhouse, 2003) on the expression e, using the desire to apply the induction hypotheses as the driving force for the calculation process. In the base case, $Val\ n$, the calculation is trivial:

$$\begin{array}{ll} & eval_S\ (Val\ n)\ s \\ = & \quad \{\ \text{specification (1)}\ \} \\ & eval\ (Val\ n) : s \\ = & \quad \{\ \text{definition of } eval\ \} \\ & n : s \\ = & \quad \{\ \text{define: } push_S\ n\ s = n : s\ \} \\ & push_S\ n\ s \end{array}$$

Note that in the final step we defined an auxiliary function, $push_S$, that captures the idea of pushing a number onto the stack. In the inductive case, $Add\ x\ y$, we proceed as follows:

$$\begin{array}{ll} & eval_S\ (Add\ x\ y)\ s \\ = & \quad \{\ \text{specification (1)}\ \} \\ & eval\ (Add\ x\ y) : s \\ = & \quad \{\ \text{definition of } eval\ \} \\ & (eval\ x + eval\ y) : s \end{array}$$

Now we appear to be stuck, as no further definitions can be applied. However, as we are performing an inductive calculation, we can make use of the induction hypotheses for the

two argument expressions x and y, namely:

$$eval_{\mathsf{S}}\ x\ s' = eval\ x : s'$$

$$eval_{\mathsf{S}}\ y\ s' = eval\ y : s'$$

In order to use these hypotheses, it is clear that we must push the values *eval x* and *eval y* onto the stack, which can readily be achieved by introducing another auxiliary function, add_{S}, that captures the idea of adding together the top two numbers on the stack. The remainder of the calculation is then straightforward:

$$\begin{array}{ll} & (eval\ x + eval\ y) : s \\ = & \quad \{ \text{ define: } add_{\mathsf{S}}\ (n : m : s) = (m+n) : s \ \} \\ & add_{\mathsf{S}}\ (eval\ y : eval\ x : s) \\ = & \quad \{ \text{ induction hypothesis for } y \ \} \\ & add_{\mathsf{S}}\ (eval_{\mathsf{S}}\ y\ (eval\ x : s)) \\ = & \quad \{ \text{ induction hypothesis for } x \ \} \\ & add_{\mathsf{S}}\ (eval_{\mathsf{S}}\ y\ (eval_{\mathsf{S}}\ x\ s)) \end{array}$$

Note that pushing *eval x* onto the stack before *eval y* in this calculation corresponds to our intuition that addition should evaluate its arguments from left-to-right. It would be perfectly valid to push the values in the opposite order, which would correspond to right-to-left evaluation. In conclusion, we have calculated the following definition

$$\begin{array}{ll} eval_{\mathsf{S}} & :: Expr \to Stack \to Stack \\ eval_{\mathsf{S}}\ (Val\ n)\ s & = push_{\mathsf{S}}\ n\ s \\ eval_{\mathsf{S}}\ (Add\ x\ y)\ s & = add_{\mathsf{S}}\ (eval_{\mathsf{S}}\ y\ (eval_{\mathsf{S}}\ x\ s)) \end{array}$$

where

$$\begin{array}{ll} push_{\mathsf{S}} & :: Int \to Stack \to Stack \\ push_{\mathsf{S}}\ n\ s & = n : s \\ add_{\mathsf{S}} & :: Stack \to Stack \\ add_{\mathsf{S}}\ (n : m : s) & = (m+n) : s \end{array}$$

Finally, our original evaluation function *eval* can now be recovered from our new function by substituting the empty stack into equation (1) from which $eval_{\mathsf{S}}$ was constructed, and selecting the unique value in the resulting singleton stack:

$$\begin{array}{ll} eval & :: Expr \to Int \\ eval\ e & = head\ (eval_{\mathsf{S}}\ e\ [\,]) \end{array}$$

2.3 Step 3 – Transform into continuation-passing style

The next step is to transform the new function $eval_{\mathsf{S}}$ into *continuation-passing style* (CPS) (Reynolds, 1972), in order to make the flow of control explicit. In particular, we seek to derive a more general evaluation function, $eval_{\mathsf{C}}$, that takes a function from stacks to stacks (the continuation) as an additional argument, which is used to process the stack that results from evaluating the expression. More precisely, if we define a type for continuations

type $Cont = Stack \to Stack$

then we seek to derive a function

$$eval_{\mathsf{C}} :: Expr \to Cont \to Cont$$

such that:

$$eval_{\mathsf{C}}\ e\ c\ s \quad = \quad c\ (eval_{\mathsf{S}}\ e\ s) \tag{2}$$

We calculate the definition for $eval_{\mathsf{C}}$ directly from this equation by constructive induction on the expression e. The base case is once again trivial,

$$\begin{array}{ll} & eval_{\mathsf{C}}\ (Val\ n)\ c\ s \\ = & \quad \{\ \text{specification (2)}\ \} \\ & c\ (eval_{\mathsf{S}}\ (Val\ n)\ s) \\ = & \quad \{\ \text{definition of } eval_{\mathsf{S}}\ \} \\ & c\ (push_{\mathsf{S}}\ n\ s) \end{array}$$

while for the inductive case we calculate as follows:

$$\begin{array}{ll} & eval_{\mathsf{C}}\ (Add\ x\ y)\ c\ s \\ = & \quad \{\ \text{specification (2)}\ \} \\ & c\ (eval_{\mathsf{S}}\ (Add\ x\ y)\ s) \\ = & \quad \{\ \text{definition of } eval_{\mathsf{S}}\ \} \\ & c\ (add_{\mathsf{S}}\ (eval_{\mathsf{S}}\ y\ (eval_{\mathsf{S}}\ x\ s))) \\ = & \quad \{\ \text{function composition}\ \} \\ & (c \circ add_{\mathsf{S}})\ (eval_{\mathsf{S}}\ y\ (eval_{\mathsf{S}}\ x\ s)) \\ = & \quad \{\ \text{induction hypothesis for } y\ \} \\ & eval_{\mathsf{C}}\ y\ (c \circ add_{\mathsf{S}})\ (eval_{\mathsf{S}}\ x\ s) \\ = & \quad \{\ \text{induction hypothesis for } x\ \} \\ & eval_{\mathsf{C}}\ x\ (eval_{\mathsf{C}}\ y\ (c \circ add_{\mathsf{S}}))\ s \end{array}$$

In conclusion, we have calculated the following definition:

$$\begin{array}{ll} eval_{\mathsf{C}} & :: Expr \to Cont \to Cont \\ eval_{\mathsf{C}}\ (Val\ n)\ c\ s & = c\ (push_{\mathsf{S}}\ n\ s) \\ eval_{\mathsf{C}}\ (Add\ x\ y)\ c\ s & = eval_{\mathsf{C}}\ x\ (eval_{\mathsf{C}}\ y\ (c \circ add_{\mathsf{S}}))\ s \end{array}$$

Our previous evaluation function $eval_{\mathsf{S}}$ can then be recovered by substituting the identity continuation into equation (2) from which $eval_{\mathsf{C}}$ was constructed:

$$\begin{array}{ll} eval_{\mathsf{S}} & :: Expr \to Cont \\ eval_{\mathsf{S}}\ e & = eval_{\mathsf{C}}\ e\ (\lambda s \to s) \end{array}$$

2.4 Step 4 – Transform back to first-order style

The final step is to transform the evaluation function back into first-order style, using the technique of *defunctionalisation* (Reynolds, 1972). In particular, rather than using the function type $Cont = Stack \to Stack$ for continuations passed as arguments and returned

as results, we define a datatype that represents the forms of continuations that we need for our evaluation function, rather than using the actual functions themselves.

Within the definitions for $eval_S$ and $eval_C$, there are only three forms of continuations that are actually used, namely one to invoke the evaluator, one to push an integer onto the stack, and one to add the top two values on the stack. We begin by separating out these three forms, by giving them names and abstracting over their free variables. That is, we define three combinators for constructing the required forms of continuations:

$$
\begin{array}{ll}
halt_C & :: Cont \\
halt_C & = \lambda s \to s \\
push_C & :: Int \to Cont \to Cont \\
push_C\ n\ c & = c \circ push_S\ n \\
add_C & :: Cont \to Cont \\
add_C\ c & = c \circ add_S
\end{array}
$$

Using these combinators, our evaluation functions can now be rewritten as follows:

$$
\begin{array}{ll}
eval_S & :: Expr \to Cont \\
eval_S\ e & = eval_C\ e\ halt_C \\
eval_C & :: Expr \to Cont \to Cont \\
eval_C\ (Val\ n)\ c & = push_C\ n\ c \\
eval_C\ (Add\ x\ y)\ c & = eval_C\ x\ (eval_C\ y\ (add_C\ c))
\end{array}
$$

It is easy to check by unfolding definitions that these new definitions are equivalent to the previous versions. The next stage in applying defunctionalisation is to define a new datatype, *Code*, whose constructors represent the three combinators that we have isolated. We write the definition in GADT style to highlight the correspondence:

$$
\begin{array}{l}
\textbf{data}\ Code\ \textbf{where} \\
\quad HALT :: Code \\
\quad PUSH :: Int \to Code \to Code \\
\quad ADD\ \ :: Code \to Code
\end{array}
$$

The types for the constructors in this definition are obtained simply by replacing occurrences of *Cont* in the types for the combinators by *Code*. The use of the name *Code* for the type reflects the fact that its values represent code sequences for a virtual machine that evaluates arithmetic expressions using a stack. For example, $PUSH\ 1\ (PUSH\ 2\ (ADD\ HALT))$ is the code that corresponds to the expression $Add\ (Val\ 1)\ (Val\ 2)$.

The fact that values of type *Code* represent continuations of type *Cont* can be formalised by defining a function that maps from one to the other:

$$
\begin{array}{ll}
exec & :: Code \to Cont \\
exec\ HALT & = halt_C \\
exec\ (PUSH\ n\ c) & = push_C\ n\ (exec\ c) \\
exec\ (ADD\ c) & = add_C\ (exec\ c)
\end{array}
$$

By expanding out the definitions for the type *Cont* and its three combinators, we see that *exec* is a first-order, tail recursive function that executes code using an initial stack to give a final stack. That is, *exec* is a virtual machine for executing code:

$$\begin{array}{ll} exec & :: Code \to Stack \to Stack \\ exec\ HALT\ s & = s \\ exec\ (PUSH\ n\ c)\ s & = exec\ c\ (n : s) \\ exec\ (ADD\ c)\ (n : m : s) & = exec\ c\ ((m+n) : s) \end{array}$$

Finally, defunctionalisation itself proceeds by replacing occurrences of the combinators $push_{\mathsf{C}}$, add_{C} and $halt_{\mathsf{C}}$ in the evaluation functions $eval_{\mathsf{S}}$ and $eval_{\mathsf{C}}$ by their respective counterparts from the datatype *Code*, which results in the following two definitions:

$$\begin{array}{ll} comp & :: Expr \to Code \\ comp\ e & = comp'\ e\ HALT \\ comp' & :: Expr \to Code \to Code \\ comp'\ (Val\ n)\ c & = PUSH\ n\ c \\ comp'\ (Add\ x\ y)\ c & = comp'\ x\ (comp'\ y\ (ADD\ c)) \end{array}$$

That is, we have now derived a function *comp* that compiles an expression to code, which is itself defined in terms of an auxiliary function $comp'$ that takes a code continuation as an additional argument. This is essentially the same compiler as developed by Hutton (2007, Chapter 13), except that all the required compilation machinery — compiler, target language, and virtual machine — has now been systematically derived from a high-level semantics for the source language using equational reasoning techniques.

The correctness of the compilation functions *comp* and $comp'$ is captured by the following two equations, which are consequences of defunctionalisation (or can be verified by simple inductive proofs on the expression argument):

$$exec\ (comp\ e)\ s = eval_{\mathsf{S}}\ e\ s$$

$$exec\ (comp'\ e\ c)\ s = eval_{\mathsf{C}}\ e\ (exec\ c)\ s$$

In order to understand these equations, we expand their right-hand sides using the original specifications (1) and (2) for the new evaluation functions, to give:

$$exec\ (comp\ e)\ s = eval\ e : s$$

$$exec\ (comp'\ e\ c)\ s = exec\ c\ (eval\ e : s)$$

The first equation now states that executing the compiled code for an expression produces the same result as pushing the value of the expression onto the stack, which establishes the correctness of *comp*. In turn, the second equation states that compiling an expression and then executing the resulting code together with additional code gives the same result as executing the additional code with the value of the expression on top of the stack, which establishes the correctness of $comp'$. These are the same correctness conditions as used by Hutton (2007, Chapter 13), except that they are now satisfied *by construction*.

2.5 Combining the transformation steps

We have now shown how a compiler for simple arithmetic expressions can be developed using a systematic four-step process, which is summarised below:

1. Define an evaluation function in a compositional manner;
2. Calculate a generalised version that uses a stack;
3. Calculate a further generalised version that uses continuations;
4. Defunctionalise to produce a compiler and a virtual machine.

However, there appears to be some opportunities for simplifying this process. In particular, steps 2 and 3 both calculate generalised versions of the original evaluation function. Could these steps be combined to avoid the need for two separate generalisation steps? In turn, step 3 introduces the use of continuations, which are then immediately removed in step 4. Could these steps be combined to avoid the need for continuations? In fact, it turns out that *all* the transformation steps 2–4 can be combined together. This section shows how this can be achieved, and explains the benefits that result from doing so.

In order to simplify the above stepwise process, let us first consider the types and functions that are involved in more detail. We started off by defining a datatype *Expr* that represents the syntax of the source language, together with a function $eval :: Expr \to Int$ that provides a semantics for the language, and a datatype *Stack* that corresponds to a stack of integer values. Then we derived four additional components:

- A datatype *Code* that represents the code for the virtual machine;
- A function $comp :: Expr \to Code$ that compiles expressions to code;
- A function $comp' :: Expr \to Code \to Code$ that also takes a code continuation;
- A function $exec :: Code \to Stack \to Stack$ that provides a semantics for code.

Moreover, the relationships between the semantics, compilers and virtual machine were captured by the following two correctness equations:

$$exec\ (comp\ e)\ s = eval\ e : s \tag{3}$$

$$exec\ (comp'\ e\ c)\ s = exec\ c\ (eval\ e : s) \tag{4}$$

The key to combining the transformation steps is to use these two equations directly as a specification for the four additional components, from which we then aim to calculate definitions that satisfy the specification. Given that the equations involve three known definitions (*Expr*, *eval* and *Stack*) and four unknown definitions (*Code*, *comp*, *comp*', and *exec*), this may seem like an impossible task. However, with the benefit of the experience gained from our earlier calculations, it turns out to be straightforward.

With begin with equation (4), and proceed by constructive induction on the expression e. In each case, we aim to rewrite the left-hand side $exec\ (comp'\ e\ c)\ s$ of the equation into the form $exec\ c'\ s$ for some code c', from which we can then conclude that the definition $comp'\ e\ c = c'$ satisfies the specification in this case. 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 function *exec*. In the base case, $Val\ n$, we proceed as follows:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Val}\ n)\ c)\ s \\
= & \quad \{\ \text{specification (4)}\ \} \\
 & \mathit{exec}\ c\ (\mathit{eval}\ (\mathit{Val}\ n) : s) \\
= & \quad \{\ \text{definition of } \mathit{eval}\ \} \\
 & \mathit{exec}\ c\ (n : s)
\end{array}
$$

Now we appear to be stuck, as no further definitions can be applied. However, recall that we are aiming to end up with an expression of the form $\mathit{exec}\ c'\ s$ for some code c'. That is, in order to complete the calculation we need to solve the equation:

$$\mathit{exec}\ c'\ s \quad = \quad \mathit{exec}\ c\ (n : s)$$

Note that we can't simply use this equation as a definition for *exec*, because the variables n and c are unbound in the body of the equation. The solution is to package these two variables up in the code argument c' by means of a new constructor in the *Code* datatype that takes these two variables as arguments,

$$\mathit{PUSH} :: \mathit{Int} \to \mathit{Code} \to \mathit{Code}$$

and define a new equation for *exec* as follows:

$$\mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s = \mathit{exec}\ c\ (n : s)$$

That is, executing the code $\mathit{PUSH}\ n\ c$ proceeds by pushing the value n onto the stack and then executing the code c, hence the choice of the name for the new constructor. Using these ideas, it is now straightforward to complete the calculation:

$$
\begin{array}{ll}
 & \mathit{exec}\ c\ (n : s) \\
= & \quad \{\ \text{definition of } \mathit{exec}\ \} \\
 & \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s
\end{array}
$$

The final expression now has the form $\mathit{exec}\ c'\ s$, where $c' = \mathit{PUSH}\ n\ c$, from which we conclude that the following definition satisfies the specification in the base case:

$$\mathit{comp}'\ (\mathit{Val}\ n)\ c = \mathit{PUSH}\ n\ c$$

For the inductive case, $\mathit{Add}\ x\ y$, we begin in the same way as above by first applying the specification and the definition of the evaluation function:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Add}\ x\ y)\ c)\ s \\
= & \quad \{\ \text{specification (4)}\ \} \\
 & \mathit{exec}\ c\ (\mathit{eval}\ (\mathit{Add}\ x\ y) : s) \\
= & \quad \{\ \text{definition of } \mathit{eval}\ \} \\
 & \mathit{exec}\ c\ (\mathit{eval}\ x + \mathit{eval}\ y : s)
\end{array}
$$

Once again we appear to be stuck, as no further definitions can be applied. However, as we are performing an inductive calculation, we can make use of the induction hypotheses for the two argument expressions x and y, namely

$$\mathit{exec}\ (\mathit{comp}'\ x\ c')\ s' = \mathit{exec}\ c'\ (\mathit{eval}\ x : s')$$

$$\mathit{exec}\ (\mathit{comp}'\ y\ c')\ s' = \mathit{exec}\ c'\ (\mathit{eval}\ y : s')$$

In order to use these hypotheses, it is clear that we must push *eval x* and *eval y* onto the stack, by transforming the expression that we are manipulating into the form $exec\ c'\ (eval\ y : eval\ x : s)$ for some code c'. That is, we need to solve the equation

$$exec\ c'\ (eval\ y : eval\ x : s) \quad = \quad exec\ c\ (eval\ x + eval\ y : s)$$

First of all, we generalise from the specific values *eval x* and *eval y* to give:

$$exec\ c'\ (m : n : s) \quad = \quad exec\ c\ ((n+m) : s)$$

Once again, however, we can't simply use this equation as a definition for *exec*, this time because the variable c is unbound in the body. The solution is to package this variable up in the code argument c' by means of a new constructor in the *Code* datatype

$$ADD :: Code \to Code$$

and define a new equation for *exec* as follows:

$$exec\ (ADD\ c)\ (m : n : s) = exec\ c\ ((n+m) : s)$$

That is, executing the code *ADD c* proceeds by adding the top two values on the stack and then executing the code c, hence the choice of the name for the new constructor. Using these ideas, the remainder of the calculation is straightforward:

$$\begin{array}{ll} & exec\ c\ (eval\ x + eval\ y : s) \\ = & \quad \{\ \text{definition of } exec\ \} \\ & exec\ (ADD\ c)\ (eval\ y : eval\ x : s) \\ = & \quad \{\ \text{induction hypothesis for } y\ \} \\ & exec\ (comp'\ y\ (ADD\ c))\ (eval\ x : s) \\ = & \quad \{\ \text{induction hypothesis for } x\ \} \\ & exec\ (comp'\ x\ (comp'\ y\ (ADD\ c)))\ s \end{array}$$

The final expression now has the form $exec\ c'\ s$, from which we conclude that the following definition satisfies the specification in the inductive case:

$$comp'\ (Add\ x\ y)\ c = comp'\ x\ (comp'\ y\ (ADD\ c))$$

Finally, we complete the development of our compiler by considering the function $comp :: Expr \to Code$, whose correctness was specified by equation (3). In a similar manner to equation (4), we aim to rewrite the left-hand side $exec\ (comp\ e)\ s$ of the equation into the form $exec\ c\ s$ for some code c, from which we can then conclude that $comp\ e = c$ satisfies the specification. In this case there is no need to use induction as simple calculation suffices, during which we introduce a new constructor $HALT :: Code$ in order to transform the expression being manipulated into the required form:

$$\begin{array}{ll} & exec\ (comp\ e)\ s \\ = & \quad \{\ \text{specification (3)}\ \} \\ & eval\ e : s \\ = & \quad \{\ \text{define: } exec\ HALT\ s = s\ \} \\ & exec\ HALT\ (eval\ e : s) \\ = & \quad \{\ \text{specification (4)}\ \} \\ & exec\ (comp'\ e\ HALT)\ s \end{array}$$

In conclusion, we have calculated the following definitions:

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Code} & = \mathit{HALT} \mid \mathit{PUSH\ Int\ Code} \mid \mathit{ADD\ Code} \\
\mathit{comp} & :: \mathit{Expr} \to \mathit{Code} \\
\mathit{comp\ e} & = \mathit{comp'\ e\ HALT} \\
\mathit{comp'} & :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code} \\
\mathit{comp'\ (Val\ n)\ c} & = \mathit{PUSH\ n\ c} \\
\mathit{comp'\ (Add\ x\ y)\ c} & = \mathit{comp'\ x\ (comp'\ y\ (ADD\ c))} \\
\mathit{exec} & :: \mathit{Code} \to \mathit{Stack} \to \mathit{Stack} \\
\mathit{exec\ HALT\ s} & = s \\
\mathit{exec\ (PUSH\ n\ c)\ s} & = \mathit{exec\ c\ (n : s)} \\
\mathit{exec\ (ADD\ c)\ (m : n : s)} & = \mathit{exec\ c\ ((n + m) : s)}
\end{array}
$$

These are precisely the same definitions as we produced in the previous section, except that they have now been calculated directly from a specification of compiler correctness, rather than indirectly by means of a series of separate transformation steps.

In conclusion, we have shown how a compiler for simple arithmetic expressions can be developed using a combined three-step approach, which is summarised below:

1. Define an evaluation function in a compositional manner;
2. Define equations that specify the correctness of the compiler;
3. Calculate definitions that satisfy these specifications.

In the remainder of the article we show how this approach scales to more interesting and sophisticated source languages. As we shall see, as we consider more complicated source languages it becomes natural to start with an incomplete, i.e. *partial*, specification of the correctness equations for the compiler, with the missing parts in the specification also being derived as part of the calculation process.

2.6 Reflection

We conclude this section with some reflective remarks on our original and combined approaches to calculating a compiler for arithmetic expressions.

Simplicity The original approach required the use of continuations and defunctionalisation, which are traditionally regarding as being 'advanced' concepts, and may not be familiar to some users who may be interested in calculating compilers. In contrast, the combined approach only uses simple equational reasoning techniques, in the form of constructive induction on the syntax of the source language.

Directness The original approach was driven by the desire to define generalised versions of the semantics for the source language, and the correctness of the resulting compiler arose indirectly as a consequence of the use of defunctionalisation. In contrast, the combined approach starts directly from the compiler correctness equations, from which the goal is then to calculate definitions that satisfy these equations.

Similarity The calculations in the combined approach proceed in a very similar manner to those in the original approach. Indeed, if we combine the original steps that introduce a stack and continuation into a single step by means of the specification

$$eval_{\mathsf{C}}\ e\ c\ s \quad = \quad c\ (eval\ e : s) \tag{5}$$

then the calculations have *precisely* the same structure, except that in the original approach we introduce continuation combinators that are defunctionalised to code constructors, whereas in the combined approach we introduce the code constructors directly. The correspondence also becomes syntactically evident if we use an infix operator, say \$\$, for the function *exec*. Then the specification for $comp'$ in the combined approach becomes

$$comp'\ e\ c\ \$\$\ s \quad = \quad c\ \$\$\ (eval\ e : s) \tag{6}$$

which has the same structure as specification (5) above for $eval_{\mathsf{C}}$, except that we use \$\$ rather than function application (itself sometimes written as infix \$), $comp'$ rather than $eval_{\mathsf{C}}$, and code rather than continuations. Using these specifications, the two calculations then become essentially the same. To illustrate this point, the base cases are shown side-by-side below; the inductive cases are just as similar too.

$$\begin{array}{ll}
& eval_{\mathsf{C}}\ (Val\ n)\ c\ s \\
= & \quad \{\ \text{specification (5)}\ \} \\
& c\ (eval\ (Val\ n) : s) \\
= & \quad \{\ \text{definition of } eval\ \} \\
& c\ (n : s) \\
= & \quad \{\ \text{define: } push\ n\ c\ s = c\ (n : s)\ \} \\
& push\ n\ c\ s
\end{array}
\qquad
\begin{array}{ll}
& comp'\ (Val\ n)\ c\ \$\$\ s \\
= & \quad \{\ \text{specification (6)}\ \} \\
& c\ \$\$\ (eval\ (Val\ n) : s) \\
= & \quad \{\ \text{definition of } eval\ \} \\
& c\ \$\$\ (n : s) \\
= & \quad \{\ \text{define: } PUSH\ n\ c\ \$\$\ s = c\ \$\$\ (n : s)\ \} \\
& PUSH\ n\ c\ \$\$\ s
\end{array}$$

Mechanisation Eliminating the use of continuations also has important benefits from the point of view of mechanically verifying our calculations. In particular, when using our original approach to calculate compilers for more sophisticated languages, we sometimes needed to store continuations on the stack. For example, this arises when considering languages that support exception handling as we shall do in section 3. However, this has the consequence that the stack type becomes non-strictly-positive, and hence unsuitable for formalisation in proof assistants such as Coq and Agda. In contrast, there are no such foundational problems when mechanising the calculations in our combined approach. All our compiler calculations have been mechanically verified using the Coq system, and the proof scripts are available online as supplementary material.

Exposition Given all the benefits of the combined approach noted above, why didn't we simply present this approach straight off rather than first presenting a more complicated approach? The primary reason is that the original, stepwise approach provides *motivation and explanation* for the specifications and calculations that are used in the combined approach, without which they may be difficult to understand. Moreover, starting off with the stepwise approach also facilities a comparison with related work (section 6), which is traditionally based upon the use of continuations and defunctionalisation.

3 Exceptions

We now extend the language of arithmetic expressions from section 2 with simple primitives for throwing and catching an exception:

data *Expr* = *Val Int* | *Add Expr Expr* | *Throw* | *Catch Expr Expr*

Informally, *Throw* aborts the current evaluation and throws an exception, while *Catch x h* behaves as the expression *x* unless it throws an exception in which case the catch behaves as the *handler* expression *h*. To define the semantics for this extended language in the form of an evaluation function, we first recall the *Maybe* type:

data *Maybe a* = *Just a* | *Nothing*

That is, a value of type *Maybe a* is either *Nothing*, which we view as an exceptional value, or has the form *Just x*, which we view as a normal value. Using this type, our original evaluator can be rewritten to take account of exceptions as follows:

```
eval                :: Expr → Maybe Int
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
```

This function could also be defined more concisely by exploiting the fact that the *Maybe* type is monadic, but for calculation purposes we prefer the above definition. The same comment applies to a number of other functions in this article.

The next step is to define equations that specify the correctness of the compiler for the extended language, by refining the correctness equations for the language of arithmetic expressions. As the source language becomes more complex, the more reasonable alternatives there are for how such a refinement is made. Because the calculation process is driven by the form of the specification, the choice of the specification plays a key role in determining the resulting implementations. We illustrate this idea by considering two alternative approaches for exceptions. Moreover, we will also see a refinement of the calculation process itself, in particular by starting with a *partial specification* for the compiler and a *partial definition* for the stack type, with the missing components in the specification also being derived during the calculation process.

3.1 First approach: one code continuation

The first approach simply extrapolates the specification from section 2, in which the compilation function $comp'$ takes a single code continuation as an additional argument. To this end, we use the same type for the new version of this function:

$comp' :: Expr \to Code \to Code$

However, rather than taking $Stack = [Int]$ as before, we use a different but isomorphic types for stacks, in which the elements are wrapped up in a new datatype *Elem*:

type $Stack = [Elem]$
data $Elem = VAL\ Int$

The reason for this change is that we will extend *Elem* with a new constructor during the calculation process. We could also start with the original stack type and observe during the calculation that we need to change the definition to make it extensible. Indeed, this is precisely what happened when we did this calculation for the first time.

For arithmetic expressions, the desired behaviour of $comp'$ was specified by the equation $exec\ (comp'\ e\ c)\ s = exec\ c\ (eval\ e : s)$. In the presence of exceptions, this equation needs to be refined to take account of the fact that *eval* now returns a value of type *Maybe Int* rather than *Int*. When *eval* succeeds, it is straightforward to modify the specification:

$$exec\ (comp'\ e\ c)\ s = exec\ c\ (VAL\ n : s) \qquad \text{if } eval\ e = Just\ n$$

However, if *eval* fails it is not clear how $comp'$ should behave, which we make explicit by introducing a new, but as yet undefined, function *fail* to handle this case:

$$exec\ (comp'\ e\ c)\ s = fail\ e\ c\ s \qquad \text{if } eval\ e = Nothing$$

Just as with $comp'$ itself, we aim to derive a definition for *fail* that satisfies this equation during the calculation process. In fact, however, there is no need for the function *fail* to take all three variables e, c and s as arguments, as the stack s on its own turns out to be sufficient for the purposes of the calculation:

$$exec\ (comp'\ e\ c)\ s = fail\ s \qquad \text{if } eval\ e = Nothing$$

Moreover, this simplification turns out to be necessary. In particular, if we start with the specification using three arguments the resulting calculation gets stuck due to the need to keep all the arguments aligned across the two cases for the specification of $comp'$. Again, this is what happened when we did this calculation for the first time.

In summary, we now have the following *partial specification* for the new compilation function $comp'$ in terms of an as yet undefined function $fail :: Stack \to Stack$:

$$\begin{array}{lcl} exec\ (comp'\ e\ c)\ s & = & \textbf{case}\ eval\ e\ \textbf{of} \\ & & \quad Just\ n \quad \to exec\ c\ (VAL\ n : s) \\ & & \quad Nothing \to fail\ s \end{array} \tag{7}$$

We now calculate a definition for $comp'$ from this equation by constructive induction on e, aiming to rewrite the left-hand side $exec\ (comp'\ e\ c)\ s$ into the form $exec\ c'\ s$ for some code c', from which we can then conclude that the definition $comp'\ e\ c = c'$ satisfies the specification in this case. As in the previous section, in order to do this we will find that we need to introduce new constructors into the code type, along with their interpretation by *exec*. Moreover, this time around we will also need to add a new constructor to the stack type. To simplify the presentation, we introduce these new components within the calculations as we go along. The base cases for *Val n* and *Throw* are trivial:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Val}\ n)\ c\ s) \\
= & \quad \{\ \text{specification (7)}\ \} \\
 & \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s = \mathit{exec}\ c\ (\mathit{VAL}\ n : s)\ \} \\
 & \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s
\end{array}
$$

and

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ \mathit{Throw}\ c\ s) \\
= & \quad \{\ \text{specification (7)}\ \} \\
 & \mathit{fail}\ s \\
= & \quad \{\ \text{define: } \mathit{exec}\ \mathit{FAIL}\ s = \mathit{fail}\ s\ \} \\
 & \mathit{exec}\ \mathit{FAIL}\ s
\end{array}
$$

The inductive case for *Add x y* starts in the same manner as the language without exceptions. First we apply the specification, then we introduce a code constructor *ADD* to bring the stack arguments into the form that we need to apply the induction hypothesis:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Add}\ x\ y)\ c\ s) \\
= & \quad \{\ \text{specification (7)}\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
 & \qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{exec}\ c\ (\mathit{VAL}\ (n+m) : s) \\
 & \qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
 & \quad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) = \mathit{exec}\ c\ (\mathit{VAL}\ (n+m) : s)\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
 & \qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
 & \qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
 & \quad \mathit{Nothing} \rightarrow \mathit{fail}\ s
\end{array}
$$

However, transforming the stack in the *Just* case alone is not sufficient to allow us to apply the induction hypothesis for *y*. In particular, for the inner case expression above to match the form of specification (7), the use of the stack $\mathit{VAL}\ m : \mathit{VAL}\ n : s$ in the *Just* case means that the argument of *fail* in the *Nothing* case must be $\mathit{VAL}\ n : s$ rather than just *s*. This observation gives our first defining equation for *fail*, and we continue as follows:

$$
\begin{array}{ll}
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
 & \qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
 & \qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
 & \quad \mathit{Nothing} \rightarrow \mathit{fail}\ s \\
= & \quad \{\ \text{define: } \mathit{fail}\ (\mathit{VAL}\ n : s) = \mathit{fail}\ s\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
 & \qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
 & \qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{fail}\ (\mathit{VAL}\ n : s)
\end{array}
$$

$$
\begin{array}{ll}
 & \quad \mathit{Nothing} \to \mathit{fail}\ s \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ (\mathit{comp}'\ y\ (\mathit{ADD}\ c))\ (\mathit{VAL}\ n : s) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ s \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{ADD}\ c))\ s
\end{array}
$$

Finally, we consider the inductive case for *Catch x h*. For this case, getting to the application of the induction hypothesis for *h* is straightforward:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Catch}\ x\ h)\ c)\ s \\
= & \quad \{\ \text{specification (7)}\ \} \\
 & \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
 & \quad \mathit{Nothing} \to \mathbf{case}\ \mathit{eval}\ h\ \mathbf{of} \\
 & \qquad\qquad\qquad \mathit{Just}\ m \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ m : s) \\
 & \qquad\qquad\qquad \mathit{Nothing} \to \mathit{fail}\ s \\
= & \quad \{\ \text{induction hypothesis for } h\ \} \\
 & \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
 & \quad \mathit{Nothing} \to \mathit{exec}\ (\mathit{comp}'\ h\ c)\ s
\end{array}
$$

Now we are in a similar position to the calculation for *Add*, i.e. the *Nothing* case does not match the form of specification (7). In order for this to match, the *Nothing* case needs to be of the form *fail s*. That is, we need to solve the equation:

$$\mathit{fail}\ s = \mathit{exec}\ (\mathit{comp}'\ h\ c)\ s$$

Note that we can't simply use this equation as a definition for *fail*, because *h* and *c* are unbound in the body of the equation. As we only have the stack argument *s* at our disposal, one approach would be to modify this argument. In particular, we could assume that the handler *h* and its code continuation *c* are provided on the stack by means of a new constructor *HAN* in the *Elem* datatype, and define a new equation for *fail* as follows:

$$\mathit{fail}\ (\mathit{HAN}\ h\ c : s) = \mathit{exec}\ (\mathit{comp}'\ h\ c)\ s$$

However, this approach would result in the source language expression *h* being stored on the stack by the compiler, whereas it is natural to expect all expressions in the source language to be compiled away. An alternative approach that avoids this problem is to assume that the entire handler code $\mathit{comp}'\ h\ c$ is provided on the stack by means of a *HAN* constructor with a single argument. In particular, if we define

$$\mathit{fail}\ (\mathit{HAN}\ c' : s) = \mathit{exec}\ c'\ s$$

then by taking $c' = \mathit{comp}'\ h\ c$ we obtain the equation

$$\mathit{fail}\ (\mathit{HAN}\ (\mathit{comp}'\ h\ c) : s) = \mathit{exec}\ (\mathit{comp}'\ h\ c)\ s$$

which is now close to the form that we need. Based upon this idea, we resume the calculation, during which we introduce a code constructor *UNMARK* to bring the stack argument

in the *Just* case into the form that we need to apply the induction hypothesis for x by removing the unused handler element, a process known as 'unmarking' the stack:

$$
\begin{array}{ll}
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
 & \quad \mathit{Nothing} \to \mathit{exec}\ (\mathit{comp}'\ h\ c)\ s \\
= & \quad \{\ \text{define: } \mathit{fail}\ (\mathit{HAN}\ c' : s) = \mathit{exec}\ c'\ s\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{HAN}\ (\mathit{comp}'\ h\ c) : s) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ \_ : s) = \mathit{exec}\ c\ (\mathit{VAL}\ n : s)\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ (\mathit{comp}'\ h\ c) : s) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{HAN}\ (\mathit{comp}'\ h\ c) : s) \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{UNMARK}\ c))\ (\mathit{HAN}\ (\mathit{comp}'\ h\ c) : s) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{MARK}\ c'\ c)\ s = \mathit{exec}\ c\ (\mathit{HAN}\ c' : s)\ \} \\
 & \mathit{exec}\ (\mathit{MARK}\ (\mathit{comp}'\ h\ c)\ (\mathit{comp}'\ x\ (\mathit{UNMARK}\ c)))\ s
\end{array}
$$

The final step above introduces a code constructor *MARK* that encapsulates the process of pushing handler code onto the stack, similarly to the *PUSH* constructor for values.

We complete the development of our compiler by considering the top-level compilation function $\mathit{comp} :: \mathit{Expr} \to \mathit{Code}$. For arithmetic expressions, the desired behaviour of *comp* was specified by the equation $\mathit{exec}\ (\mathit{comp}\ e)\ s = \mathit{eval}\ e : s$. Based upon our experience with comp', in the presence of exceptions we refine this equation as follows:

$$
\begin{array}{lcll}
\mathit{exec}\ (\mathit{comp}\ e)\ s & = & \textbf{case}\ \mathit{eval}\ e\ \textbf{of} & (8) \\
 & & \quad \mathit{Just}\ n \quad \to \mathit{VAL}\ n : s & \\
 & & \quad \mathit{Nothing} \to \mathit{fail}\ s &
\end{array}
$$

To calculate a definition for *comp* from this equation, we aim to rewrite the left-hand side $\mathit{exec}\ (\mathit{comp}\ e)\ s$ into the form $\mathit{exec}\ c'\ s$ for some code c, and hence define $\mathit{comp}\ e = c'$. The calculation proceeds in the same manner as the previous section, during which we introduce a new code constructor *HALT* to bring the stack argument in the *Just* case into the form that we need to apply the specification for comp':

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}\ e)\ s \\
= & \quad \{\ \text{specification (8)}\ \} \\
 & \textbf{case}\ \mathit{eval}\ e\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{VAL}\ n : s \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ s \\
= & \quad \{\ \text{define: } \mathit{exec}\ \mathit{HALT}\ s = s\ \} \\
 & \textbf{case}\ \mathit{eval}\ e\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ \mathit{HALT}\ (\mathit{VAL}\ n : s) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ s \\
= & \quad \{\ \text{specification (7)}\ \} \\
 & \mathit{exec}\ (\mathit{comp}'\ e\ \mathit{HALT})\ s
\end{array}
$$

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

Target language:

$$\begin{array}{l}\textbf{data}\ \mathit{Code} = \mathit{HALT} \mid \mathit{PUSH}\ \mathit{Int}\ \mathit{Code} \mid \mathit{ADD}\ \mathit{Code} \mid \\ \qquad\qquad \mathit{FAIL} \mid \mathit{MARK}\ \mathit{Code}\ \mathit{Code} \mid \mathit{UNMARK}\ \mathit{Code}\end{array}$$

Compiler:

$$\begin{array}{ll}\mathit{comp} & :: \mathit{Expr} \to \mathit{Code} \\ \mathit{comp}\ e & = \mathit{comp}'\ e\ \mathit{HALT} \\ \mathit{comp}' & :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code} \\ \mathit{comp}'\ (\mathit{Val}\ n)\ c & = \mathit{PUSH}\ n\ c \\ \mathit{comp}'\ (\mathit{Add}\ x\ y)\ c & = \mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{ADD}\ c)) \\ \mathit{comp}'\ \mathit{Throw}\ c & = \mathit{FAIL} \\ \mathit{comp}'\ (\mathit{Catch}\ x\ h)\ c & = \mathit{MARK}\ (\mathit{comp}'\ h\ c)\ (\mathit{comp}'\ x\ (\mathit{UNMARK}\ c))\end{array}$$

Virtual machine:

$$\begin{array}{ll}\textbf{type}\ \mathit{Stack} & = [\mathit{Elem}] \\ \textbf{data}\ \mathit{Elem} & = \mathit{VAL}\ \mathit{Int} \mid \mathit{HAN}\ \mathit{Code} \\ \mathit{exec} & :: \mathit{Code} \to \mathit{Stack} \to \mathit{Stack} \\ \mathit{exec}\ \mathit{HALT}\ s & = s \\ \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s & = \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\ \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) & = \mathit{exec}\ c\ (\mathit{VAL}\ (n+m) : s) \\ \mathit{exec}\ \mathit{FAIL}\ s & = \mathit{fail}\ s \\ \mathit{exec}\ (\mathit{MARK}\ c'\ c)\ s & = \mathit{exec}\ c\ (\mathit{HAN}\ c' : s) \\ \mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ \_ : s) & = \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\ \mathit{fail} & :: \mathit{Stack} \to \mathit{Stack} \\ \mathit{fail}\ [\,] & = [\,] \\ \mathit{fail}\ (\mathit{VAL}\ n : s) & = \mathit{fail}\ s \\ \mathit{fail}\ (\mathit{HAN}\ c : s) & = \mathit{exec}\ c\ s\end{array}$$

Note that the two equations that we derived for the function *fail* do not yield a total definition, because there is no equation for empty stack. In the definition above, we have chosen to define $\mathit{fail}\ [\,] = [\,]$ in this case. In principle any choice would be fine, because the calculation does not depend it. However, for our choice if we instantiate $s = [\,]$ in specification (7) we then obtain the empty stack as the result when evaluation fails, which is a natural representation of an uncaught exception.

Note also that *exec* and *fail* are defined mutually recursively, and correspond to two execution modes for the virtual machine, the first for when execution is proceeding normally, and the second for when an exception has been thrown and a handler is being sought. In the latter case, the function *fail* implements the process known as 'unwinding' the stack, in which elements are popped from the stack until an exception handler is found, at which point execution then transfers to the handler code.

The compiler derived above is essentially the same as that presented in (Hutton & Wright, 2004), except that our compiler uses code continuations, and has been derived

directly from a specification of its correctness using simple equational reasoning, with all the compilation machinery falling naturally out of the calculation process.

3.2 Second approach: two code continuations

The approach presented in the previous section started with the same type for $comp'$ as for simple arithmetic expressions in section 2. In the context of exceptions, however, this approach made it more difficult to formulate the specification for $comp'$, as the type for the function does not provide an explicit mechanism for dealing with failure.

In this second approach we modify the type for $comp'$ to reflect the addition of exceptions to the language. In particular, just as the evaluation function *eval* returns a *Maybe* type to represent the two forms of results that can be produced, we refine the type of *comp* to take two code continuations as arguments rather than just one:

$$comp' :: Expr \to Code \to Code \to Code$$

The initial type for stacks is unchanged:

$$\textbf{type}\ Stack = [Elem]$$
$$\textbf{data}\ Elem = VAL\ Int$$

The idea behind the new type for $comp'$ is that the first continuation argument will be used if evaluation is successful and the second if evaluation fails, an approach sometimes called *double-barrelled* continuations (Thielecke, 2002). This intuition is formalised in the following specification for the intended behaviour $comp'$, in which the arguments s and f are the success and failure code continuations, and k is the stack:

$$\begin{array}{lcl} exec\ (comp'\ e\ s\ f)\ k & = & \textbf{case}\ eval\ e\ \textbf{of} \\ & & \quad Just\ n \to exec\ s\ (VAL\ n : k) \\ & & \quad Nothing \to exec\ f\ k \end{array} \tag{9}$$

From this specification, we calculate the definition for $comp'$ by constructive induction on the expression e. The cases for *Val* and *Throw* are again trivial:

$$\begin{array}{ll} & exec\ (comp'\ (Val\ n)\ s\ f)\ k \\ = & \quad \{\ \text{specification (9)}\ \} \\ & exec\ s\ (VAL\ n : k) \\ = & \quad \{\ \text{define: } exec\ (PUSH\ n\ s)\ k = exec\ s\ (VAL\ n : k)\ \} \\ & exec\ (PUSH\ n\ s)\ k \end{array}$$

and

$$\begin{array}{ll} & exec\ (comp'\ Throw\ s\ f)\ k \\ = & \quad \{\ \text{specification (9)}\ \} \\ & exec\ f\ k \end{array}$$

Because the failure continuation is built into $comp'$, the calculation for *Catch* now becomes much simpler. In particular, we don't have to manipulate the *Nothing* case into a form that uses *fail*, as the execution of any code sequence with a stack of the appropriate shape suffices. Hence, we can immediately apply the induction hypotheses:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Catch}\ x\ h)\ s\ f)\ k \\
= & \quad \{\ \text{specification (9)}\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ s\ (\mathit{VAL}\ n : k) \\
 & \quad \mathit{Nothing} \to \textbf{case}\ h\ \textbf{of} \\
 & \qquad\qquad \mathit{Just}\ m \quad \to \mathit{exec}\ s\ (\mathit{VAL}\ m : k) \\
 & \qquad\qquad \mathit{Nothing} \to \mathit{exec}\ f\ k \\
= & \quad \{\ \text{induction hypothesis for } h\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to s\ (\mathit{VAL}\ n : k) \\
 & \quad \mathit{Nothing} \to \mathit{exec}\ (\mathit{comp}'\ h\ s\ f)\ k \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{exec}\ (\mathit{comp}'\ x\ s\ (\mathit{comp}'\ h\ s\ f))\ k
\end{array}
$$

The calculation for *Add* also becomes simpler. However, we still need to bring the stack arguments into the right form for the induction hypotheses. As before, we introduce a code constructor *ADD* that does this for the *Just* case. Adjusting the stack argument for the *Nothing* case is now simpler compared to the calculation in section 3.1 as we may use any code sequence, for which purposes we introduce a *POP* constructor:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Add}\ x\ y)\ s\ f)\ k \\
= & \quad \{\ \text{specification (9)}\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
 & \qquad\qquad \mathit{Just}\ m \quad \to \mathit{exec}\ s\ (\mathit{VAL}\ (n+m) : k) \\
 & \qquad\qquad \mathit{Nothing} \to \mathit{exec}\ f\ k \\
 & \quad \mathit{Nothing} \to \mathit{exec}\ f\ k \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{ADD}\ s)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : k) = \mathit{exec}\ s\ (\mathit{VAL}\ (n+m) : k)\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
 & \qquad\qquad \mathit{Just}\ m \quad \to \mathit{exec}\ (\mathit{ADD}\ s)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : k) \\
 & \qquad\qquad \mathit{Nothing} \to \mathit{exec}\ f\ k \\
 & \quad \mathit{Nothing} \to \mathit{exec}\ f\ k \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{POP}\ f)\ (\mathit{VAL}\ \_ : k) = \mathit{exec}\ f\ k\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
 & \qquad\qquad \mathit{Just}\ m \quad \to \mathit{exec}\ (\mathit{ADD}\ s)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : k) \\
 & \qquad\qquad \mathit{Nothing} \to \mathit{exec}\ (\mathit{POP}\ f)\ (\mathit{VAL}\ n : k) \\
 & \quad \mathit{Nothing} \to \mathit{exec}\ f\ k \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{exec}\ (\mathit{comp}'\ y\ (\mathit{ADD}\ s)\ (\mathit{POP}\ f))\ (\mathit{VAL}\ n : k) \\
 & \quad \mathit{Nothing} \to \mathit{exec}\ f\ k \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{ADD}\ s)\ (\mathit{POP}\ f))\ f)\ k
\end{array}
$$

We complete the calculation by considering the top-level compilation function $comp :: Expr \to Code$. Starting from a specification of the desired behaviour,

$$\begin{array}{lcl} exec\ (comp\ e)\ k & = & \textbf{case}\ eval\ e\ \textbf{of} \\ & & \quad Just\ n \quad \to VAL\ n : k \\ & & \quad Nothing \to k \end{array} \tag{10}$$

we calculate a definition for *comp* as follows, during which we introduce a new code constructor *HALT* that is used in both the success and failure cases:

$$\begin{array}{ll} & exec\ (comp\ e)\ k \\ = & \quad \{\ \text{specification (10)}\ \} \\ & \textbf{case}\ eval\ e\ \textbf{of} \\ & \quad Just\ n \quad \to VAL\ n : k \\ & \quad Nothing \to k \\ = & \quad \{\ \text{define: } exec\ HALT\ k = k\ \} \\ & \textbf{case}\ eval\ e\ \textbf{of} \\ & \quad Just\ n \quad \to exec\ HALT\ (VAL\ n : k) \\ & \quad Nothing \to exec\ HALT\ k \\ = & \quad \{\ \text{specification (9)}\ \} \\ & exec\ (comp'\ HALT\ HALT)\ k \end{array}$$

We could also have introduced a special-purpose code constructor for the failure case, say $exec\ CRASH\ k = k$, but for our simple exception language it suffices to use *HALT* for both cases. However, for a more sophisticated source language in which we distinguish between different kinds of exceptions, making such a distinction may be important.

In summary, we have calculated the following definitions:

$$\begin{array}{ll} \textbf{data}\ Code & = HALT \mid PUSH\ Int\ Code \mid ADD\ Code \mid POP\ Code \\ comp & :: Expr \to Code \\ comp\ e & = comp'\ e\ HALT\ HALT \\ comp' & :: Expr \to Code \to Code \to Code \\ comp'\ (Val\ n)\ s\ f & = PUSH\ n\ s \\ comp'\ (Add\ x\ y)\ s\ f & = comp'\ x\ (comp'\ y\ (ADD\ s)\ (POP\ f))\ f \\ comp'\ Throw\ s\ f & = f \\ comp'\ (Catch\ x\ h)\ s\ f & = comp'\ x\ s\ (comp'\ h\ s\ f) \\ exec & :: Code \to Stack \to Stack \\ exec\ HALT\ k & = k \\ exec\ (PUSH\ n\ c)\ k & = exec\ c\ (VAL\ n : k) \\ exec\ (ADD\ c)\ (VAL\ n : VAL\ m : k) & = exec\ c\ (VAL\ (m+n) : k) \\ exec\ (POP\ c)\ (VAL\ \_ : k) & = exec\ c\ k \end{array}$$

3.3 Reflection

We conclude this section with some comments on the two approaches to calculating a compiler for exceptions, concerning scalability and partiality.

Scalability In the approach using a single code continuation, the partial specification for $comp'$ in terms of an undefined function *fail* means that additional effort is required to derive a definition for *fail*. However, the benefit of this approach is that we obtained a compiler that implements exceptions using the idea of stack unwinding by purely calculational methods, with all the required compilation techniques arising naturally during the calculation process, driven once again by the desire to apply the induction hypotheses. This approach scales well to more sophisticated languages as it does not require static knowledge about the scope in which an exception is thrown. In contrast, such static knowledge is exploited in the approach using two code continuations, in the form of the failure continuation. However, such knowledge is not available if we consider, for example, a higher-order language as we shall do in section 5.

We can also identify a third approach, which combines the benefits of the first two. This 'hybrid' approach is based upon a function $comp'$ with separate code continuations for success and failure as in the second approach, whose behaviour in the case when evaluation fails is specified in terms of an undefined function *fail* as in the first:

$$\begin{array}{lcl} exec\ (comp'\ e\ s\ f)\ k & = & \textbf{case}\ eval\ e\ \textbf{of} \\ & & \quad Just\ n \quad \to exec\ s\ (VAL\ n : k) \\ & & \quad Nothing \to fail\ f\ k \end{array}$$

The compiler that results from this specification avoids the explicit cleaning up of the stack with *POP* instructions of the second approach, but instead relies on stack unwinding in a similar manner to the first. In the course of the calculation a new stack element constructor similar to *HAN* is introduced but without a handler argument, which is not necessary as we have an explicit failure code continuation as part of $comp'$.

Partiality The calculations in this section followed the general approach from section 2. However, we used two additional techniques to make the approach more powerful:

- We used a *partial specification* for the $comp'$ function. The specification for $comp'$ is effectively the induction hypothesis for the calculation of its definition. For the simple expression language in section 2, determining the appropriate induction hypothesis was straightforward. However, the more sophisticated the source language becomes, the more difficult this becomes. The technique of using a partial specification leaves some of the details of the induction hypothesis open and allows us to derive these during the calculation itself. Equally importantly, this technique also allows us to derive the definition for auxiliary functions such as *fail*.
- We used a *partial definition* for the *Stack* type. This technique is crucial for more sophisticated languages. While our recipe is targetted at deriving stack machines, the actual details of the stack type are difficult to anticipate as they will only become apparent as we calculate the definition for $comp'$.

Both of the above techniques are measures to reduce the amount of required prior knowledge of the result. The calculations in this section start with very few assumptions about the final outcome. Indeed, these assumptions, expressed in the specification for $comp'$, can be summarised as "if evaluation is successful put the resulting value on the

stack and continue execution, otherwise do something else". The calculation process then fills out the details of how this is achieved and what "something else" means.

4 State

In this section we extend our source language further, with primitives for reading and writing a mutable state that stores an integer value:

$$\textbf{type}\ \mathit{State} = \mathit{Int}$$
$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Throw} \mid \mathit{Catch}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Get} \mid \mathit{Put}\ \mathit{Expr}\ \mathit{Expr}$$

Informally, *Get* returns the current value of the state, while *Put x y* sets the state to the value of the expression x and then behaves as the expression y. Having *Put* take two arguments in this manner avoids the need for a separate sequencing operator.

The addition of state is particularly is interesting as it interacts with the exception handling mechanism of the language. In particular, there are two different ways of combining exceptions and state from a semantic perspective, depending on whether the current state is retained or discarded when an exception is thrown. If the state is retained then an exception handler sees the state as it was when the exception was thrown. If the state is discarded then the handler sees the state as it was when the enclosing *Catch* was entered. We refer to the former case as *global state*, and the latter as *local state* (Day & Hutton, 2013).

We shall calculate a compiler for each of the two semantics. In the previous section we considered one semantics for exceptions but varied the specification for the compiler, which resulted in different calculations and different compilers. This time we shall start with different semantics but use essentially the same specification for the compiler. Because this specification is in terms of two different semantics, we will get different outcomes. Our calculations extend the 'one continuation' approach from section 3.1, but we could just as well use any other approach from section 3.

4.1 Global State

The *global state* semantics retains the current state in case of an exception, which is reflected in the new type for the evaluation function as follows:

$$\mathit{eval} :: \mathit{Expr} \to \mathit{State} \to (\mathit{Maybe}\ \mathit{Int}, \mathit{State})$$

That is, no matter whether an exception is thrown or not, the function *eval* always returns a new state. Using this type, the evaluation function from section 3.1 can be refined to take account of state by simply threading through the current state:

$$\begin{array}{lll}
\mathit{eval}\ (\mathit{Val}\ n)\ s & = (\mathit{Just}\ n, s) \\
\mathit{eval}\ (\mathit{Add}\ x\ y)\ s & = \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
 & \quad (\mathit{Just}\ n, s') \to \textbf{case}\ \mathit{eval}\ y\ s'\ \textbf{of} \\
 & \qquad (\mathit{Just}\ m, s'') \to (\mathit{Just}\ (n+m), s'') \\
 & \qquad (\mathit{Nothing}, s'') \to (\mathit{Nothing}, s'') \\
 & \quad (\mathit{Nothing}, s') \to (\mathit{Nothing}, s') \\
\mathit{eval}\ \mathit{Throw}\ s & = (\mathit{Nothing}, s)
\end{array}$$

$$
\begin{array}{lll}
\mathit{eval}\ (\mathit{Catch}\ x\ h)\ s = \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} & & \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Just}\ n, s') & \to (\mathit{Just}\ n, s') & \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Nothing}, s') & \to \mathit{eval}\ h\ s' & \\
\mathit{eval}\ \mathit{Get}\ s & = (\mathit{Just}\ s, s) & \\
\mathit{eval}\ (\mathit{Put}\ x\ y)\ s & = \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} & \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Just}\ n, s') & \to \mathit{eval}\ y\ n & \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Nothing}, s') & \to (\mathit{Nothing}, s') &
\end{array}
$$

Note that in the case for *Catch*, when the handler h is invoked it uses the state s' from when the exception was thrown, which formalises our earlier intuition for global state. Extending the specification of the compilation function comp' from section 3.1 to state is straightforward. First of all, the type for comp' itself remains the same,

$$\mathit{comp}' :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code}$$

but we refine the type of the execution function *exec* to transform pairs comprising a stack and a state, which we term *configurations*, rather than just transforming a stack:

$$
\begin{array}{l}
\mathit{exec} :: \mathit{Code} \to \mathit{Conf} \to \mathit{Conf} \\
\textbf{type}\ \mathit{Conf} = (\mathit{Stack}, \mathit{State})
\end{array}
$$

More generally, the same principle also applies to semantics that utilise environments or heaps: all additional data structures required for the semantics are combined with the stack to form a configuration of type *Conf*, and the execution function *exec* transform such configurations. The previous type for *exec* was just the special case where no additional data structures were required. The initial type for stacks is the same as before:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Stack} = [\mathit{Elem}] \\
\textbf{data}\ \mathit{Elem} = \mathit{VAL}\ \mathit{Int}
\end{array}
$$

The specification for the desired behaviour of comp' is similar to the case without state, except that we now have to thread through the current state:

$$
\begin{array}{lll}
\mathit{exec}\ (\mathit{comp}'\ e\ c)\ (k, s) & = & \textbf{case}\ \mathit{eval}\ e\ s\ \textbf{of} \\
& & \qquad (\mathit{Just}\ n, s') \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : k, s') \\
& & \qquad (\mathit{Nothing}, s') \to \mathit{fail}\ (k, s')
\end{array}
\tag{11}
$$

This is again a partial specification in terms of an as yet undefined function *fail* for the case when evaluation fails, this time of type $\mathit{Conf} \to \mathit{Conf}$. We now calculate a definition for comp' from this specification by constructive induction on e, during which we also derive a definition for *fail*. The cases for *Val* and *Throw* are as usual trivial:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Val}\ n)\ c)\ (k, s) \\
= & \quad \{\ \text{specification (11)}\ \} \\
 & \mathit{exec}\ c\ (\mathit{VAL}\ n : k, s) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (k, s) = \mathit{exec}\ c\ (\mathit{VAL}\ n : k, s)\ \} \\
 & \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (k, s)
\end{array}
$$

and

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ \mathit{Throw}\ c)\ (k,s) \\
= & \quad \{\ \text{specification (11)}\ \} \\
 & \mathit{fail}\ (k,s) \\
= & \quad \{\ \text{define: } \mathit{exec}\ \mathit{FAIL}\ (k,s) = \mathit{fail}\ (k,s)\ \} \\
 & \mathit{exec}\ \mathit{FAIL}\ (k,s)
\end{array}
$$

The cases for *Add* and *Catch* proceed along similar lines to section 3.1:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Add}\ x\ y)\ c)\ (k,s) \\
= & \quad \{\ \text{specification (11)}\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
 & \quad (\mathit{Just}\ n,s') \quad \to \textbf{case}\ \mathit{eval}\ y\ s'\ \textbf{of} \\
 & \qquad\qquad\qquad (\mathit{Just}\ m,s'') \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ (n+m) : k,s'') \\
 & \qquad\qquad\qquad (\mathit{Nothing},s'') \to \mathit{fail}\ (k,s'') \\
 & \quad (\mathit{Nothing},s') \to \mathit{fail}\ (k,s') \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : k,s'') = \mathit{exec}\ c\ (\mathit{VAL}\ (n+m) : k,s'')\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
 & \quad (\mathit{Just}\ n,s') \quad \to \textbf{case}\ \mathit{eval}\ y\ s'\ \textbf{of} \\
 & \qquad\qquad\qquad (\mathit{Just}\ m,s'') \quad \to \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : k,s'') \\
 & \qquad\qquad\qquad (\mathit{Nothing},s'') \to \mathit{fail}\ (k,s'') \\
 & \quad (\mathit{Nothing},s') \to \mathit{fail}\ (k,s') \\
= & \quad \{\ \text{define: } \mathit{fail}\ (\mathit{VAL}\ n : k,s'') = \mathit{fail}\ (k,s'')\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
 & \quad (\mathit{Just}\ n,s') \quad \to \textbf{case}\ \mathit{eval}\ y\ s'\ \textbf{of} \\
 & \qquad\qquad\qquad (\mathit{Just}\ m,s'') \quad \to \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : k,s'') \\
 & \qquad\qquad\qquad (\mathit{Nothing},s'') \to \mathit{fail}\ (\mathit{VAL}\ n : k,s'') \\
 & \quad (\mathit{Nothing},s') \to \mathit{fail}\ (k,s') \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
 & \quad (\mathit{Just}\ n,s') \quad \to \mathit{exec}\ (\mathit{comp}'\ y\ (\mathit{ADD}\ c))\ (\mathit{VAL}\ n : k,s') \\
 & \quad (\mathit{Nothing},s') \to \mathit{fail}\ (k,s') \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{ADD}\ c)))\ (k,s)
\end{array}
$$

and

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Catch}\ x\ h)\ c)\ (k,s) \\
= & \quad \{\ \text{specification (11)}\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
 & \quad (\mathit{Just}\ n,s') \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : k,s') \\
 & \quad (\mathit{Nothing},s') \to \textbf{case}\ \mathit{eval}\ h\ s'\ \textbf{of} \\
 & \qquad\qquad\qquad (\mathit{Just}\ m,s'') \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ m : k,s'') \\
 & \qquad\qquad\qquad (\mathit{Nothing},s'') \to \mathit{fail}\ (k,s'') \\
= & \quad \{\ \text{induction hypothesis for } h\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
 & \quad (\mathit{Just}\ n,s') \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : k,s') \\
 & \quad (\mathit{Nothing},s') \to \mathit{exec}\ (\mathit{comp}'\ h\ c)\ (k,s')
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ define: } \mathit{fail}\ (\mathit{HAN}\ c' : k, s') = \mathit{exec}\ c'\ (k, s') \ \} \\
& \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
& \quad (\mathit{Just}\ n, s') \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ n : k, s') \\
& \quad (\mathit{Nothing}, s') \to \mathit{fail}\ (\mathit{HAN}\ (\mathit{comp}'\ h\ c) : k, s') \\
= & \{ \text{ define: } \mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ \_ : k, s') = \mathit{exex}\ c\ (\mathit{VAL}\ n : k, s') \ \} \\
& \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
& \quad (\mathit{Just}\ n, s') \quad \to \mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ (\mathit{comp}'\ h\ c) : k, s') \\
& \quad (\mathit{Nothing}, s') \to \mathit{fail}\ (\mathit{HAN}\ (\mathit{comp}'\ h\ c) : k, s') \\
= & \{ \text{ induction hypothesis for } x \ \} \\
& \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{UNMARK}\ c))\ (\mathit{HAN}\ (\mathit{comp}'\ h\ c) : k, s) \\
= & \{ \text{ define: } \mathit{exec}\ (\mathit{MARK}\ c''\ c')\ (k, s) = \mathit{exex}\ c'\ (\mathit{HAN}\ c'' : k, s) \ \} \\
& \mathit{exex}\ (\mathit{MARK}\ (\mathit{comp}'\ h\ c)\ (\mathit{comp}'\ x\ (\mathit{UNMARK}\ c)))\ (k, s)
\end{array}
$$

Finally, we come to the calculations for the new language features. The case for *Get* is straightforward, and introduces a code constructor *LOAD* that encapsulates the process of pushing the current value of the state onto the top of the stack:

$$
\begin{array}{ll}
& \mathit{exec}\ (\mathit{comp}'\ \mathit{Get}\ c)\ (k, s) \\
= & \{ \text{ specification (11) } \} \\
& \mathit{exec}\ c\ (\mathit{VAL}\ s : k, s) \\
= & \{ \text{ define: } \mathit{exec}\ (\mathit{LOAD}\ c)\ (k, s) = \mathit{exec}\ c\ (\mathit{VAL}\ s : k, s) \ \} \\
& \mathit{exec}\ (\mathit{LOAD}\ c)\ (k, s)
\end{array}
$$

The case for *Put* is more interesting. However, it follows a common pattern that we have seen a number of times now: we introduce a code constructor *SAVE* to bring the stack argument into the form that we need to apply an induction hypothesis, in this case by popping the top value from the stack and setting the state to this value:

$$
\begin{array}{ll}
& \mathit{exec}\ (\mathit{comp}'\ (\mathit{Put}\ x\ y)\ c)\ (k, s) \\
= & \{ \text{ specification (11) } \} \\
& \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
& \quad (\mathit{Just}\ n, s') \quad \to \textbf{case}\ \mathit{eval}\ y\ n\ \textbf{of} \\
& \qquad\qquad\qquad\qquad (\mathit{Just}\ m, s'') \quad \to \mathit{exec}\ c\ (\mathit{VAL}\ m : k, s'') \\
& \qquad\qquad\qquad\qquad (\mathit{Nothing}, s'') \to \mathit{fail}\ (k, s'') \\
& \quad (\mathit{Nothing}, s') \to \mathit{fail}\ (k, s') \\
= & \{ \text{ induction hypothesis for } y \ \} \\
& \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
& \quad (\mathit{Just}\ n, s') \quad \to \mathit{exec}\ (\mathit{comp}'\ y\ c)\ (k, n) \\
& \quad (\mathit{Nothing}, s') \to \mathit{fail}\ (k, s') \\
= & \{ \text{ define: } \mathit{exec}\ (\mathit{SAVE}\ c')\ (\mathit{VAL}\ n : k, s') = \mathit{exec}\ c'\ (k, n) \ \} \\
& \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
& \quad (\mathit{Just}\ n, s') \quad \to \mathit{exec}\ (\mathit{SAVE}\ (\mathit{comp}'\ y\ c))\ (\mathit{VAL}\ n : k, s') \\
& \quad (\mathit{Nothing}, s') \to \mathit{fail}\ (k, s') \\
= & \{ \text{ induction hypothesis for } x \ \} \\
& \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{SAVE}\ (\mathit{comp}'\ y\ c)))\ (k, s)
\end{array}
$$

In summary, we have calculated the definitions shown below. As in section 3.1, we make *fail* into a total function by adding an equation for the case when the stack is empty, and define the top-level compilation function *comp* by simply applying $comp'$ to *HALT*.

Target language:

$$\begin{aligned}&\textbf{data}\ Code = HALT \mid PUSH\ Int\ Code \mid ADD\ Code \mid\\ &\qquad\qquad FAIL \mid MARK\ Code\ Code \mid UNMARK\ Code \mid\\ &\qquad\qquad LOAD\ Code \mid SAVE\ Code\end{aligned}$$

Compiler:

$$\begin{aligned}
&comp && :: Expr \to Code\\
&comp\ e && = comp'\ e\ HALT\\
&comp' && :: Expr \to Code \to Code\\
&comp'\ (Val\ n)\ c && = PUSH\ n\ c\\
&comp'\ (Add\ x\ y)\ c && = comp'\ x\ (comp'\ y\ (ADD\ c))\\
&comp'\ Throw\ c && = FAIL\\
&comp'\ (Catch\ x\ h)\ c && = MARK\ (comp'\ h\ c)\ (comp'\ s\ (UNMARK\ c))\\
&comp'\ Get\ c && = LOAD\ c\\
&comp'\ (Put\ x\ y)\ c && = comp'\ x\ (SAVE\ (comp'\ y\ c))
\end{aligned}$$

Virtual machine:

$$\begin{aligned}
&\textbf{data}\ Elem && = VAL\ Int \mid HAN\ Code\\
&exec && :: Code \to Conf \to Conf\\
&exec\ HALT\ (k,s) && = (k,s)\\
&exec\ (PUSH\ n\ c)\ (k,s) && = exec\ c\ (VAL\ n : k,s)\\
&exec\ (ADD\ c)\ (VAL\ m : VAL\ n : k,s) && = exec\ c\ (VAL\ (n+m) : k,s)\\
&exec\ FAIL\ (k,s) && = fail\ (k,s)\\
&exec\ (MARK\ h\ c)\ (k,s) && = exec\ c\ (HAN\ h : k,s)\\
&exec\ (UNMARK\ c)\ (VAL\ n : HAN\ _ : k,s) && = exec\ c\ (VAL\ n : k,s)\\
&exec\ (LOAD\ c)\ (k,s) && = exec\ c\ (VAL\ s : k,s)\\
&exec\ (SAVE\ c)\ (VAL\ n : k,s) && = exec\ c\ (k,n)\\
&fail && :: Conf \to Conf\\
&fail\ ([\,],s) && = ([\,],s)\\
&fail\ (VAL\ n : k,s) && = fail\ (k,s)\\
&fail\ (HAN\ h : k,s) && = exec\ h\ (k,s)
\end{aligned}$$

4.2 Local State

We now consider the *local* approach to combining exceptions and state, in which the current state is discarded when an exception is thrown. This idea is reflected in the type for evaluation by moving the output state 'inside' the *Maybe* type:

$$eval :: Expr \to State \to Maybe\ (Int, State)$$

That is, if evaluation succeeds then *eval* returns an integer value and a new state, and if an exception is thrown it returns *Nothing*. The definition for *eval* is similar to the previous section except there is now no state to propagate when evaluation fails, and in the case for *Catch* the handler uses the state from when the catch was entered:

$$
\begin{array}{lll}
\mathit{eval}\ (\mathit{Val}\ n)\ s & = \mathit{Just}\ (n,s) \\
\mathit{eval}\ (\mathit{Add}\ x\ y)\ s & = \mathbf{case}\ \mathit{eval}\ x\ s\ \mathbf{of} \\
& \qquad \mathit{Just}\ (n,s') \to \mathbf{case}\ \mathit{eval}\ y\ s'\ \mathbf{of} \\
& \qquad\qquad \mathit{Just}\ (m,s'') \to \mathit{Just}\ (n+m,s'') \\
& \qquad\qquad \mathit{Nothing} \quad \to \mathit{Nothing} \\
& \qquad \mathit{Nothing} \quad \to \mathit{Nothing} \\
\mathit{eval}\ \mathit{Throw}\ s & = \mathit{Nothing} \\
\mathit{eval}\ (\mathit{Catch}\ x\ h)\ s & = \mathbf{case}\ \mathit{eval}\ x\ s\ \mathbf{of} \\
& \qquad \mathit{Just}\ (n,s') \to \mathit{Just}\ (n,s') \\
& \qquad \mathit{Nothing} \quad \to \mathit{eval}\ h\ s \\
\mathit{eval}\ \mathit{Get}\ s & = \mathit{Just}\ (s,s) \\
\mathit{eval}\ (\mathit{Put}\ x\ y)\ s & = \mathbf{case}\ \mathit{eval}\ x\ s\ \mathbf{of} \\
& \qquad \mathit{Just}\ (n,s') \to \mathit{eval}\ y\ n \\
& \qquad \mathit{Nothing} \quad \to \mathit{Nothing}
\end{array}
$$

For the purposes of the derivation of the compilation function $\mathit{comp}' :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code}$ we use the same types as for the global state semantics:

$$
\begin{array}{l}
\mathit{exec} :: \mathit{Code} \to \mathit{Conf} \to \mathit{Conf} \\
\mathbf{type}\ \mathit{Conf} = (\mathit{Stack}, \mathit{State}) \\
\mathbf{type}\ \mathit{Stack} = [\mathit{Elem}] \\
\mathbf{data}\ \mathit{Elem} = \mathit{VAL}\ \mathit{Int}
\end{array}
$$

The specification for the desired behaviour of comp' is essentially the same as for global state, except that when evaluation fails we no longer have an output state to consider and hence the function *fail* only takes a stack as argument:

$$
\begin{array}{lll}
\mathit{exec}\ (\mathit{comp}'\ e\ c)\ (k,s) & = & \mathbf{case}\ \mathit{eval}\ e\ s\ \mathbf{of} \\
& & \qquad \mathit{Just}\ (n,s') \to \mathit{exec}\ c\ (\mathit{VAL}\ n : k, s') \\
& & \qquad \mathit{Nothing} \quad \to \mathit{fail}\ k
\end{array} \tag{12}
$$

However, to ensure type correctness of the specification, *fail* must still return a configuration, i.e. $\mathit{fail} :: \mathit{Stack} \to \mathit{Conf}$. An alternative would be to supply the input state s as an argument to *fail*, which is a valid choice that would lead to a different compiler. We start the derivation for comp' with the trivial cases for *Val n*, *Throw* and *Get*:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Val}\ n)\ c)\ (k,s) \\
= & \quad \{\ \text{specification (12)}\ \} \\
 & \mathit{exec}\ c\ (\mathit{VAL}\ n : k, s) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (k,s) = \mathit{exec}\ c\ (\mathit{VAL}\ n : k, s)\ \} \\
 & \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (k,s)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ \mathit{Throw}\ c)\ (k,s) \\
= & \quad \{\ \text{specification (12)}\ \} \\
 & \mathit{fail}\ k \\
= & \quad \{\ \text{define: } \mathit{exec}\ \mathit{FAIL}\ (k,s) = \mathit{fail}\ k\ \} \\
 & \mathit{exec}\ \mathit{FAIL}\ (k,s)
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ \mathit{Get}\ c)\ (k,s) \\
= & \quad \{\ \text{specification (12)}\ \} \\
 & \mathit{exec}\ c\ (\mathit{VAL}\ s : k, s) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{LOAD}\ c)\ (k,s) = \mathit{exec}\ c\ (\mathit{VAL}\ s : k, s)\ \} \\
 & \mathit{exec}\ (\mathit{LOAD}\ c)\ (k,s)
\end{array}
$$

The case for *Add* follows the now familiar pattern:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Add}\ x\ y)\ c)\ (k,s) \\
= & \quad \{\ \text{specification (12)}\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
 & \quad \mathit{Just}\ (n,s') \to \textbf{case}\ \mathit{eval}\ y\ s'\ \textbf{of} \\
 & \qquad\qquad \mathit{Just}\ (m,s'') \to \mathit{exec}\ c\ (\mathit{VAL}\ (n+m) : k, s'') \\
 & \qquad\qquad \mathit{Nothing} \quad\ \to \mathit{fail}\ k \\
 & \quad \mathit{Nothing} \quad\ \to \mathit{fail}\ k \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : k, s'') = \mathit{exec}\ c\ (\mathit{VAL}\ (n+m) : k, s'')\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
 & \quad \mathit{Just}\ (n,s') \to \textbf{case}\ \mathit{eval}\ y\ s'\ \textbf{of} \\
 & \qquad\qquad \mathit{Just}\ (m,s'') \to \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : k, s'') \\
 & \qquad\qquad \mathit{Nothing} \quad\ \to \mathit{fail}\ k \\
 & \quad \mathit{Nothing} \quad\ \to \mathit{fail}\ k \\
= & \quad \{\ \text{define: } \mathit{fail}\ (\mathit{VAL}\ n : k) = \mathit{fail}\ k\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
 & \quad \mathit{Just}\ (n,s') \to \textbf{case}\ \mathit{eval}\ y\ s'\ \textbf{of} \\
 & \qquad\qquad \mathit{Just}\ (m,s'') \to \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : k, s'') \\
 & \qquad\qquad \mathit{Nothing} \quad\ \to \mathit{fail}\ (\mathit{VAL}\ n : k) \\
 & \quad \mathit{Nothing} \quad\ \to \mathit{fail}\ k \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ s\ \textbf{of} \\
 & \quad \mathit{Just}\ (n,s') \to \mathit{exec}\ (\mathit{comp}'\ y\ (\mathit{ADD}\ c))\ (\mathit{VAL}\ n : k, s') \\
 & \quad \mathit{Nothing} \quad\ \to \mathit{fail}\ k \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{ADD}\ c)))\ (k,s)
\end{array}
$$

The case for *Catch* is more interesting this time. In the calculation for the global state semantics it was straightforward to bring the configuration arguments into the right form to apply the induction hypotheses. With local state, however, when an exception handler is invoked we require access to the state that was in place when the enclosing *Catch* was entered, which information we communicate via the stack:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Catch}\ x\ h)\ c)\ (k,s) \\
= & \quad \{\ \text{specification (12)}\ \} \\
 & \mathbf{case}\ \mathit{eval}\ x\ s\ \mathbf{of} \\
 & \quad \mathit{Just}\ (n,s') \to \mathit{exec}\ c\ (\mathit{VAL}\ n : k, s') \\
 & \quad \mathit{Nothing} \quad \to \mathbf{case}\ \mathit{eval}\ h\ s\ \mathbf{of} \\
 & \qquad\qquad\qquad \mathit{Just}\ (m,s'') \to \mathit{exec}\ c\ (\mathit{VAL}\ m : k, s'') \\
 & \qquad\qquad\qquad \mathit{Nothing} \quad\ \to \mathit{fail}\ k \\
= & \quad \{\ \text{induction hypothesis for } h\ \} \\
 & \mathbf{case}\ \mathit{eval}\ x\ s\ \mathbf{of} \\
 & \quad \mathit{Just}\ (n,s') \to \mathit{exec}\ c\ (\mathit{VAL}\ n : k, s') \\
 & \quad \mathit{Nothing} \quad \to \mathit{exec}\ (\mathit{comp}'\ h\ c)\ (k,s) \\
= & \quad \{\ \text{define: } \mathit{fail}\ (\mathit{HAN}\ c'\ s : k) = \mathit{exec}\ c'\ (k,s)\ \} \\
 & \mathbf{case}\ \mathit{eval}\ x\ s\ \mathbf{of} \\
 & \quad \mathit{Just}\ (n,s') \to \mathit{exec}\ c\ (\mathit{VAL}\ n : k, s') \\
 & \quad \mathit{Nothing} \quad \to \mathit{fail}\ (\mathit{HAN}\ (\mathit{comp}'\ h\ c)\ s : k) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ \_\ \_ : k, s') = \mathit{exec}\ c\ (\mathit{VAL}\ n : k, s')\ \} \\
 & \mathbf{case}\ \mathit{eval}\ x\ s\ \mathbf{of} \\
 & \quad \mathit{Just}\ (n,s') \to \mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ (\mathit{comp}'\ h\ c)\ s : k, s') \\
 & \quad \mathit{Nothing} \quad \to \mathit{fail}\ (\mathit{HAN}\ (\mathit{comp}'\ h\ c)\ s : k) \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{UNMARK}\ c))\ (\mathit{HAN}\ (\mathit{comp}'\ h\ c)\ s : k, s) \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{MARK}\ c''\ c')\ (k,s) = \mathit{exec}\ c'\ (\mathit{HAN}\ c''\ s : k, s)\ \} \\
 & \mathit{exec}\ (\mathit{MARK}\ (\mathit{comp}'\ h\ c)\ (\mathit{comp}'\ x\ (\mathit{UNMARK}\ c)))\ (k,s)
\end{array}
$$

Note that the new constructor *HAN* that is added to the *Elem* type within this calculation now has two arguments: one for the handler code (as in previous calculations), and one for the state to be used if the handler is invoked (for local state). We conclude the calculation with the case for *Put*, which proceeds in the same manner as for global state:

$$
\begin{array}{ll}
 & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Put}\ x\ y)\ c)\ (k,s) \\
= & \quad \{\ \text{specification (12)}\ \} \\
 & \mathbf{case}\ \mathit{eval}\ x\ s\ \mathbf{of} \\
 & \quad \mathit{Just}\ (n,s') \to \mathbf{case}\ \mathit{eval}\ y\ n\ \mathbf{of} \\
 & \qquad\qquad\qquad \mathit{Just}\ (m,s'') \to \mathit{exec}\ c\ (\mathit{VAL}\ m : k, s'') \\
 & \qquad\qquad\qquad \mathit{Nothing} \quad\ \to \mathit{fail}\ k \\
 & \quad \mathit{Nothing} \quad \to \mathit{fail}\ k \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & \mathbf{case}\ \mathit{eval}\ x\ s\ \mathbf{of} \\
 & \quad \mathit{Just}\ (n,s') \to \mathit{exec}\ (\mathit{comp}'\ y\ c)\ (k,n) \\
 & \quad \mathit{Nothing} \quad \to \mathit{fail}\ k \\
= & \quad \{\ \text{define: } \mathit{exec}\ (\mathit{SAVE}\ c')\ (\mathit{VAL}\ n : k, s') = \mathit{exec}\ c'\ (k,n)\ \} \\
 & \mathbf{case}\ \mathit{eval}\ x\ s\ \mathbf{of} \\
 & \quad \mathit{Just}\ (n,s') \to \mathit{exec}\ (\mathit{SAVE}\ (\mathit{comp}'\ y\ c))\ (\mathit{VAL}\ n : k, s') \\
 & \quad \mathit{Nothing} \quad \to \mathit{fail}\ k \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{SAVE}\ (\mathit{comp}'\ y\ c)))\ (k,s)
\end{array}
$$

In summary, collecting together everything that we have learned in the process of the above calculations, we obtained the following definitions.

Target language:

$$\begin{array}{l}\textbf{data}\ \mathit{Code} = \mathit{HALT} \mid \mathit{PUSH\ Int\ Code} \mid \mathit{ADD\ Code} \mid \\ \qquad\qquad \mathit{FAIL} \mid \mathit{MARK\ Code\ Code} \mid \mathit{UNMARK\ Code} \mid \\ \qquad\qquad \mathit{LOAD\ Code} \mid \mathit{SAVE\ Code}\end{array}$$

Compiler:

$$\begin{array}{ll}\mathit{comp} & :: \mathit{Expr} \to \mathit{Code} \\ \mathit{comp}\ e & = \mathit{comp}'\ e\ \mathit{HALT} \\ \mathit{comp}' & :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code} \\ \mathit{comp}'\ (\mathit{Val}\ n)\ c & = \mathit{PUSH}\ n\ c \\ \mathit{comp}'\ (\mathit{Add}\ x\ y)\ c & = \mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{ADD}\ c)) \\ \mathit{comp}'\ \mathit{Throw}\ c & = \mathit{FAIL} \\ \mathit{comp}'\ (\mathit{Catch}\ x\ h)\ c & = \mathit{MARK}\ (\mathit{comp}'\ h\ c)\ (\mathit{comp}'\ s\ (\mathit{UNMARK}\ c)) \\ \mathit{comp}'\ \mathit{Get}\ c & = \mathit{LOAD}\ c \\ \mathit{comp}'\ (\mathit{Put}\ x\ y)\ c & = \mathit{comp}'\ x\ (\mathit{SAVE}\ (\mathit{comp}'\ y\ c))\end{array}$$

Virtual machine:

$$\begin{array}{ll}\textbf{data}\ \mathit{Elem} & = \mathit{VAL\ Int} \mid \mathit{HAN\ Code\ State} \\ \mathit{exec} & :: \mathit{Code} \to \mathit{Conf} \to \mathit{Conf} \\ \mathit{exec}\ \mathit{HALT}\ (k,s) & = (k,s) \\ \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (k,s) & = \mathit{exec}\ c\ (\mathit{VAL}\ n : k,s) \\ \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : k,s) & = \mathit{exec}\ c\ (\mathit{VAL}\ (n+m) : k,s) \\ \mathit{exec}\ \mathit{FAIL}\ (k,s) & = \mathit{fail}\ k \\ \mathit{exec}\ (\mathit{MARK}\ h\ c)\ (k,s) & = \mathit{exec}\ c\ (\mathit{HAN}\ h\ s : k,s) \\ \mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ \_\ \_ : k,s) & = \mathit{exec}\ c\ (\mathit{VAL}\ n : k,s) \\ \mathit{exec}\ (\mathit{LOAD}\ c)\ (k,s) & = \mathit{exec}\ c\ (\mathit{VAL}\ s : k,s) \\ \mathit{exec}\ (\mathit{SAVE}\ c)\ (\mathit{VAL}\ n : k,s) & = \mathit{exec}\ c\ (k,n) \\ \mathit{fail} & :: \mathit{Stack} \to \mathit{Conf} \\ \mathit{fail}\ [\,] & = ([\,],0) \\ \mathit{fail}\ (\mathit{VAL}\ n : k) & = \mathit{fail}\ k \\ \mathit{fail}\ (\mathit{HAN}\ h\ s : k) & = \mathit{exec}\ h\ (k,s)\end{array}$$

Note that, as previously, we added an equation to *fail* for the case when the stack is empty in order to make the definition complete. Because *fail* does not take a state as an argument, we can only give a fixed output state as the result, for which purposes we simply return the value 0. As before, the choice for this additional equation has no impact on the correctness of the above calculations because they do not depend on this choice.

4.3 Reflection

Configurations The introduction of state only required a single amendment to our approach: instead of operating on a stack, the virtual machine *exec* now operates on a config-

urations comprising a stack and a state. This generalisation from stacks to configurations arose from the type of the evaluation function *eval* for global state, which takes an input state and produces an output state. However, this is an instance of a more general principle, in which all additional data structures on which *eval* depends are packaged up in the type of configurations alongside the stack. This also includes the state in the case of the local state semantics, even though an output state is not always returned. Similarly, in other cases where *eval* takes a data structure as an argument without returning an updated version, we included it in the configuration type. For example, in a language with variable binding, as we shall consider in section 5, *eval* takes an environment as input but does not return an updated version, but we include the environment in the configuration type.

Global vs local It is interesting to note that the compilers that result from the two semantics for state are precisely the same, with the difference being reflected in the virtual machines. In particular, in the case of local state the machine operation that marks that stack with handler code also stores the current state, which is subsequently restored if the handler is invoked, while for global state the current state is used when a handler is invoked. As in all our calculations, these behaviours arose naturally from the desire to apply induction hypotheses during the calculation process, and didn't require any prior knowledge of how the two forms of state can or should be implemented.

5 Lambda Calculus

For our final example, we consider a call-by-value variant of the lambda calculus. To simplify the presentation, we base our language on simple arithmetic expressions, but the same techniques apply if the language is extended with other features such as exceptions and state, and if the evaluation strategy is changed to call-by-name or call-by-need. We will also see a further refinement of the calculation process, in particular by starting with a semantics that is defined in relational rather than functional form.

5.1 Syntax

We extend our language of arithmetic expressions with the three basic primitives of the lambda calculus: variables, abstraction, and application. To avoid having to consider issues of variable capture and renaming, we represent variables using de Bruijn indices:

$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Var}\ \mathit{Int} \mid \mathit{Abs}\ \mathit{Expr} \mid \mathit{App}\ \mathit{Expr}\ \mathit{Expr}$$

Informally, *Var i* is the variable with de Bruijn index $i \geqslant 0$, *Abs x* constructs an abstraction over the expression x, and *App x y* applies the abstraction that results from evaluating the expression x to the value of the expression y. For example, the function $\lambda n \to (\lambda m \to n+m)$ that adds two integer values is represented as follows:

$$\begin{aligned} &\mathit{add} :: \mathit{Expr} \\ &\mathit{add} = \mathit{Abs}\ (\mathit{Abs}\ (\mathit{Add}\ (\mathit{Var}\ 1)\ (\mathit{Var}\ 0))) \end{aligned}$$

5.2 Semantics

Because the language now has first-class functions, it no longer suffices to use integers as the value domain for the semantics, and we also need to consider functional values:

$$\textbf{data}\ \mathit{Value} = \mathit{Num}\ \mathit{Int} \mid \mathit{Fun}\ (\mathit{Value} \to \mathit{Value})$$

Moreover, the semantics also requires an *environment* in order to interpret free variables. Using de Bruijn indices we can represent an environment simply as a list of values, with the value of variable i given by indexing into the list at position i:

$$\textbf{type}\ \mathit{Env} = [\mathit{Value}]$$

Using these types, it is now straightforward to define a function that evaluates an expression to a value within the context of a given environment:

$$\begin{array}{ll}
\mathit{eval} & :: \mathit{Expr} \to \mathit{Env} \to \mathit{Value} \\
\mathit{eval}\ (\mathit{Val}\ n)\ e & = \mathit{Num}\ n \\
\mathit{eval}\ (\mathit{Add}\ x\ y)\ e & = \textbf{case}\ \mathit{eval}\ x\ e\ \textbf{of} \\
 & \qquad \mathit{Num}\ n \to \textbf{case}\ \mathit{eval}\ y\ e\ \textbf{of} \\
 & \qquad\qquad \mathit{Num}\ m \to \mathit{Num}\ (n+m) \\
\mathit{eval}\ (\mathit{Var}\ i)\ e & = e\ !!\ i \\
\mathit{eval}\ (\mathit{Abs}\ x)\ e & = \mathit{Fun}\ (\lambda v \to \mathit{eval}\ x\ (v : e)) \\
\mathit{eval}\ (\mathit{App}\ x\ y)\ e & = \textbf{case}\ \mathit{eval}\ x\ e\ \textbf{of} \\
 & \qquad \mathit{Fun}\ f \to f\ (\mathit{eval}\ y\ e)
\end{array}$$

For example, applying *eval* to the expression $\mathit{App}\ (\mathit{App}\ \mathit{add}\ (\mathit{Val}\ 1))\ (\mathit{Val}\ 2)$ and the empty environment $[\,]$ gives the result $\mathit{Num}\ 3$, as expected. Note that because expressions in our source language may be badly formed or fail to terminate, *eval* is now a partial function. We will return to this issue at the end of this section.

We could now attempt to calculate a compiler based upon the above semantics. However, we would get stuck in the *Abs* case, at least if we used a straightforward specification for the compiler, due to the fact that *eval* is now a higher-order function, by virtue of the fact that abstractions denote functions of type $\mathit{Value} \to \mathit{Value}$. However, this problem is easily addressed by defunctionalising functions of this type into a new type *Lam* for lambda abstractions. Within the definition for *eval* there is only one form of such functions that is actually used, namely in the case for *Abs* when we return $\lambda v \to \mathit{eval}\ x\ (v : e)$. We represent functions of this form by means of a single constructor *Clo* for the *Lam* type, which takes the expression x and environment e as arguments:

$$\textbf{data}\ \mathit{Lam} = \mathit{Clo}\ \mathit{Expr}\ \mathit{Env}$$

The name of the constructor corresponds to the fact that an expression combined with an environment that captures its free variables is known as a *closure*. The fact that values of type *Lam* represent functions of type $\mathit{Value} \to \mathit{Value}$ can be formalised by defining a function that maps from one to the other:

$$\begin{array}{ll}
\mathit{apply} & :: \mathit{Lam} \to (\mathit{Value} \to \mathit{Value}) \\
\mathit{apply}\ (\mathit{Clo}\ x\ e) & = \lambda v \to \mathit{eval}\ x\ (v : e)
\end{array}$$

The name of this function derives from the fact that when its type is written in curried form as $Lam \to Value \to Value$, it can be viewed as applying the representation of a lambda expression to an argument value to give a result value. Using these ideas, we can now apply defunctionalisation to rewrite the semantics for our language in first-order form by replacing functions of type $Value \to Value$ by values of type Lam:

$$
\begin{array}{lll}
\textbf{data}\ Value & = & Num\ Int \mid Fun\ Lam \\
eval & :: & Expr \to Env \to Value \\
eval\ (Val\ n)\ e & = & Num\ n \\
eval\ (Add\ x\ y)\ e & = & \textbf{case}\ eval\ x\ e\ \textbf{of} \\
 & & \quad Num\ n \to \textbf{case}\ eval\ y\ e\ \textbf{of} \\
 & & \quad\quad Num\ m \to Num\ (n+m) \\
eval\ (Var\ i)\ e & = & e\ !!\ i \\
eval\ (Abs\ x)\ e & = & Fun\ (Clo\ x\ e) \\
eval\ (App\ x\ y)\ e & = & \textbf{case}\ eval\ x\ e\ \textbf{of} \\
 & & \quad Fun\ c \to apply\ c\ (eval\ y\ e)
\end{array}
$$

Because the definitions for *Lam* and *apply* are both just single equations, we inline them to simplify the definitions, resulting in the following semantics:

$$
\begin{array}{lll}
\textbf{data}\ Value & = & Num\ Int \mid Clo\ Expr\ Env \\
eval & :: & Expr \to Env \to Value \\
eval\ (Val\ n)\ e & = & Num\ n \\
eval\ (Add\ x\ y)\ e & = & \textbf{case}\ eval\ x\ e\ \textbf{of} \\
 & & \quad Num\ n \to \textbf{case}\ eval\ y\ e\ \textbf{of} \\
 & & \quad\quad Num\ m \to Num\ (n+m) \\
eval\ (Var\ i)\ e & = & e\ !!\ i \\
eval\ (Abs\ x)\ e & = & Clo\ x\ e \\
eval\ (App\ x\ y)\ e & = & \textbf{case}\ eval\ x\ e\ \textbf{of} \\
 & & \quad Clo\ x'\ e' \to eval\ x'\ (eval\ y\ e : e')
\end{array}
$$

However, in rewriting *eval* in first-order form we have now introduced another problem: the semantics is no longer compositional, i.e. structurally recursive, because in the case for $App\ x\ y$ we make a recursive call $eval\ x'$ on the auxiliary expression x' that results from evaluating the argument expression x. Hence, when calculating a compiler based upon this semantics we can no longer use simple structural induction as in our previous examples, but must use the more general approach of *rule induction* (Harper, 2013).

In order to facilitate the direct use of rule induction, the final step in this section is to redefine our functional evaluation semantics *eval* in a relational manner as a *big-step operational* (or natural) semantics, writing $x \Downarrow_e v$ to mean that the expression x can evaluate to the value v within the environment e. Formally, the evaluation relation $\Downarrow \subseteq Expr \times Env \times Value$ is defined by the following set of inference rules, which are obtained simply by rewriting the above definition for *eval* in relational style:

$$
\frac{}{Val\ n \Downarrow_e Num\ n} \qquad \frac{x \Downarrow_e Num\ n \qquad y \Downarrow_e Num\ m}{Add\ x\ y \Downarrow_e Num\ (n+m)} \qquad \frac{}{Var\ i \Downarrow_e e\ !!\ i}
$$

$$\frac{}{Abs\ x\ \Downarrow_e\ Clo\ x\ e} \qquad \frac{x\ \Downarrow_e\ Clo\ x'\ e' \qquad y\ \Downarrow_e\ v \qquad x'\ \Downarrow_{v:e'}\ w}{App\ x\ y\ \Downarrow_e\ w}$$

5.3 Specification

For the purposes of calculating a compiler based upon the above semantics, the types for the compilation function and virtual machine remain the same as for state:

$$comp' :: Expr \to Code \to Code$$
$$exec\ \ :: Code \to Conf \to Conf$$

However, because the semantics now requires the use of environment, this is included in the type for configurations, following the advice from section 4.3:

$$\textbf{type}\ Conf = (Stack, Env)$$

As with previous examples, a stack is initially defined as a list of values, with the element type being extended as and when required during the calculation process:

$$\textbf{type}\ Stack = [Elem]$$
$$\textbf{data}\ Elem = VAL\ Value$$

The specification for $comp'$ is similar to the original case for simple arithmetic expressions, except that our semantics is now defined as an evaluation relation $\Downarrow$, and the virtual machine now operates on configurations that comprise a stack and an environment:

$$exec\ (comp'\ x\ c)\ (s,e) = exec\ c\ (VAL\ v : s, e) \qquad \text{if } x\ \Downarrow_e\ v$$

It is straightforward to calculate a compiler from this specification. However, the result is not satisfactory. In particular, the fact that a value can be a closure that includes an unevaluated expression means that such expressions will be manipulated by the resulting virtual machine, whereas as we already noted with exceptions, it is natural to expect all expressions in the source language to be compiled away. The solution is the same as for exceptions: we simply replace the expression component of a closure by compiled code for the expression, by means of the following new type definitions:

$$\textbf{data}\ Value' = Num'\ Int \mid Clo'\ Code\ Env'$$
$$\textbf{type}\ Env' \quad = [Value']$$

In turn, these new types are then used to redefine the other basic types:

$$\textbf{type}\ Conf\ = (Stack, Env')$$
$$\textbf{type}\ Stack = [Elem]$$
$$\textbf{data}\ Elem = VAL\ Value'$$

Changing these definitions means that the above specification for $comp'$ is no longer type correct, because *eval* and *exec* now operate on different versions of the value type, namely *Value* and *Value*$'$, respectively. We therefore need a conversion function between

the two types. The case for *Num* is trivial, while we leave the case for *Clo* undefined at present, and aim to derive a definition for this case during the calculation process:

$$\begin{aligned} &\mathit{conv} && :: \mathit{Value} \to \mathit{Value}' \\ &\mathit{conv}\ (\mathit{Num}\ n) && = \mathit{Num}'\ n \\ &\mathit{conv}\ (\mathit{Clo}\ x\ e) && = \ ??? \end{aligned}$$

When applied to an environment e comprising a list of values, we write $\overline{e}$ as an abbreviation for *map conv e*. Using these ideas, it is now straightforward to modify the specification of comp' to take care of the necessary type conversions:

$$\mathit{exec}\ (\mathit{comp}'\ x\ c)\ (s,\overline{e}) = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{conv}\ v) : s,\overline{e}) \qquad \text{if } x \Downarrow_e v \tag{13}$$

5.4 Calculation

Based upon specification (13), we now calculate definitions for the compiler and the virtual machine by constructive rule induction on the assumption $x \Downarrow_e v$. In each case, we aim to rewrite the left-hand side $\mathit{exec}\ (\mathit{comp}'\ x\ c)\ (s,\overline{e})$ of the equation into the form $\mathit{exec}\ c'\ (s,\overline{e})$ for some code c', from which we can then conclude that the definition $\mathit{comp}'\ x\ c = c'$ satisfies the specification in this case. As with previous examples, along the way we will introduce new constructors into the code and stack types, and new equations for the function *exec*. Moreover, as part of the calculation we will also complete the definition for the conversion function *conv*. The cases for *Val* and *Var* are straightforward:

$$\begin{aligned} & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Val}\ n)\ c)\ (s,\overline{e}) \\ = \quad & \{\ \text{specification (13)}\ \} \\ & \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{conv}\ (\mathit{Num}\ n)) : s,\overline{e}) \\ = \quad & \{\ \text{definition of } \mathit{conv}\ \} \\ & \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}'\ n) : s,\overline{e}) \\ = \quad & \{\ \text{define: } \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (s,e) = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}'\ n) : s,e)\ \} \\ & \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (s,\overline{e}) \end{aligned}$$

and

$$\begin{aligned} & \mathit{exec}\ (\mathit{comp}'\ (\mathit{Var}\ i)\ c)\ (s,\overline{e}) \\ = \quad & \{\ \text{specification (13)}\ \} \\ & \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{conv}\ (e\ !!\ i)) : s,\overline{e}) \\ = \quad & \{\ \text{indexing lemma}\ \} \\ & \mathit{exec}\ c\ (\mathit{VAL}\ ((\mathit{map}\ \mathit{conv}\ e)\ !!\ i) : s,\overline{e}) \\ = \quad & \{\ \text{definition of } \overline{e}\ \} \\ & \mathit{exec}\ c\ (\mathit{VAL}\ (\overline{e}\ !!\ i) : s,\overline{e}) \\ = \quad & \{\ \text{define: } \mathit{exec}\ (\mathit{LOOKUP}\ i\ c)\ (s,e) = \mathit{exec}\ c\ (\mathit{VAL}\ (e\ !!\ i) : s,e)\ \} \\ & \mathit{exec}\ (\mathit{LOOKUP}\ i\ c)\ (s,\overline{e}) \end{aligned}$$

The indexing lemma used above is that $f\ (xs\ !!\ i) = (\mathit{map}\ f\ xs)\ !!\ i$, for any strict function f, list xs, and index i of the appropriate types. This lemma allows us to generalise over $\overline{e}$ when defining the behaviour of *exec* for the new code constructor *LOOKUP* that encapsulates the process of looking up a variable in the environment. Strictness of the function *conv* follows from the fact that it is defined by pattern matching on its argument value. Alternatively, we

could have avoided reasoning about strictness by using a list indexing operator that makes the possibility of failure explicit by returning a *Maybe* type.

In the case for *Add*, we can assume $x \Downarrow_e Num\ n$ and $y \Downarrow_e Num\ m$ by the inference rule that defines the behaviour of *Add x y*, together with induction hypotheses for the expressions x and y. The calculation then follows the same pattern as for simple arithmetic expressions, with the minor addition of applying the conversion function *conv*:

$$
\begin{array}{ll}
 & exec\ (comp'\ (Add\ x\ y)\ c)\ (s,\overline{e}) \\
= & \quad \{\ \text{specification (13)}\ \} \\
 & exec\ c\ (VAL\ (conv\ (Num\ (n+m)))) : s,\overline{e}) \\
= & \quad \{\ \text{definition of } conv\ \} \\
 & exec\ c\ (VAL\ (Num'\ (n+m)) : s,\overline{e}) \\
= & \quad \left\{ \begin{array}{l} \text{define: } exec\ (ADD\ c)\ (VAL\ (Num'\ m) : VAL\ (Num'\ n) : s,e) \\ \quad = exec\ c\ (VAL\ (Num'\ (n+m)) : s,e) \end{array} \right\} \\
 & exec\ (ADD\ c)\ (VAL\ (Num'\ m) : VAL\ (Num'\ n) : s,\overline{e}) \\
= & \quad \{\ \text{definition of } conv\ \} \\
 & exec\ (ADD\ c)\ (VAL\ (conv\ (Num\ m)) : VAL\ (conv\ (Num\ n)) : s,\overline{e}) \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & exec\ (comp'\ y\ (ADD\ c))\ (VAL\ (conv\ (Num\ n)) : s,\overline{e}) \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & exec\ (comp'\ x\ (comp'\ y\ (ADD\ c)))\ (s,\overline{e})
\end{array}
$$

In a similar manner, in the case for *App* we can assume that $x \Downarrow_e Clo\ x'\ e'$, $y \Downarrow_e v$, and $x' \Downarrow_{v:e'} w$ by the rule that defines the behaviour of *App x y*, together with the induction hypotheses for x, y and x'. The calculation then proceeds in the now familiar way, by introducing code and stack constructors as necessary in order to bring the configuration arguments into the right form for the induction hypotheses. First of all, in order to apply the induction hypothesis for x', we save and restore an environment on the stack by means of a new stack constructor *ENV* and code constructor *RET*:

$$
\begin{array}{ll}
 & exec\ (comp'\ (App\ x\ y)\ c)\ (s,\overline{e}) \\
= & \quad \{\ \text{specification (13)}\ \} \\
 & exec\ c\ (VAL\ (conv\ w) : s,\overline{e}) \\
= & \quad \{\ \text{define: } exec\ (RET\ c)\ (VAL\ v : ENV\ e : s,\_) = exec\ c\ (VAL\ v : s,e)\ \} \\
 & exec\ (RET\ c)\ (VAL\ (conv\ w) : ENV\ \overline{e} : s, conv\ v : \overline{e'}) \\
= & \quad \{\ \text{induction hypothesis for } x'\ \} \\
 & exec\ (comp'\ x'\ (RET\ c))\ (ENV\ \overline{e} : s, conv\ v : \overline{e'})
\end{array}
$$

In turn, to apply the induction hypothesis for y we introduce a new code constructor *APP* that encapsulates the idea of applying a closure to an argument value, with both the closure and the argument being supplied on the stack:

$$
\begin{array}{ll}
= & \quad \{\ \text{define: } exec\ APP\ (VAL\ v : VAL\ (Clo'\ c'\ e') : s,e) = exec\ c'\ (ENV\ e : s, v : e')\ \} \\
 & exec\ APP\ (VAL\ (conv\ v) : VAL\ (Clo'\ (comp'\ x'\ (RET\ c))\ \overline{e'}) : s,\overline{e}) \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & exec\ (comp'\ y\ APP)\ (VAL\ (Clo'\ (comp'\ x'\ (RET\ c))\ \overline{e'}) : s,\overline{e})
\end{array}
$$

To complete the calculation, we would now like to apply the induction hypothesis for x. For the above expression to have the required form, we need to solve the equation:

$$conv\ (Clo\ x'\ e') = Clo'\ (comp'\ x'\ (RET\ c))\ \overline{e'}$$

However, we can't simply use this equation as a definition for *conv* in the case of closures, because the code variable c is unbound in the body of the equation. At this point it now becomes evident that our earlier choice for defining the behaviour of the *RET* instruction was incorrect. In particular, this instruction should not take the code c as an argument, but rather take it from the stack. That is, we replace the earlier definition

$$exec\ (RET\ c)\ (VAL\ v : ENV\ e : s, \_) = exec\ c\ (VAL\ v : s, e)$$

by the following new version, in which the stack constructor *ENV* is replaced by a more general constructor *CLO* that takes both code and an environment as arguments:

$$exec\ RET\ (VAL\ v : CLO\ c\ e : s, \_) = exec\ c\ (VAL\ v : s, e)$$

Using this idea we restart the calculation for the *App* case, which now proceeds to completion in a straightforward manner, including the definition of *conv* for closures:

$$\begin{aligned}
&exec\ (comp'\ (App\ x\ y)\ c)\ (s, \overline{e}) \\
=\quad &\{\ \text{specification (13)}\ \} \\
&exec\ c\ (VAL\ (conv\ w) : s, \overline{e}) \\
=\quad &\{\ \text{define: } exec\ RET\ (VAL\ v : CLO\ c\ e : s, _) = exec\ c\ (VAL\ v : s, e)\ \} \\
&exec\ RET\ (VAL\ (conv\ w) : CLO\ c\ \overline{e} : s, conv\ v : \overline{e'}) \\
=\quad &\{\ \text{induction hypothesis for } x'\ \} \\
&exec\ (comp'\ x'\ RET)\ (CLO\ c\ \overline{e} : s, conv\ v : \overline{e'}) \\
=\quad &\{\ \text{define: } exec\ (APP\ c)\ (VAL\ v : VAL\ (Clo'\ c'\ e') : s, e) = exec\ c'\ (CLO\ c\ e : s, v : e')\ \} \\
&exec\ (APP\ c)\ (VAL\ (conv\ v) : VAL\ (Clo'\ (comp'\ x'\ RET)\ \overline{e'}) : s, \overline{e}) \\
=\quad &\{\ \text{induction hypothesis for } y\ \} \\
&exec\ (comp'\ y\ (APP\ c))\ (VAL\ (Clo'\ (comp'\ x'\ RET)\ \overline{e'}) : s, \overline{e}) \\
=\quad &\{\ \text{define: } conv\ (Clo\ x\ e) = Clo'\ (comp'\ x\ RET)\ \overline{e}\ \} \\
&exec\ (comp'\ y\ (APP\ c))\ (VAL\ (conv\ (Clo\ x'\ e')) : s, \overline{e}) \\
=\quad &\{\ \text{induction hypothesis for } x\ \} \\
&exec\ (comp'\ x\ (comp'\ y\ (APP\ c)))\ (s, \overline{e})
\end{aligned}$$

Finally, using the new equation for *conv*, the case for *Abs* simply introduces a code constructor *ABS* that encapsulates the process of putting a closure onto the stack:

$$\begin{aligned}
&exec\ (comp'\ (Abs\ x)\ c)\ (s, \overline{e}) \\
=\quad &\{\ \text{specification (13)}\ \} \\
&exec\ c\ (VAL\ (conv\ (Clo\ x\ e)) : s, \overline{e}) \\
=\quad &\{\ \text{definition for } conv\ \} \\
&exec\ c\ (VAL\ (Clo'\ (comp'\ x\ RET)\ \overline{e}) : s, \overline{e}) \\
=\quad &\{\ \text{define: } exec\ (ABS\ c'\ c)\ (s, e) = exec\ c\ (VAL\ (Clo'\ c'\ e) : s, e)\ \} \\
&exec\ (ABS\ (comp'\ x\ RET)\ c)\ (s, \overline{e})
\end{aligned}$$

In summary, we have calculated the definitions below. As with a number of earlier examples, the top-level compilation function *comp* is defined simply by applying *comp'*

to a trivial code constructor *HALT* that returns the current configuration.

Target language:

$$\begin{array}{l}\textbf{data}\ \mathit{Code} = \mathit{HALT} \mid \mathit{PUSH}\ \mathit{Int}\ \mathit{Code} \mid \mathit{ADD}\ \mathit{Code} \mid \mathit{LOOKUP}\ \mathit{Int}\ \mathit{Code} \mid \\ \qquad\qquad \mathit{ABS}\ \mathit{Code}\ \mathit{Code} \mid \mathit{RET} \mid \mathit{APP}\ \mathit{Code}\end{array}$$

Compiler:

$$\begin{array}{ll}\mathit{comp} & :: \mathit{Expr} \to \mathit{Code} \\ \mathit{comp}\ x & = \mathit{comp}'\ x\ \mathit{HALT} \\ \mathit{comp}' & :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code} \\ \mathit{comp}'\ (\mathit{Val}\ n)\ c & = \mathit{PUSH}\ n\ c \\ \mathit{comp}'\ (\mathit{Add}\ x\ y)\ c & = \mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{ADD}\ c)) \\ \mathit{comp}'\ (\mathit{Var}\ i)\ c & = \mathit{LOOKUP}\ i\ c \\ \mathit{comp}'\ (\mathit{Abs}\ x)\ c & = \mathit{ABS}\ (\mathit{comp}'\ x\ \mathit{RET})\ c \\ \mathit{comp}'\ (\mathit{App}\ x\ y)\ c & = \mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{APP}\ c))\end{array}$$

Virtual machine:

$$\begin{array}{ll}\textbf{data}\ \mathit{Elem} & = \mathit{VAL}\ \mathit{Value}' \mid \mathit{CLO}\ \mathit{Code}\ \mathit{Env}' \\ \mathit{exec} & :: \mathit{Code} \to \mathit{Conf} \to \mathit{Conf} \\ \mathit{exec}\ \mathit{HALT}\ (s,e) & = (s,e) \\ \mathit{exec}\ (\mathit{PUSH}\ n\ c)\ (s,e) & = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}'\ n) : s, e) \\ \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ (\mathit{Num}'\ m) : \mathit{VAL}\ (\mathit{Num}'\ n) : s, e) & = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Num}'\ (n+m)) : s, e) \\ \mathit{exec}\ (\mathit{LOOKUP}\ i\ c)\ (s,e) & = \mathit{exec}\ c\ (\mathit{VAL}\ (e\ !!\ i) : s, e) \\ \mathit{exec}\ (\mathit{ABS}\ c'\ c)\ (s,e) & = \mathit{exec}\ c\ (\mathit{VAL}\ (\mathit{Clo}'\ c'\ e) : s, e) \\ \mathit{exec}\ \mathit{RET}\ (\mathit{VAL}\ v : \mathit{CLO}\ c\ e : s, \_) & = \mathit{exec}\ c\ (\mathit{VAL}\ v : s, e) \\ \mathit{exec}\ (\mathit{APP}\ c)\ (\mathit{VAL}\ v : \mathit{VAL}\ (\mathit{Clo}'\ c'\ e') : s, e) & = \mathit{exec}\ c'\ (\mathit{CLO}\ c\ e : s, v : e')\end{array}$$

Conversion function:

$$\begin{array}{ll}\mathit{conv} & :: \mathit{Value} \to \mathit{Value}' \\ \mathit{conv}\ (\mathit{Num}\ n) & = \mathit{Num}'\ n \\ \mathit{conv}\ (\mathit{Clo}\ x\ e) & = \mathit{Clo}'\ (\mathit{comp}'\ x\ \mathit{RET})\ (\mathit{map}\ \mathit{conv}\ e)\end{array}$$

The above compiler is essentially the same at that presented in (Day & Hutton, 2013), except that it has now been calculated directly from a specification of its correctness.

5.5 Reflection

Defunctionalisation The key idea that facilitates a simple calculation in this section is the use of defunctionalisation to transform the semantics into first-order form. Without this initial step, formulating an appropriate specification for the lambda calculus compiler becomes significantly more complicated, as in (Meijer, 1992), due to the presence of a function type in the value domain. The same idea was also used in the work of Ager *et al.* (2003a) to simplify the derivation of abstract machines.

Relational semantics The use of a relational rather than functional semantics arose from the shift to rule induction rather than structural induction as the basis for the calculation. In addition, the relational semantics serves another purpose: it expresses the partiality of the semantics in a natural way. We can calculate the same compiler using the final functional semantics in section 5.2, but the calculation is complicated by the need to pay careful attention to the partiality of the evaluation function. In contrast, using a relational semantics allowed the calculation to proceed in the same straightforward manner as our previous examples, except that we used the more general technique of constructive rule induction on the evaluation relation, rather than constructive structural induction on the syntax for the source language. In this manner, starting from a relational semantics is a natural generalisation of our previous functional approach.

Soundness and completeness Specification (13) was sufficient for the purposes of calculating the compiler. However, due to the partiality of the underlying semantics, the specification only explicitly captures one half of compiler correctness for the lambda calculus, namely completeness. In particular, the specification states that compiled code can produce *every* result value that is permitted by our semantics. The dual property of soundness is just as important, to ensure that compiled code can *only* produce results that are permitted by the semantics. The example languages that we considered prior to this section all had a total (and deterministic) semantics, for which the resulting calculations also established soundness. Similarly, if we restrict the lambda calculus to a fragment for which the semantics is total, such as simply type lambda terms, we immediately obtain the soundness property from specification (13) as well. In general, however, if we have a relational semantics that is genuinely partial or non-deterministic, we need to explicitly consider both aspects of compiler correctness, as in (Hutton & Wright, 2007).

Partial specification In the definition for the conversion function *conv*, we initially left the case for closures undefined, as it was not yet clear how it should behave in this case. As such, equation (13) is a partial specification in terms of an incomplete definition for the function *conv*. In a similar manner to the *fail* function for exceptions, we derived the missing parts of the definition for *conv* during the calculation of the compiler. Once again, this approach is part of our desire to avoid predetermining implementation decisions, but rather letting these emerge naturally from the calculation process.

Design decisions During the calculation for the case of expressions of the form $App\ x\ y$, we made a design decision concerning the management of the stack that we subsequently had to revise because the calculation got stuck. This kind of behaviour is again characteristic of our approach, in which we try to make as few assumptions as possible, and let ourselves be guided by the desire to complete calculations by applying induction hypotheses. However, sometimes we then become stuck, and need to revisit our assumptions and decisions. In this way, we try to minimise the amount of foresight that is required.

Scalability The approach presented in this section also applies to call-by-name and call-by-need semantics. In the case of call-by-need, the semantics introduces a heap, which then becomes a component of the virtual machine's configuration type, similarly to a

state or environment. Our approach also scales to languages that combine lambda calculi with effects such as state and exceptions. However, when reformulating the functional semantics for lambda calculi with additional effects, some care is required. In particular, each equation in the original functional semantics should be translated to precisely one rule in the relational semantics. For a language with exceptions, the resulting semantics may not be the most natural formulation. But it is important that there is only one rule per language construct. Recall that in the calculation for exceptions, we needed to keep the *Just* and the *Nothing* cases aligned. If we were to decompose the semantics into different rules to deal with the different cases, we would lose this crucial interaction.

6 Related Work

As noted at the start of this article, the desire to calculate compilers from semantics has been a key objective in the field of program transformation for many years. In this section we review a range of related work, and explain how our approach compares.

Definitional Interpreters for Higher-Order Programming Languages (Reynolds, 1972) Many of the techniques used to derive compilers are due to the seminal work of Reynolds (1972). In particular, he introduced three key ideas. First of all, the notion of a 'definitional interpreter', to express the semantics for a language as an interpreter written in compositional style. Secondly, the idea of transforming such a semantics into continuation-passing style, to make control flow explicit in a manner that is independent of the evaluation order of the semantic meta-language. And finally, the concept of defunctionalisation, to transform higher-order programs into first-order form by representing functions as data structures. Using these techniques, Reynolds showed how to transform a definitional interpreter for a higher-order language into an equivalent abstract machine.

Deriving Target Code as a Representation of Continuation Semantics (Wand, 1982) The derivation of compilers was first considered by Wand (1982). Starting from a continuation semantics for the source language, Wand derives a compiler in a series of steps. Firstly, he reformulates the semantics in an equivalent point-free form using a generalised composition operator for functions with multiple arguments. During this process, he also introduces combinators that capture particular forms of argument manipulation. The resulting semantics is then defunctionalised to produce a compiler and a virtual machine. However, the machine code that results from this process is 'tree-shaped' rather than linear. In order to rectify this, Wand exploits the fact that the generalised composition operator can always be associated to the right to augment the compiler with on-the-fly 'rotation' operations that transform the resulting code into linear form.

In terms of comparing with our approach, the first difference is that Wand begins with a semantics that is already rather operational in style, in the form of a continuation semantics. This makes it more difficult to argue that the semantics is 'obviously correct'. Secondly, while rewriting the semantics using generalised composition leads to the introduction of a stack in the virtual machine, it requires the use of rotation to produce linear code. In contrast, our approach starts from a compiler specification that explicitly includes a stack, and does not require the use of rotation. Moreover, whereas Wand introduces continuation

combinators that are defunctionalised to code constructors, in our approach we introduce the code constructors directly during the calculation, without the need to go via a continuation semantics. Finally, another important difference is that Wand does not consider correctness proofs for his derivations, remarking that the article was devoted to a "heuristic development of the methodology rather than to proofs of its correctness". While he hints at possible techniques for proving the correctness of the resulting compilers, it is not the starting point for the derivation process as it is in our approach.

From Interpreter to Compiler and Virtual Machine: A Functional Derivation (Ager *et al.*, 2003b) Another approach to deriving compilers from semantics has been developed by Ager *et al.* (2003b). In this approach, one begins with a definitional interpreter, from which an abstract machine is derived by first rewriting the semantics in continuation-passing style and then defunctionalising. One then 'factorises' the resulting abstract machine into a compiler and virtual machine, by introducing a term model that implements a non-standard interpretation of the operations of the machine. However, this process involves transformation steps such as "make the definition compositional" and "factorize into a composition of combinators and recursive calls", without giving general principles for how these are achieved. Moreover, there is no argument about the correctness of the resulting compiler, apart from the statement that all the transformations are semantics preserving. But the goals of the authors are different to ours: they want to provide more insight into existing abstract/virtual machines and interpreters for lambda calculi, study relationships between them, and synthesise new machines and interpreters.

The fundamental difference to our work is best understood by looking at the derivation of abstract machines in Ager *et al.* (2003a), on which their later work (Ager *et al.*, 2003b) is based. We formulated our original calculational approach in sections 2.1 to 2.4 as the combination of three transformations steps that first introduce a stack, then a continuation, and finally defunctionalise. If we omit the introduction of a stack, we obtain the method of Ager *et al.* (2003a) to derive abstract machines. From this observation we can also conclude that the approach presented by Ager *et al.* (2003a) can be simplified by combining the two transformation steps together in the manner of section 2.5.

Calculating Compilers (Meijer, 1992) In his PhD dissertation, Meijer (1992) develops a number of techniques to calculate compilers from semantics for a variety of languages including a call-by-name lambda calculus, an imperative language with *if* statements and *while* loops, and a simple non-deterministic language.

In his lambda calculus calculation, Meijer starts with a higher-order functional semantics, in which compositionality is made explicit by defining the semantics using a *fold* operator on the syntax for the language. He then specifies an equivalent stack-based semantics, for which an implementation is calculated using algebraic properties of folds such as fusion and universality. The resulting stack-based semantics is then defunctionalised to produce a compiler and virtual machine. While Meijer emphasises the idea of *calculating* compilers as we do, his approach of starting with a higher-order semantics defined as a fold significantly complicates the methodology. In particular, the specification for the stack-semantics has the form of an adjunction rather than a simple equation as in our approach, which results in a much more complicated calculation process.

Meijer's calculation for the imperative language is impressive. As in our original stepwise approach in section 2, he calculates a semantics in continuation-passing style, but instead of a stack machine, he targets a register machine. The main calculation proceeds using structural induction, but the presence of unbounded loops leads to an auxiliary use of fixpoint induction in which we are required to 'guess' the correct induction hypothesis. The use of explicit (register) names in order to target a register machine also makes the calculation much more cumbersome. But the result is a compiler and virtual machine that is more closely aligned with typical hardware architectures. Our approach can also be applied to a language with unbounded loops. In contrast to Meijer's work, however, we do not need to use fixpoint induction or guess an induction hypothesis.

In his calculation for the non-deterministic language, Meijer also uses continuation-passing style. Moreover, as in our second approach to exceptions in section 3.2, he uses two continuations to distinguish between success and failure. However, in order to deal with non-determinism, he begins with a semantics expressed as a set-valued function. The same idea can also be used to adapt our approach to non-deterministic languages.

Meijer is able to calculate fairly realistic compilers by also considering optimising transformations that improve the quality of the compiled code. However, in general his approach requires more upfront knowledge about the desired compiler, whereas we aimed to reduce such knowledge as much as possible by using partial specifications.

Deriving a Lazy Abstract Machine (Sestoft, 1997) In this work the author derives an abstract machine for a call-by-need lambda calculus from a big-step operational semantics. While Sestoft's article derives an abstract machine rather than a compiler, it is still valuable to compare with our approach. His work is also noteworthy as it does not rely on the use of continuations or defunctionalisation, in contrast to the other related work above. Instead, the author presents a derivation that is guided by his insight into the source language.

The derivation given by Sestoft specifically targets the call-by-need lambda calculus, rather than being more generally applicable. He analyses the characteristics of the semantics, such as how laziness is handled and substitutions are represented, and presents techniques to reflect these characteristics in an efficient manner in an abstract machine. The correctness of the resulting machine is established separately. In contrast, our approach tries to minimise the insight that is necessary to transform a semantics into a compiler. Moreover, in our approach the derivation *is* the correctness proof. However, in return for the added effort in Sestoft's derivation, the resulting abstract machine implementation is able to perform a number of optimisations that improve its performance.

7 Conclusion and Further Work

In this article we presented a new approach to the problem of calculating correct compilers. Our approach builds upon previous work in the field, and was developed and refined by considering a series of languages of increasing complexity. Figure 1 summarises the general methodology, which can then be adapted as necessary depending on the nature of the source language. For example, as we have seen, for a number of language features and their combination it suffices to use the initial functional semantics as the basis for

1. Define a semantics for the language:
   - Define an evaluation function in a compositional manner
   - Defunctionalise to produce a first-order semantics
   - Rewrite the semantics in relational style
2. Define equations that specify the correctness of the compiler:
   - The equations relate the compiler to the semantics via a virtual machine
   - The specification may contain additional undefined components
   - The virtual machine operates on configurations comprising a stack and any additional data structures on which the semantics depends
3. Calculate definitions that satisfy the specifications:
   - The calculations proceed by constructive rule induction
   - We calculate all unknown components in the specification
   - The driving force is the desire to apply induction hypotheses

Fig. 1: General methodology for calculating correct compilers

the compiler specification, and to calculate the compiler by structural induction on the language syntax. The key attributes of our approach are as follows:

- *Directness* – it is based upon the idea of calculating compilers directly from high-level specifications of their correctness, rather than indirectly by applying a series of transformations to a semantics for the source language;
- *Simplicity* – it only requires simple equational reasoning techniques in the form of constructive induction, and avoids the need for more sophisticated concepts such as continuations and defunctionalisation during the calculation phase;
- *Partiality* – it uses partial (incomplete) specifications and definitions when necessary to avoid predetermining implementation decisions, with the missing components also being derived as part of the calculation process;
- *Goal driven* – it avoids the need for 'Eureka steps' by using the desire to apply induction hypotheses as the clear goal for the calculation process, from which the compilation machinery then arises in a natural manner;
- *Flexible* – it considers alternative design choices, and revisits assumptions when calculations get stuck, to emphasise that calculating compilers is usually not a purely deterministic process but still requires flexibility and creativity;
- *Mechanisation* – it is readily amenable to mechanisation, and all the calculations in the article have been mechanically verified using the Coq system, with the proofs being available online as supplementary material.

There are many possible avenues for developing the approach further. Interesting topics for further work include: providing mechanical support for the calculation process in an equational reasoning system such as HERMIT (Sculthorpe *et al.*, 2013); adapting the approach to different forms of virtual machines, such as register-based machines or machines with specific instruction sets; considering how to exploit additional algebraic structure during the calculation process, such as folds and monads; extending the approach to source

languages that are typed; exploring additional compilation concepts such as optimisation and modularity; and applying the technique to larger languages.

Acknowledgements

Discussions with Ralf Hinze at the WG 2.1 meeting in Zeegse led to the idea of combining the transformations in our original stepwise approach and greatly simplified the methodology. We would also like to thank the Functional Programming Lab in Nottingham, and Arjan Boeijink for many useful comments and suggestions.

References

Ager, Mads Sig, Biernacki, Dariusz, Danvy, Olivier, & Midtgaard, Jan. (2003a). A Functional Correspondence Between Evaluators and Abstract Machines. *Pages 8–19 of: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*. PPDP '03. New York, NY, USA: ACM.

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

Backhouse, Roland. (2003). *Program Construction: Calculating Implementations from Specifications*. John Wiley and Sons, Inc.

Day, Laurence E., & Hutton, Graham. (2013). Compilation à la Carte. To appear in the volume of selected papers from the 25th International Symposium on Implementation and Application of Functional Languages, Nijmegen, The Netherlands, August 2013.

Harper, Robert. (2013). *Practical Foundations for Programming Languages*. Cambridge University Press.

Hutton, Graham. (2007). *Programming in Haskell*. Cambridge University Press.

Hutton, Graham, & Wright, Joel. (2004). Compiling Exceptions Correctly. *Pages 211–227 of:* Kozen, Dexter (ed), *Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 3125. Springer Berlin / Heidelberg.

Hutton, Graham, & Wright, Joel. (2007). What is the Meaning of These Constant Interruptions? *Journal of Functional Programming*, **17**(06), 777–792.

Meijer, Erik. (1992). *Calculating Compilers*. Ph.D. thesis, Katholieke Universiteit Nijmegen.

Reynolds, John C. (1972). Definitional Interpreters for Higher-Order Programming Languages. *Pages 717–740 of: ACM '72: Proceedings of the ACM Annual Conference*. New York, NY, USA: ACM.

Schmidt, David A. (1986). *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc.

Sculthorpe, Neil, Farmer, Andrew, & Gill, Andy. (2013). The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language. *Pages 86–103 of: Implementation and Application of Functional Languages 2012*. Lecture Notes in Computer Science, vol. 8241. Springer.

Sestoft, Peter. (1997). Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, **7**(03), 231–264.

Thielecke, Hayo. (2002). Comparing Control Constructs by Double-Barrelled CPS. *Higher-Order and Symbolic Computation*, **15**(2/3).

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