JFP **25**, e14, 47 pages, 2015.
doi:10.1017/S0956796815000180

Calculating correct compilers

PATRICK BAHR

Department of Computer Science, University of Copenhagen, Denmark
(*e-mail:* `paba@diku.dk`)

GRAHAM HUTTON

School of Computer Science, University of Nottingham, UK
(*e-mail:* `graham.hutton@nottingham.ac.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 details of the implementation of the compilers 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, state, various forms of lambda calculi, bounded and unbounded loops, non-determinism and interrupts. All the calculations in the article have been formalised using the Coq proof assistant, which serves as a convenient interactive tool for developing and verifying the calculations.

1 Introduction

The ability 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 (1982a), 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 techniques, using standard equational reasoning. Our new approach builds upon previous work in the area, and focuses specifically on compilers that target stack-based virtual machines.

The starting point of our calculation process is the semantics for the source language in the form of an evaluation function. We then formulate an equational specification that captures the correctness of the compiler. Using this specification, we calculate definitions of the compiler and the virtual machine by *constructive induction* (Backhouse, 2003), using the desire to apply the induction hypotheses as the driving force for the calculation process. While our approach avoids direct use of continuations and defunctionalisation, these concepts are nonetheless useful for explaining the underlying ideas, and for comparing to other work in the literature. Therefore, we present our approach in two stages, firstly introducing the basic ideas in a series of transformation steps that include the use of continuations and defunctionalisation, and then showing how these steps can be combined into a single step that calculates directly from the compiler specification.

The techniques that we use are all well known. Our contribution is to show how they can be applied in a novel manner to give a new approach to calculate compilers that is both simple and generally applicable. It 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. A key ingredient for the scalability of our approach is the use of *partial specifications* to avoid predetermining implementation decisions. For example, the specification of a compiler for a language with exceptions may not stipulate how the compiler should behave when the result is an uncaught exception, as this requires up-front knowledge about how exceptions are to be implemented. Rather, the details of this behaviour are determined during the calculation process itself.

We develop our approach gradually. We introduce the basic methodology using a simple expression language, starting with a stepwise calculation, which we then combine into a single calculation (Section 2). Subsequently, we refine the methodology as we apply it to languages of increasing complexity: the use of partial specifications is demonstrated on a language with exceptions (Section 3); the use of configurations is demonstrated on a language with state (Section 4) and finally the use of rule induction for dealing with non-compositional semantics is demonstrated on a lambda calculus (Section 5).

All our programs and calculations 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 brevity, in this article, they are the central focus, so they are presented in detail. All the calculations have also been mechanically verified using the Coq proof assistant, and the proof scripts are available as online supplementary material at http://dx.doi.org/10.1017/S0956796815000180, together with all Haskell code and an appendix that covers an additional example.

2 Arithmetic expressions

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

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

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. These transformation steps involve continuations and defunctionalisation. However, we then simplify the process by combining the separate transformation steps, which results in a simple but powerful new approach to calculate 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:

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

Note that the definition for *eval* is *compositional*, 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 use simple inductive proof methods, we will typically aim to define our semantics in such a compositional manner. However, this may not always be possible, and in Section 5, we will see an example that uses a non-compositional semantics.

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, $\mathit{eval}_{\mathrm{S}}$, 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)

$$\textbf{type}\ \mathit{Stack} = [\mathit{Int}]$$

then we seek to derive a function

$$\mathit{eval}_{\mathrm{S}} :: \mathit{Expr} \rightarrow \mathit{Stack} \rightarrow \mathit{Stack}$$

such that:

$$\mathit{eval}_{\mathrm{S}}\ x\ s \quad = \quad \mathit{eval}\ x : s \tag{1}$$

The operator : is the list constructor in Haskell, which associates to the right. For example, $m : n : s$ is the list obtained by prepending two elements m and n to the list s.

Rather than first defining the function $\mathit{eval}_{\mathrm{S}}$ and then separately proving by induction that it satisfies the above equation, we aim to *calculate* a definition for $\mathit{eval}_{\mathrm{S}}$ that satisfies the equation by *constructive induction* (Backhouse, 2003) on the expression x, using the desire to apply the induction hypotheses as the driving force for the calculation process.

Specifically, we will start with the term $eval_{\mathsf{S}}\ x\ s$ and gradually transform it by equational reasoning. The goal is to arrive at a term t such that we can take $eval_{\mathsf{S}}\ x\ s = t$ as a defining equation for $eval_{\mathsf{S}}$. We do this by induction on the expression x, so we have to do a calculation for each case of x. In the base case, $Val\ n$, the calculation is easy:

$$
\begin{array}{ll}
 & eval_{\mathsf{S}}\ (Val\ n)\ s \\
= & \{\ \text{specification}\ \} \\
 & eval\ (Val\ n) : s \\
= & \{\ \text{definition of } eval\ \} \\
 & n : s \\
= & \{\ \text{define: } push_{\mathsf{S}}\ n\ s = n : s\ \} \\
 & push_{\mathsf{S}}\ n\ s
\end{array}
$$

Note that in the final step we defined an auxiliary function, $push_{\mathsf{S}}$, that captures the idea of pushing a number onto the stack. With the above calculation, we have discovered the definition of $eval_{\mathsf{S}}$ for expressions of the form $Val\ n$, namely

$$eval_{\mathsf{S}}\ (Val\ n)\ s = push_{\mathsf{S}}\ n\ s$$

In the inductive case, $Add\ x\ y$, we proceed as follows:

$$
\begin{array}{ll}
 & eval_{\mathsf{S}}\ (Add\ x\ y)\ s \\
= & \{\ \text{specification}\ \} \\
 & eval\ (Add\ x\ y) : s \\
= & \{\ \text{definition of } eval\ \} \\
 & (eval\ x + eval\ y) : s
\end{array}
$$

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:

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

Note that pushing $eval\ x$ onto the stack before $eval\ y$ in this calculation corresponds to the addition operator evaluating its arguments from left-to-right. It would be

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

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:

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

where

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

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:

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

We conclude by noting that introducing $push_{\mathsf{S}}$ and add_{S} may seem rather unnecessary at this point, and indeed, the above calculation can be performed without them. But we will see that subsequent steps are based on being able to encapsulate such operations as functions. However, the issue of when we need to introduce new definitions will become clear when the separate steps are combined together in Section 2.5.

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

$$\textbf{type}\ Cont = Stack \rightarrow Stack$$

then we seek to derive a function

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

such that

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

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

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

$$
\begin{aligned}
& \mathit{eval}_{\mathsf{C}}\ (\mathit{Val}\ n)\ c\ s \\
= & \quad \{\ \text{specification (2)}\ \} \\
& c\ (\mathit{eval}_{\mathsf{S}}\ (\mathit{Val}\ n)\ s) \\
= & \quad \{\ \text{definition of } \mathit{eval}_{\mathsf{S}}\ \} \\
& c\ (\mathit{push}_{\mathsf{S}}\ n\ s)
\end{aligned}
$$

while for the inductive case we calculate as follows:

$$
\begin{aligned}
& \mathit{eval}_{\mathsf{C}}\ (\mathit{Add}\ x\ y)\ c\ s \\
= & \quad \{\ \text{specification (2)}\ \} \\
& c\ (\mathit{eval}_{\mathsf{S}}\ (\mathit{Add}\ x\ y)\ s) \\
= & \quad \{\ \text{definition of } \mathit{eval}_{\mathsf{S}}\ \} \\
& c\ (\mathit{add}_{\mathsf{S}}\ (\mathit{eval}_{\mathsf{S}}\ y\ (\mathit{eval}_{\mathsf{S}}\ x\ s))) \\
= & \quad \{\ \text{function composition}\ \} \\
& (c \circ \mathit{add}_{\mathsf{S}})\ (\mathit{eval}_{\mathsf{S}}\ y\ (\mathit{eval}_{\mathsf{S}}\ x\ s)) \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
& \mathit{eval}_{\mathsf{C}}\ y\ (c \circ \mathit{add}_{\mathsf{S}})\ (\mathit{eval}_{\mathsf{S}}\ x\ s) \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
& \mathit{eval}_{\mathsf{C}}\ x\ (\mathit{eval}_{\mathsf{C}}\ y\ (c \circ \mathit{add}_{\mathsf{S}}))\ s
\end{aligned}
$$

In conclusion, we have calculated the following definition:

$$
\begin{aligned}
& \mathit{eval}_{\mathsf{C}} && :: \mathit{Expr} \to \mathit{Cont} \to \mathit{Cont} \\
& \mathit{eval}_{\mathsf{C}}\ (\mathit{Val}\ n)\ c\ s && = c\ (\mathit{push}_{\mathsf{S}}\ n\ s) \\
& \mathit{eval}_{\mathsf{C}}\ (\mathit{Add}\ x\ y)\ c\ s && = \mathit{eval}_{\mathsf{C}}\ x\ (\mathit{eval}_{\mathsf{C}}\ y\ (c \circ \mathit{add}_{\mathsf{S}}))\ s
\end{aligned}
$$

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

$$
\begin{aligned}
& \mathit{eval}_{\mathsf{S}} && :: \mathit{Expr} \to \mathit{Cont} \\
& \mathit{eval}_{\mathsf{S}}\ x && = \mathit{eval}_{\mathsf{C}}\ x\ (\lambda s \to s)
\end{aligned}
$$

The notation $\lambda x \to e$ is Haskell syntax for a lambda abstraction, in which x is the name of the bound variable and the expression e is the body.

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 functions of type $\mathit{Cont} = \mathit{Stack} \to \mathit{Stack}$ for continuations passed as arguments and returned as results, we define a datatype that represents the specific forms of continuations that we actually need for the purposes of our evaluation function.

Within the definitions for $\mathit{eval}_{\mathsf{S}}$ and $\mathit{eval}_{\mathsf{C}}$, there are only three forms of continuations that are 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:

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

$$
\begin{array}{ll}
halt_{\mathsf{C}} & :: Cont \\
halt_{\mathsf{C}} & = \lambda s \rightarrow s \\[4pt]
push_{\mathsf{C}} & :: Int \rightarrow Cont \rightarrow Cont \\
push_{\mathsf{C}}\ n\ c & = c \circ push_{\mathsf{S}}\ n \\[4pt]
add_{\mathsf{C}} & :: Cont \rightarrow Cont \\
add_{\mathsf{C}}\ c & = c \circ add_{\mathsf{S}}
\end{array}
$$

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

$$
\begin{array}{ll}
eval_{\mathsf{S}} & :: Expr \rightarrow Cont \\
eval_{\mathsf{S}}\ x & = eval_{\mathsf{C}}\ x\ halt_{\mathsf{C}} \\[4pt]
eval_{\mathsf{C}} & :: Expr \rightarrow Cont \rightarrow Cont \\
eval_{\mathsf{C}}\ (Val\ n)\ c & = push_{\mathsf{C}}\ n\ c \\
eval_{\mathsf{C}}\ (Add\ x\ y)\ c & = eval_{\mathsf{C}}\ x\ (eval_{\mathsf{C}}\ y\ (add_{\mathsf{C}}\ c))
\end{array}
$$

It is easy to check by unfolding definitions that these 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. We write the definition in generalised algebraic datatype style to highlight the correspondence:

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

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 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* is formalised by the function *exec*, which maps the former to the latter:

$$
\begin{array}{ll}
exec & :: Code \rightarrow Cont \\
exec\ HALT & = halt_{\mathsf{C}} \\
exec\ (PUSH\ n\ c) & = push_{\mathsf{C}}\ n\ (exec\ c) \\
exec\ (ADD\ c) & = add_{\mathsf{C}}\ (exec\ c)
\end{array}
$$

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:

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

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

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:

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

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 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.

Note that the code produced by our compiler is not a sequence of instructions, the form that one would typically associate with machine code. Rather, the code is in a form called *CPS notation* (Appel, 1991). This representation of code was first used in early compilers for Scheme (Steele, 1978; Adams *et al.*, 1986), and has proved to be beneficial for implementing optimising compilers (Appel, 1991). Despite sharing the same name, one should not confuse code represented in this style with the CPS semantics in Section 2.3. In the former, continuations are represented symbolically, whereas in the latter continuations are functions.

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:

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

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

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

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

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

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 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 appear 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:

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

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

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.

We begin with equation (4), and proceed by constructive induction on the expression x. In each case, we aim to rewrite the left-hand side $exec\ (comp'\ x\ c)\ s$ of the equation into the form $exec\ c'\ s$ for some code c', from which we can then

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

conclude that the definition $comp'\ x\ c = c'$ 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:

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

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 $exec\ c'\ s$ for some code c'. That is, in order to complete the calculation we need to solve the equation

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

Note that we can't simply use this equation as a definition for *exec*, because the variables n and c would be unbound in the body of the definition. 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,

$$PUSH :: Int \to Code \to Code$$

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

$$exec\ (PUSH\ n\ c)\ s = exec\ c\ (n : s)$$

That is, executing the code $PUSH\ n\ c$ 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:

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

The final expression now has the form $exec\ c'\ s$, where $c' = PUSH\ n\ c$, from which we conclude that the following definition satisfies specification (4) in the base case:

$$comp'\ (Val\ n)\ c = PUSH\ n\ c$$

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

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

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

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

hypotheses for the two argument expressions x and y, namely

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

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

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

$$ADD :: Code \rightarrow Code$$

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:

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

Note that as in Section 2.2, we chose to transform the stack into the form $eval\ y : eval\ x : s$. We could have equally well chosen the opposite order, $eval\ x : eval\ y : s$, which would have resulted in right-to-left evaluation for Add. We have this freedom in the calculation because the semantics defined by $eval$ does not specify an evaluation order.

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

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 $exec\ (comp\ x)\ s$ of the equation into the form $exec\ c\ s$ for some code c, from which we can then conclude that the definition $comp\ x = c$ 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:

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

In conclusion, we have calculated the following definitions:

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

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 summary, 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.

Our full methodology for calculating compilers is given at the end of the article in Figure 1 on page 44. For the purpose of exposition, however, we will introduce the details of the general approach step-by-step using example languages of increasing complexity, gradually refining our approach as we progress. These refinements to the methodology should not be confused with its application to calculate correct compilers.

2.6 Reflection

We conclude this section with some reflective remarks on our original and combined approaches to calculate a compiler for arithmetic expressions, together with some comments on the relationship between derivations and proofs.

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 readers 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. The use of equations of this form to express and then prove compiler correctness can be traced back to the pioneering work on compiler verification by McCarthy & Painter (1967).

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

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

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

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

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.

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

$$
\begin{aligned}
& \mathit{eval}_{\mathsf{C}}\ (\mathit{Val}\ n)\ c\ s \\
= \quad & \{\ \text{specification (5)}\ \} \\
& c\ (\mathit{eval}\ (\mathit{Val}\ n) : s) \\
= \quad & \{\ \text{definition of } \mathit{eval}\ \} \\
& c\ (n : s) \\
= \quad & \{\ \text{define: } \mathit{push}\ n\ c\ s = c\ (n : s)\ \} \\
& \mathit{push}\ n\ c\ s
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{comp}'\ (\mathit{Val}\ n)\ c\ \$\$\ s \\
= \quad & \{\ \text{specification (6)}\ \} \\
& c\ \$\$\ (\mathit{eval}\ (\mathit{Val}\ n) : s) \\
= \quad & \{\ \text{definition of } \mathit{eval}\ \} \\
& c\ \$\$\ (n : s) \\
= \quad & \{\ \text{define: } \mathit{PUSH}\ n\ c\ \$\$\ s = c\ \$\$\ (n : s)\ \} \\
& \mathit{PUSH}\ n\ c\ \$\$\ s
\end{aligned}
$$

Mechanisation. Eliminating the use of continuations is also important 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 (Dybjer, 1994). In contrast, there is no such problem when mechanising the calculations in our combined approach. All our compiler calculations have been mechanically verified in the Coq system, and the proof scripts are available online as supplementary material. The only difference between the calculations in the article and their formalisation in Coq is that in the latter case, we define the virtual machines as relations rather than as functions, because the termination checker for Coq only accepts functions whose definitions are structurally recursive.

Partiality. Because the *ADD* instruction fails if the stack does not contain at least two values, the function *exec* implements the virtual machine is partial. As remarked by Ager *et al.* (2003a), such partiality is 'inherent to programming abstract machines in an ML-like language'. If desired, *exec* could be turned into a total function by using a dependently- typed language to make the stack demands of each machine instruction explicit in its type (McKinna & Wright, 2006). However, we do not require such additional effort here as we are only interested in the behaviour of *exec* for well-formed code produced by our compiler, as expressed in specifications (3) and (4).

Exposition. Given the benefits of the combined approach, why didn't we simply present this 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. 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.

Derivation versus proof. The purpose of our calculations is to *derive* definitions that satisfy their specifications. In addition, the calculations can also be read as

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

proofs that the definitions satisfy their specifications. In particular, each of our calculations starts off by applying a specification; if we remove this first step from the calculation and add a new step at the end that applies the definition, the calculation can then be read as a proof. For example, our calculation of the definition $comp'\ (Val\ n)\ c = PUSH\ \ n\ c$ from specification (4),

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

can also be read as a proof that this definition satisfies the specification:

$$
\begin{aligned}
& exec\ c\ (eval\ (Val\ n) : s) \\
= \quad & \{ \text{ definition of } eval \ \} \\
& exec\ c\ (n : s) \\
= \quad & \{ \text{ define: } exec\ (PUSH\ \ n\ c)\ s = c\ (n : s) \ \} \\
& exec\ (PUSH\ \ n\ c)\ s \\
= \quad & \{ \text{ definition of } comp' \ \} \\
& exec\ (comp'\ (Val\ n)\ c)\ s
\end{aligned}
$$

We could have performed all calculations in the article in this form instead. Indeed, our calculations in Coq proceed in this way. However, from the point of view of *discovering* definitions, as opposed to verifying them, we prefer the derivation-based approach.

3 Exceptions

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

$$\mathbf{data}\ Expr = Val\ Int \mid Add\ \ Expr\ \ Expr \mid Throw \mid Catch\ \ Expr\ \ Expr$$

Informally, $Catch\ \ x\ \ h$ behaves as the expression x unless evaluation of x throws an exception, in which case the catch behaves as the *handler* expression h. An exception is thrown if evaluation of $Throw$ is attempted. To define the semantics for this extended language in the form of an evaluation function, we first recall the $Maybe$ type:

$$\mathbf{data}\ Maybe\ a = Just\ \ a \mid Nothing$$

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 (Spivey, 1990). Using this type, our original evaluator can be rewritten to take account of exceptions as follows:

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

$$
\begin{array}{lll}
\mathit{eval} & :: \mathit{Expr} \rightarrow \mathit{Maybe\ Int} & \\
\mathit{eval}\ (\mathit{Val}\ n) & = \mathit{Just}\ n & \\
\mathit{eval}\ (\mathit{Add}\ x\ y) & = \textbf{case}\ \mathit{eval}\ x\ \textbf{of} & \\
 & \quad \mathit{Just}\ n \quad \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of} & \\
 & \quad\quad\quad\quad \mathit{Just}\ m \quad \rightarrow \mathit{Just}\ (n+m) & \\
 & \quad\quad\quad\quad \mathit{Nothing} \rightarrow \mathit{Nothing} & \\
 & \quad \mathit{Nothing} \rightarrow \mathit{Nothing} & \\
\mathit{eval}\ \mathit{Throw} & = \mathit{Nothing} & \\
\mathit{eval}\ (\mathit{Catch}\ x\ h) & = \textbf{case}\ \mathit{eval}\ x\ \textbf{of} & \\
 & \quad \mathit{Just}\ n \quad \rightarrow \mathit{Just}\ n & \\
 & \quad \mathit{Nothing} \rightarrow \mathit{eval}\ h &
\end{array}
$$

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. Monads are an excellent tool for abstraction, in particular, for hiding the underlying 'plumbing' of computations. However, when calculating compilers such low-level details matter, in particular, how different language features interact, so we prefer to use non-monadic definitions. 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 equations for 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, its choice 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, including a *partial definition* for the type of stack elements. The missing components in the specification are then derived during the calculation process. We will also see an example of a calculation that gets stuck, which requires us to go back and change the specification accordingly.

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:

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

However, rather than taking $\mathit{Stack} = [\mathit{Int}]$ as before, we use an alternative representation of stacks, in which the elements are wrapped up in a new datatype *Elem*:

$$\textbf{type}\ \mathit{Stack} = [\mathit{Elem}]$$

$$\textbf{data}\ \mathit{Elem} = \mathit{VAL}\ \mathit{Int}$$

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

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

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

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:

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

Just as with the function $comp'$ itself, we aim to derive a definition for *fail* that satisfies this equation during the calculation process. In summary, we now have the following *partial specification* for the new compilation function $comp'$ in terms of an as yet undefined function $fail :: Expr \to Code \to Stack \to Stack$:

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

We could now start to calculate a definition for $comp'$ from this equation by constructive induction on x. However, the calculation would soon get stuck. In particular, note that each of the variables x, c and s has two occurrences in the case expression in specification (7). Consequently, in order to use the induction hypotheses during the calculation, we have to make sure that the instantiations of x, c and s are aligned. For example, during the calculation for addition, we would encounter the following term:

$$\begin{aligned} & \mathbf{case}\ eval\ y\ \mathbf{of} \\ & \quad Just\ m \quad \to exec\ (ADD\ c)\ (VAL\ m : VAL\ n : s) \\ & \quad Nothing \to fail\ (Add\ x\ y)\ c\ s \end{aligned}$$

To apply the induction hypothesis for y, this term would need to be rewritten to match the form of specification (7). To this end, the use of the code $ADD\ c$ and the stack $VAL\ m : VAL\ n : s$ in the *Just* case above means that the *Nothing* case needs to be rewritten into the form $fail\ y\ (ADD\ c)\ (VAL\ n : s)$. The natural way to achieve this would be to introduce $fail\ y\ (ADD\ c)\ (VAL\ n : s) = fail\ (Add\ x\ y)\ c\ s$ as a new defining equation for *fail*. However, this is not a valid definition because the expression x is unbound in the body. In conclusion, we get stuck trying to keep the expression argument to *fail* aligned. A similar issue occurs with the code argument when applying the induction hypothesis for x.

Fortunately, there is a simple solution to the problem of keeping the arguments to *fail* aligned that allows the calculation to proceed: we remove the *Expr* and *Code* arguments that caused problems, as these turn out to be unnecessary. This yields the following revised specification, where *fail* has now the type $Stack \rightarrow Stack$:

$$
\begin{array}{lll}
exec\ (comp'\ x\ c)\ s & = & \textbf{case}\ eval\ x\ \textbf{of} \\
 & & \quad Just\ n \quad \rightarrow exec\ c\ (VAL\ n : s) \\
 & & \quad Nothing \rightarrow fail\ s
\end{array} \tag{8}
$$

We now calculate a definition for $comp'$ from this equation by constructive induction on x, aiming to rewrite the left-hand side $exec\ (comp'\ x\ c)\ s$ into the form $exec\ c'\ s$ for some code c', from which we can then conclude that the definition $comp'\ x\ c = c'$ 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 easy:

$$
\begin{array}{ll}
 & exec\ (comp'\ (Val\ n)\ c)\ s \\
= & \quad \{\ \text{specification (8)}\ \} \\
 & exec\ c\ (VAL\ n : s) \\
= & \quad \{\ \text{define: } exec\ (PUSH\ \ n\ c)\ s = exec\ c\ (VAL\ n : s)\ \} \\
 & exec\ (PUSH\ \ n\ c)\ s
\end{array}
$$

and

$$
\begin{array}{ll}
 & exec\ (comp'\ Throw\ \ c)\ s \\
= & \quad \{\ \text{specification (8)}\ \} \\
 & fail\ s \\
= & \quad \{\ \text{define: } exec\ FAIL\ s = fail\ s\ \} \\
 & exec\ FAIL\ s
\end{array}
$$

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:

$$
\begin{array}{ll}
 & exec\ (comp'\ (Add\ x\ y)\ c)\ s \\
= & \quad \{\ \text{specification (8)}\ \} \\
 & \textbf{case}\ eval\ x\ \textbf{of} \\
 & \quad Just\ n \quad \rightarrow \textbf{case}\ eval\ y\ \textbf{of} \\
 & \qquad\qquad\qquad Just\ m \quad \rightarrow exec\ c\ (VAL\ (n+m) : s) \\
 & \qquad\qquad\qquad Nothing \rightarrow fail\ s \\
 & \quad Nothing \rightarrow fail\ s \\
= & \quad \{\ \text{define: } exec\ (ADD\ \ c)\ (VAL\ m : VAL\ n : s) = exec\ c\ (VAL\ (n+m) : s)\ \} \\
 & \textbf{case}\ eval\ x\ \textbf{of} \\
 & \quad Just\ n \quad \rightarrow \textbf{case}\ eval\ y\ \textbf{of} \\
 & \qquad\qquad\qquad Just\ m \quad \rightarrow exec\ (ADD\ \ c)\ (VAL\ m : VAL\ n : s) \\
 & \qquad\qquad\qquad Nothing \rightarrow fail\ s \\
 & \quad Nothing \rightarrow fail\ s
\end{array}
$$

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

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 (8), the use of the stack $VAL\ m : VAL\ n : s$ in the *Just* case means that the argument of *fail* in the *Nothing* case must be $VAL\ n : s$ rather than just s. This observation gives our first defining equation for *fail*, and we continue as follows:

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

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:

$$
\begin{array}{ll}
& exec\ (comp'\ (Catch\ x\ h)\ c)\ s \\
= & \quad \{\ \text{specification (8)}\ \} \\
& \textbf{case}\ eval\ x\ \textbf{of} \\
& \quad Just\ n \quad \rightarrow exec\ c\ (VAL\ n : s) \\
& \quad Nothing \rightarrow \textbf{case}\ eval\ h\ \textbf{of} \\
& \qquad\qquad\qquad Just\ m \quad \rightarrow exec\ c\ (VAL\ m : s) \\
& \qquad\qquad\qquad Nothing \rightarrow fail\ s \\
= & \quad \{\ \text{induction hypothesis for } h\ \} \\
& \textbf{case}\ eval\ x\ \textbf{of} \\
& \quad Just\ n \quad \rightarrow exec\ c\ (VAL\ n : s) \\
& \quad Nothing \rightarrow exec\ (comp'\ h\ c)\ s
\end{array}
$$

Now, we are in a similar position to the calculation for *Add*, i.e. the *Nothing* case does not match the form of specification (8). 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

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

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:

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

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 $comp'\ h\ c$ is provided on the stack by means of a HAN constructor with a single argument. In particular, if we define

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

then by taking $c' = comp'\ h\ c$, we obtain the equation

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

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:

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

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 $comp :: Expr \to Code$. For arithmetic expressions, the desired behaviour of $comp$ was specified by the equation $exec\ (comp\ x)\ s = eval\ x : s$. Based upon our experience with $comp'$, in the presence of exceptions we refine this

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

equation as follows:

$$exec\ (comp\ x)\ s \quad = \quad \mathbf{case}\ eval\ x\ \mathbf{of} \qquad (9)$$
$$\qquad Just\ n \quad \rightarrow VAL\ n : s$$
$$\qquad Nothing \rightarrow fail\ s$$

To calculate a definition for *comp* from this equation, we aim to rewrite the left-hand side $exec\ (comp\ x)\ s$ into the form $exec\ c'\ s$ for some code c', and hence define $comp\ x = c'$. The calculation proceeds in the same manner as in Section 2.5, 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*′:

$$
\begin{array}{ll}
 & exec\ (comp\ x)\ s \\
= & \quad \{\ \text{specification (9)}\ \} \\
 & \mathbf{case}\ eval\ x\ \mathbf{of} \\
 & \quad Just\ n \quad \rightarrow VAL\ n : s \\
 & \quad Nothing \rightarrow fail\ s \\
= & \quad \{\ \text{define: } exec\ HALT\ s = s\ \} \\
 & \mathbf{case}\ eval\ x\ \mathbf{of} \\
 & \quad Just\ n \quad \rightarrow exec\ HALT\ (VAL\ n : s) \\
 & \quad Nothing \rightarrow fail\ s \\
= & \quad \{\ \text{specification (8)}\ \} \\
 & exec\ (comp'\ x\ HALT)\ s
\end{array}
$$

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

Target language:

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

Compiler:

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

Virtual machine:

$$
\begin{array}{ll}
\mathbf{type}\ Stack & = [Elem] \\
\mathbf{data}\ Elem & = VAL\ Int \mid HAN\ Code \\
exec & :: Code \rightarrow Stack \rightarrow Stack \\
exec\ HALT\ s & = s \\
exec\ (PUSH\ n\ c)\ s & = exec\ c\ (VAL\ n : s) \\
exec\ (ADD\ c)\ (VAL\ m : VAL\ n : s) & = exec\ c\ (VAL\ (n + m) : s)
\end{array}
$$

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

$$
\begin{array}{ll}
\mathit{exec}\ \mathit{FAIL}\ s & = \mathit{fail}\ s \\
\mathit{exec}\ (\mathit{MARK}\ c'\ c)\ s & = \mathit{exec}\ c\ (\mathit{HAN}\ c' : s) \\
\mathit{exec}\ (\mathit{UNMARK}\ c)\ (\mathit{VAL}\ n : \mathit{HAN}\ \_ : s) & = \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
\mathit{fail} & :: \mathit{Stack} \rightarrow \mathit{Stack} \\
\mathit{fail}\ [\,] & = [\,] \\
\mathit{fail}\ (\mathit{VAL}\ n : s) & = \mathit{fail}\ s \\
\mathit{fail}\ (\mathit{HAN}\ c : s) & = \mathit{exec}\ c\ s
\end{array}
$$

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 *fail* $[\,] = [\,]$ in this case. In principle, any choice would be fine, because the calculation does not depend on it. Our choice is motivated by the following observation: if we instantiate $s = [\,]$ in specification (8), 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 (Chase, 1994a; Chase, 1994b), 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 by Hutton & Wright (2004), except that our compiler here uses code continuations, and has been derived directly from a specification of its correctness, with all the compilation machinery falling naturally out of the calculation process. There was little room for alternative choices in the process: we could have compiled addition differently using the fact that it is commutative, and we could have compiled exception handlers dynamically as described above. Otherwise, the calculation process was fully determined by the desire to apply the induction hypotheses and to arrive at a term of the form $\mathit{exec}\ c'\ s$. This observation underlines the systematic nature of our approach, which only leaves a few design choices.

Finally, we note that the code produced by the above compiler is not fully linear, because the *MARK* constructor takes two arguments of type *Code*. This branching structure corresponds to the underlying branching in control flow in the semantics of the *Catch* operation of the language. However, as demonstrated by Hutton & Wright (2004), if desired we can systematically transform the compiler to produce linear code, by modifying *MARK* to take a *code pointer* as its first argument rather than code itself. Moreover, this transformation requires little additional effort to establish its correctness (Bahr, 2014).

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:

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

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 of *comp′*, in which the arguments *sc* and *fc* are the success and failure code continuations, and *s* is the stack:

$$\begin{array}{lcl} exec\ (comp'\ x\ sc\ fc)\ s & = & \mathbf{case}\ eval\ x\ \mathbf{of} \\ & & \quad Just\ n \quad\ \to exec\ sc\ (VAL\ n : s) \\ & & \quad Nothing \to exec\ fc\ s \end{array} \tag{10}$$

From this specification, we calculate the definition for *comp′* by constructive induction on the expression x. The cases for *Val* and *Throw* are again easy:

$$\begin{array}{cl} & exec\ (comp'\ (Val\ n)\ sc\ fc)\ s \\ = & \{\ \text{specification (10)}\ \} \\ & exec\ sc\ (VAL\ n : s) \\ = & \{\ \text{define: } exec\ (PUSH\ \ n\ c)\ s = exec\ c\ (VAL\ n : s)\ \} \\ & exec\ (PUSH\ \ n\ sc)\ s \end{array}$$

and

$$\begin{array}{cl} & exec\ (comp'\ Throw\ \ sc\ fc)\ s \\ = & \{\ \text{specification (10)}\ \} \\ & exec\ fc\ s \end{array}$$

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:

$$\begin{array}{cl} & exec\ (comp'\ (Catch\ \ x\ \ h)\ sc\ fc)\ s \\ = & \{\ \text{specification (10)}\ \} \\ & \mathbf{case}\ eval\ \ x\ \ \mathbf{of} \end{array}$$

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

$$
\begin{array}{ll}
& \quad \mathit{Just}\ n \quad \rightarrow \mathit{exec}\ \mathit{sc}\ (\mathit{VAL}\ n : s) \\
& \quad \mathit{Nothing} \rightarrow \mathbf{case}\ h\ \mathbf{of} \\
& \qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{exec}\ \mathit{sc}\ (\mathit{VAL}\ m : s) \\
& \qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{exec}\ \mathit{fc}\ s \\
= & \quad \{ \text{ induction hypothesis for } h \ \} \\
& \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
& \quad \mathit{Just}\ n \quad \rightarrow \mathit{exec}\ \mathit{sc}\ (\mathit{VAL}\ n : s) \\
& \quad \mathit{Nothing} \rightarrow \mathit{exec}\ (\mathit{comp}'\ h\ \mathit{sc}\ \mathit{fc})\ s \\
= & \quad \{ \text{ induction hypothesis for } x \ \} \\
& \mathit{exec}\ (\mathit{comp}'\ x\ \mathit{sc}\ (\mathit{comp}'\ h\ \mathit{sc}\ \mathit{fc}))\ s
\end{array}
$$

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 purpose we introduce a *POP* constructor:

$$
\begin{array}{ll}
& \mathit{exec}\ (\mathit{comp}'\ (\mathit{Add}\ x\ y)\ \mathit{sc}\ \mathit{fc})\ s \\
= & \quad \{ \text{ specification (10) } \} \\
& \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
& \quad \mathit{Just}\ n \quad \rightarrow \mathbf{case}\ \mathit{eval}\ y\ \mathbf{of} \\
& \qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{exec}\ \mathit{sc}\ (\mathit{VAL}\ (n+m) : s) \\
& \qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{exec}\ \mathit{fc}\ s \\
& \quad \mathit{Nothing} \rightarrow \mathit{exec}\ \mathit{fc}\ s \\
= & \quad \{ \text{ define: } \mathit{exec}\ (\mathit{ADD}\ c)\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) = \mathit{exec}\ c\ (\mathit{VAL}\ (n+m) : s) \ \} \\
& \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
& \quad \mathit{Just}\ n \quad \rightarrow \mathbf{case}\ \mathit{eval}\ y\ \mathbf{of} \\
& \qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{exec}\ (\mathit{ADD}\ \mathit{sc})\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
& \qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{exec}\ \mathit{fc}\ s \\
& \quad \mathit{Nothing} \rightarrow \mathit{exec}\ \mathit{fc}\ s \\
= & \quad \{ \text{ define: } \mathit{exec}\ (\mathit{POP}\ c)\ (\mathit{VAL}\ \_ : s) = \mathit{exec}\ c\ s \ \} \\
& \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
& \quad \mathit{Just}\ n \quad \rightarrow \mathbf{case}\ \mathit{eval}\ y\ \mathbf{of} \\
& \qquad\qquad\qquad \mathit{Just}\ m \quad \rightarrow \mathit{exec}\ (\mathit{ADD}\ \mathit{sc})\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
& \qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{exec}\ (\mathit{POP}\ \mathit{fc})\ (\mathit{VAL}\ n : s) \\
& \quad \mathit{Nothing} \rightarrow \mathit{exec}\ \mathit{fc}\ s \\
= & \quad \{ \text{ induction hypothesis for } y \ \} \\
& \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
& \quad \mathit{Just}\ n \quad \rightarrow \mathit{exec}\ (\mathit{comp}'\ y\ (\mathit{ADD}\ \mathit{sc})\ (\mathit{POP}\ \mathit{fc}))\ (\mathit{VAL}\ n : s) \\
& \quad \mathit{Nothing} \rightarrow \mathit{exec}\ \mathit{fc}\ s \\
= & \quad \{ \text{ induction hypothesis for } x \ \} \\
& \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{ADD}\ \mathit{sc})\ (\mathit{POP}\ \mathit{fc}))\ \mathit{fc})\ s
\end{array}
$$

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

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

$$exec\ (comp\ x)\ s \quad = \quad \mathbf{case}\ eval\ x\ \mathbf{of} \\ \qquad\qquad Just\ n \quad \rightarrow VAL\ n : s \\ \qquad\qquad Nothing \rightarrow s \tag{11}$$

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:

$$
\begin{array}{ll}
& exec\ (comp\ x)\ s \\
= & \quad \{\ \text{specification (11)}\ \} \\
& \mathbf{case}\ eval\ x\ \mathbf{of} \\
& \quad Just\ n \quad \rightarrow VAL\ n : s \\
& \quad Nothing \rightarrow s \\
= & \quad \{\ \text{define: } exec\ HALT\ s = s\ \} \\
& \mathbf{case}\ eval\ x\ \mathbf{of} \\
& \quad Just\ n \quad \rightarrow exec\ HALT\ (VAL\ n : s) \\
& \quad Nothing \rightarrow exec\ HALT\ s \\
= & \quad \{\ \text{specification (10)}\ \} \\
& exec\ (comp'\ x\ HALT\ HALT)\ s
\end{array}
$$

We could also have introduced a special-purpose code constructor for the failure case, say $exec\ CRASH\ s = s$, but for our simple exception language, it suffices to use *HALT* for both cases. However, for a more sophisticated source language that features different kinds of exceptions, using such an additional constructor may be important.

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

Target language:

$$\mathbf{data}\ Code = HALT \mid PUSH\ Int\ Code \mid ADD\ Code \mid POP\ Code$$

Compiler:

$$
\begin{array}{ll}
comp & :: Expr \rightarrow Code \\
comp\ x & = comp'\ x\ HALT\ HALT \\[4pt]
comp' & :: Expr \rightarrow Code \rightarrow Code \rightarrow Code \\
comp'\ (Val\ n)\ sc\ fc & = PUSH\ n\ sc \\
comp'\ (Add\ x\ y)\ sc\ fc & = comp'\ x\ (comp'\ y\ (ADD\ sc)\ (POP\ fc))\ fc \\
comp'\ Throw\ sc\ fc & = fc \\
comp'\ (Catch\ x\ h)\ sc\ fc & = comp'\ x\ sc\ (comp'\ h\ sc\ fc)
\end{array}
$$

Virtual machine:

$$
\begin{array}{ll}
exec & :: Code \rightarrow Stack \rightarrow Stack \\
exec\ HALT\ s & = s \\
exec\ (PUSH\ n\ c)\ s & = exec\ c\ (VAL\ n : s)
\end{array}
$$

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

$$
\begin{aligned}
&exec\ (ADD\ c)\ (VAL\ m : VAL\ n : s) = exec\ c\ (VAL\ (n+m) : s)\\
&exec\ (POP\ c)\ (VAL\ \_ : s) \qquad\qquad\ = exec\ c\ s
\end{aligned}
$$

3.3 *Reflection*

We conclude this section with some comments on the two approaches to calculate 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. Such knowledge is not available if we consider, for example, a higher-order language, as we shall do in Section 5. In contrast, the approach using two code continuations exploited that we do have such static knowledge, in the form of the failure continuation.

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:

$$
exec\ (comp'\ x\ sc\ fc)\ s \quad = \quad \mathbf{case}\ eval\ x\ \mathbf{of}
\begin{array}{l}
\\
Just\ n \ \ \rightarrow exec\ sc\ (VAL\ n : s)\\
Nothing \rightarrow fail\ fc\ s
\end{array}
$$

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 no handler argument is 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 grows, 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. Part of the difficulty of determining an appropriate induction hypothesis lies in the fact that it may need to explicitly refer to details of the virtual machine implementation. By using a partial specification, these details are left open and are instead derived during the calculation, such as the function *fail* that defines the behaviour of the virtual machine when an exception is thrown.

- We used a *partial definition* for the *Stack* type. This technique is crucial for more sophisticated languages. While our approach 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 reference cell that stores an integer value:

type *State* = *Int*
data *Expr* = *Val Int* | *Add Expr Expr* | *Throw* | *Catch Expr Expr* | *Get* | *Put Expr Expr*

Informally, *Get* returns the current value of the reference cell, while *Put* x y sets the cell to the value of the expression x and then behaves as the expression y. Alternatively, we could have chosen *Put* to take one argument and instead have an additional sequencing operator *Seq* that takes two arguments. However, we prefer to keep the source language small in order to focus on the essence of the problem.

The addition of state is particularly 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. For brevity, we refer to the former case as *global state*, and the latter as *local state*.

We shall calculate a compiler for the global state semantics. The calculation for the local state semantics is similar and can be found in the appendix that forms part of the online supplementary material. Our calculations are based upon the 'one continuation' approach from Section 3.1, but we could just as well use any other approach from Section 3.

4.1 Specification

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:

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

That is, no matter whether an exception is thrown or not, *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. We write the state as q, reserving the use of the symbol s for stacks throughout the article for consistency:

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

Note that in the case for *Catch*, when the handler h is invoked, it uses the state q' 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,

$$comp' :: Expr \rightarrow Code \rightarrow Code$$

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:

$$exec :: Code \rightarrow Conf \rightarrow Conf$$

$$\mathbf{type}\ Conf = (Stack, State)$$

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* transforms 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:

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

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:

$$\begin{array}{lll} exec\ (comp'\ x\ c)\ (s,q) & = & \textbf{case}\ eval\ x\ q\ \textbf{of} \\ & & \quad (Just\ n,q') \quad \rightarrow exec\ c\ (VAL\ n : s,q') \\ & & \quad (Nothing,q') \rightarrow fail\ (s,q') \end{array} \qquad (12)$$

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 $Conf \rightarrow Conf$. In a similar manner to Section 3.1, if $fail$ took x and c as additional arguments, our calculation would get stuck.

4.2 Calculation

We now calculate a definition for $comp'$ from the specification by constructive induction on x, during which we also derive $fail$. The cases for Val and $Throw$ are easy as usual:

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

and

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

The cases for Add and $Catch$ proceed along similar lines to Section 3.1. The calculations can be found in the appendix in the online supplementary material.

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:

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

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:

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

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:

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

Compiler:

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

Virtual machine:

$$
\begin{array}{ll}
\mathbf{data}\ Elem & = VAL\ Int \mid HAN\ Code \\
exec & :: Code \to Conf \to Conf
\end{array}
$$

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

$$
\begin{array}{ll}
exec\ HALT\ (s,q) & = (s,q) \\
exec\ (PUSH\ \ n\ c)\ (s,q) & = exec\ c\ (VAL\ n : s,q) \\
exec\ (ADD\ \ c)\ (VAL\ m : VAL\ n : s,q) & = exec\ c\ (VAL\ (n+m) : s,q) \\
exec\ FAIL\ (s,q) & = fail\ (s,q) \\
exec\ (MARK\ \ h\ c)\ (s,q) & = exec\ c\ (HAN\ \ h : s,q) \\
exec\ (UNMARK\ \ c)\ (VAL\ n : HAN\ \ \_ : s,q) & = exec\ c\ (VAL\ n : s,q) \\
exec\ (LOAD\ \ c)\ (s,q) & = exec\ c\ (VAL\ q : s,q) \\
exec\ (SAVE\ \ c)\ (VAL\ n : s,q) & = exec\ c\ (s,n) \\
fail & :: Conf \rightarrow Conf \\
fail\ ([\,],q) & = ([\,],q) \\
fail\ (VAL\ n : s,q) & = fail\ (s,q) \\
fail\ (HAN\ \ h : s,q) & = exec\ h\ (s,q)
\end{array}
$$

4.3 Reflection

Configurations. The introduction of state only required a single refinement to our approach: instead of operating on a stack, the virtual machine *exec* now operates on a configurations 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 include 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 versus local. The calculation for the local state semantics is very similar to the calculation for the global state semantics presented in this section. In fact 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 other approaches such as call-by-name or call-by-need. We will also see two further refinements of the calculation process: the use of defunctionalisation to transform the semantics into a first-order form, and the use of relational semantics rather than functional semantics.

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, which are not difficult but would be distracting to the presentation, we represent variables using de Bruijn indices:

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

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 $\lambda n \rightarrow (\lambda m \rightarrow n + m)$ that adds two integer values is represented as follows:

$$\begin{array}{l} add :: Expr \\ add = Abs\ (Abs\ (Add\ (Var\ 1)\ (Var\ 0))) \end{array}$$

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:

$$\mathbf{data}\ Value = Num\ Int \mid Fun\ (Value \rightarrow Value)$$

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

$$\mathbf{type}\ Env = [Value]$$

It is now straightforward to define a function that evaluates an expression to a value in the context of a given environment:

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

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

For example, applying *eval* to the expression $App\ (App\ add\ (Val\ 1))\ (Val\ 2)$ and the empty environment [] gives the result $Num\ 3$, 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 $Value \to Value$. However, this problem is easily addressed using defunctionalisation, which introduces a new data 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 $\lambda v \to eval\ x\ (v : e)$. 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:

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

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 $Value \to Value$ can be formalised by defining a function that maps from one to the other:

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

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*. This changes the definition of *eval* for the *Abs* and *App* cases as follows:

$$\begin{array}{ll} eval\ (Abs\ x)\ e & = Fun\ (Clo\ x\ e) \\ eval\ (App\ x\ y)\ e & = \textbf{case}\ eval\ x\ e\ \textbf{of} \\ & \qquad Fun\ c \to apply\ c\ (eval\ y\ e) \end{array}$$

The other cases for the function *eval* remain unchanged. Moreover, the definition of the *Value* type uses the type *Lam* instead of $Value \to Value$:

$$\textbf{data}\ Value = Num\ Int \mid Fun\ Lam$$

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

$$\begin{array}{ll} \textbf{data}\ Value & = Num\ Int \mid Clo\ Expr\ Env \\ eval & :: Expr \to Env \to Value \\ eval\ (Val\ n)\ e & = Num\ n \\ eval\ (Add\ x\ y)\ e & = \textbf{case}\ eval\ x\ e\ \textbf{of} \end{array}$$

$$
\begin{aligned}
& \qquad\qquad\qquad\qquad Num\ n \to \textbf{case}\ eval\ y\ e\ \textbf{of} \\
& \qquad\qquad\qquad\qquad\qquad\qquad Num\ m \to Num\ (n+m) \\
& eval\ (Var\ i)\ e \quad\ \ = e\ !!\ i \\
& eval\ (Abs\ x)\ e \quad = Clo\ x\ e \\
& eval\ (App\ x\ y)\ e = \textbf{case}\ eval\ x\ e\ \textbf{of} \\
& \qquad\qquad\qquad\qquad Clo\ x'\ e' \to eval\ x'\ (eval\ y\ e : e')
\end{aligned}
$$

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* (Winskel, 1993).

The use of rule induction is another refinement of our calculation methodology. In order to make this use of rule induction explicit, we reformulate the 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 the *eval* function in relational style:

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

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

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:

$$
\begin{aligned}
& comp' :: Expr \to Code \to Code \\
& exec \ \ :: Code \to Conf \to Conf
\end{aligned}
$$

However, because the semantics now requires the use of an 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:

$$
\begin{aligned}
& \textbf{type}\ Stack = [Elem] \\
& \textbf{data}\ Elem = VAL\ Value
\end{aligned}
$$

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:

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

Note that the precondition $x \Downarrow_e v$ means that the specification only applies to lambda expressions whose evaluation terminates; we will return to this issue in Section 5.5. It is straightforward to calculate a compiler from the above 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:

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

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

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

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:

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

We lift $conv$ to environments by mapping the function over the list of values:

$$\begin{aligned}
&conv_{\mathsf{E}} && :: Env \to Env' \\
&conv_{\mathsf{E}}\ e && = map\ conv\ e
\end{aligned}$$

Using these ideas, it is now straightforward to modify the specification for the compilation function $comp'$ to take care of the necessary type conversions:

$$exec\ (comp'\ x\ c)\ (s, conv_{\mathsf{E}}\ e) = exec\ c\ (VAL\ (conv\ v) : s, conv_{\mathsf{E}}\ e) \qquad \text{if } x \Downarrow_e v \quad (13)$$

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

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 $exec\ (comp'\ x\ c)\ (s, conv_{\mathsf{E}}\ e)$ of the equation into the form $exec\ c'\ (s, conv_{\mathsf{E}}\ e)$ for some code c', from which we can then conclude that the definition $comp'\ x\ c = c'$ 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 *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:

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

and

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

The indexing lemma used above is that $f\ (xs\ !!\ i) = (map\ f\ xs)\ !!\ i$, for any strict function f, list xs, and index i of the appropriate types. This lemma, which arises as the free theorem (Wadler, 1989) for the type of !!, allows us to generalise over $conv_{\mathsf{E}}\ e$ 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*:

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

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:

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

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:

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

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

$$conv\ (Clo\ x'\ e') = Clo'\ (comp'\ x'\ (RET\ c))\ (conv_{\mathsf{E}}\ e')$$

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. We

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

now see 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

$$exec\ (RET\ c)\ (VAL\ u : ENV\ d : s, \_) = exec\ c\ (VAL\ u : s, d)$$

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:

$$exec\ RET\ (VAL\ u : CLO\ c\ d : s, \_) = exec\ c\ (VAL\ u : s, d)$$

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:

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

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:

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

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 nullary code constructor $HALT$ that returns the current configuration.

Target language:

$$
\begin{aligned}
\mathbf{data}\ Code = {} & HALT \mid PUSH\ Int\ Code \mid ADD\ Code \mid LOOKUP\ Int\ Code \mid \\
& ABS\ Code\ Code \mid RET \mid APP\ Code
\end{aligned}
$$

Compiler:

$$
\begin{array}{ll}
comp & :: Expr \rightarrow Code \\
comp\ x & = comp'\ x\ HALT \\[4pt]
comp' & :: Expr \rightarrow Code \rightarrow Code \\
comp'\ (Val\ n)\ c & = PUSH\ n\ c \\
comp'\ (Add\ x\ y)\ c & = comp'\ x\ (comp'\ y\ (ADD\ c)) \\
comp'\ (Var\ i)\ c & = LOOKUP\ i\ c \\
comp'\ (Abs\ x)\ c & = ABS\ (comp'\ x\ RET)\ c \\
comp'\ (App\ x\ y)\ c & = comp'\ x\ (comp'\ y\ (APP\ c))
\end{array}
$$

Virtual machine:

$$
\begin{array}{ll}
\mathbf{data}\ Elem & = VAL\ Value' \mid CLO\ Code\ Env' \\[4pt]
exec & :: Code \rightarrow Conf \rightarrow Conf \\
exec\ HALT\ (s, e) & = (s, e) \\
exec\ (PUSH\ n\ c)\ (s, e) & = exec\ c\ (VAL\ (Num'\ n) : s, e) \\
exec\ (ADD\ c)\ (VAL\ (Num'\ m) : VAL\ (Num'\ n) : s, e) & = exec\ c\ (VAL\ (Num'\ (n + m)) : s, e) \\
exec\ (LOOKUP\ i\ c)\ (s, e) & = exec\ c\ (VAL\ (e\ !!\ i) : s, e) \\
exec\ (ABS\ c'\ c)\ (s, e) & = exec\ c\ (VAL\ (Clo'\ c'\ e) : s, e) \\
exec\ RET\ (VAL\ v : CLO\ c\ e : s, \_) & = exec\ c\ (VAL\ v : s, e) \\
exec\ (APP\ c)\ (VAL\ v : VAL\ (Clo'\ c'\ e') : s, e) & = exec\ c'\ (CLO\ c\ e : s, v : e')
\end{array}
$$

Conversion function:

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

The above compiler is essentially the same as that presented in Day & Hutton (2014), except that it has now been calculated directly from a specification of its correctness.

Note that the code produced by the compiler is not fully linear. Similarly to the *MARK* instruction in the compiler for exceptions that we calculated earlier, the *ABS* instruction takes two arguments of type *Code*. However, if desired we can transform the compiler to produce linear code in a similar manner to that described in Section 3.1.

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 rather than structural induction as the basis for the calculation.

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press

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. Alternatively, we could have made the partiality explicit by rewriting the functional semantics in monadic style using the *Maybe* monad. However, 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 typed 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 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.

We have included calculations for call-by-name and call-by-need semantics as well as a call-by-value lambda calculus with exceptions in the supplementary material.

6 Related work

As noted at the start of this article, the ability 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 CP, 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, 1982a). The derivation of compilers was first considered by Wand (1982a). 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.

The first difference from our approach is that Wand begins with a semantics that is already rather operational in style, in the form of a continuation semantics. The use of continuations can make semantic definitions more complicated, which in turn makes it more difficult to argue that they are ‘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. The third important difference is the role of correctness proofs. While the original article did not consider correctness proofs, in a later article, (Wand, 1982b) does sketch an argument to prove his compilers correct. By contrast, in our approach the correctness property is the starting point for the derivation process: the derivation of the compiler and proof of its correctness proceed simultaneously so that each informs the other.

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 CPS 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. This process involves transformation steps such as ‘make the definition compositional’ and ‘factorize into a composition of combinators and recursive calls’. While the authors show how these transformation can be performed for particular examples, how they may be applied more generally is not considered. 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 CPS, 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 fix-point 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 fix-point induction or guess an induction hypothesis.

In his calculation for the non-deterministic language, Meijer also uses CPS. 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.

1. Define a semantics for the language:
   - Define an evaluation function
   - 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 derivation given by Sestoft (1997) 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 (1997)'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 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 the compiler specification, and to calculate the compiler by structural induction on the language syntax. Moreover, it is advantageous to define the semantics in a compositional style, because the use of rule induction places additional restrictions on the format of the semantics as discussed in Section 5.5. 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;
- *Formalisation* – it is readily amenable to mechanised formalisation, and all the calculations in the article have been mechanically verified using the Coq system, with the proofs being available online as supplementary material.

Note that the formalisation in Coq is not restricted to post-hoc verification of calculations that have been performed by hand. The calculation style presented in this article can be emulated in Coq by using partial definitions, and in this way Coq can be used as an interactive tool to derive correct-by-construction compilers. The Coq system not only guides the user through the calculation process, but also checks its correctness. Moreover, using Coq's code extraction facility (Letouzey, 2008), we can extract the compiler and the virtual machine implementation fully automatically if so desired.

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; considering further language features such as (delimited) continuations and concurrency; exploring additional compilation concepts such as optimisation and modularity and applying the technique to larger languages.

Acknowledgements

Discussions with Ralf Hinze at the IFIP Working Group 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 Jeremy Gibbons, Colin Runciman, Neil Sculthorpe, Arjan Boeijink, the JFP referees and the Functional Programming lab in Nottingham for many useful comments and suggestions. This work has been partially funded by the Danish Council for Independent Research, Grant 12-132365.

References

Adams, N., Kranz, D., Kelsey, R., Rees, J., Hudak, P. & Philbin, J. (1986) ORBIT: An optimizing compiler for scheme. In Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction. New York, NY, USA: ACM, pp. 219–233.

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

Ager, M. S., Biernacki, D., Danvy, O. & Midtgaard, J. (2003b) *From Interpreter to Compiler and Virtual Machine: A Functional Derivation*. Technical Report RS-03-14. BRICS, Department of Computer Science, Aarhus, Denmark: University of Aarhus.

Appel, A. (1991) *Compiling with Continuations*. New York, NY, USA: Cambridge University Press.

Backhouse, R. (2003) *Program Construction: Calculating Implementations from Specifications*. West Sussex, UK: John Wiley and Sons, Inc.

Bahr, P. (2014) Proving correctness of compilers using structured graphs. *Functional and Logic Programming*, Lecture Notes in Computer Science, vol. 8475. Springer International Publishing, pp. 221–237.

Chase, D. (1994a) Implementation of exception handling, Part I. *J. C Lang. Transl.* **5**(4), 229–240.

Chase, D. (1994b) Implementation of exception handling, Part II. *J. C Lang. Transl.* **6**(1), 20–32.

Day, L. E. & Hutton, G. (2014) Compilation à la Carte. In Proceedings of the 25th Symposium on Implementation and Application of Functional Languages. New York, NY, USA: ACM, pp. 13–24.

Dybjer, P. (1994) Inductive families. *Formal Asp. Comput.* **6**(4), 440–465.

Hutton, G. (2007) *Programming in Haskell*. New York, NY, USA: Cambridge University Press.

Hutton, G. & Wright, J. (2004) Compiling exceptions correctly. *Mathematics of Program Construction*, Lecture Notes in Computer Science, vol. 3125. Berlin/Heidelberg: Springer, pp. 211–227.

Hutton, G. & Wright, J. (2007) What is the meaning of these constant interruptions? *J. Funct. Program.* **17**(06), 777–792.

Letouzey, P. (2008) Extraction in Coq: An overview. Logic and Theory of Algorithms: 4th Conference on Computability in Europe, Lecture Notes in Computer Science, vol. 5028. Berlin, Germany: Springer-Verlag.

McCarthy, J. & Painter, J. (1967) Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, Proceedings of Symposia in Applied Mathematics, vol. 19. American Mathematical Society, Providence, RI, USA: pp. 33–41.

McKinna, J. & Wright, J. (2006) *A Type-Correct, Stack-Safe, Provably Correct Expression Compiler in Epigram*. Unpublished manuscript.

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

Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. In Proceedings of the ACM Annual Conference. New York, NY, USA: ACM, pp. 717–740.

Sculthorpe, N., Farmer, A. & Gill, A. (2013) The HERMIT in the tree: Mechanizing program transformations in the GHC Core language. Implementation and Application of Functional Languages 2012, Lecture Notes in Computer Science, vol. 8241. New York, NY, USA: Springer.

Sestoft, P. (1997) Deriving a lazy abstract machine. *J. Funct. Program.* **7**(03), 231–264.

Spivey, M. (1990) A functional theory of exceptions. *Sci. Comput. Program.* **14**(1), 25–42.

Steele, Jr., G. L. (1978) *Rabbit: A Compiler for Scheme*. Technical Report AI-TR-474. Cambridge, MA, USA: MIT AI Lab.

Thielecke, H. (2002) Comparing control constructs by double-barrelled CPS. *Higher-Order Symb. Comput.* **15**(2–3), 141–160.

Wadler, P. (1989) Theorems for free! In Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture. New York, NY, USA: ACM Press.

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

Wand, M. (1982b) Semantics-directed machine architecture. In Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM, pp. 234–241.

Winskel, G. (1993) *The Formal Semantics of Programming Languages – An Introduction*, Foundation of Computing Series. Cambridge, MA, USA: MIT Press.

https://doi.org/10.1017/S0956796815000180 Published online by Cambridge University Press