Calculating a correct compiler using the Bahr and Hutton method

Marco Jones

April 25, 2016

Contents

1 Introduction

Ever since the invention of electronic computers the instruction set to operate them has grown massively. At the lowest level, computers directly execute a series of discrete binary electrical pulses using logical circuits to perform computation.

These binary signals are far from random; they're formally defined in a grammar, making it a *language*, more specifically the language of "Machine code" instructions. Managing operations in machine code is difficult on a large scale, so by abstracting away from the details of how a computer works, programmers have developed higher level programming languages; these languages are easier for programmers to interpret than machine code and that makes them much easier to use and bug fix. However a computer still only understands machine code[1], so clearly there is a gap to bridge in translating the *source code* of a program, which can't be executed, to machine code which can be executed.

There are two common approaches to making this translation: interpretation and compilation, which are *not mutually exclusive*, some languages use a mix of both strategies e.g. Python and C.

Interpretation involves using a interpreter program which takes the abstract syntax of a source program and immediately executes that to produce an output, these languages are known as "Interpreted" languages.

In contrast there are compiled languages, these use a program called a compiler to translate the high level source code into code of a lower level target language, this target language is not necessarily machine code. The end goal is to translate the source code into machine code but there may be several intermediate languages in the process[1, pg 8-10]. Compiled languages tend to be more efficient because the compiled code is easier the computer to translate and execute rather than use another program to interpret the source code on the machines' behalf.

Compilers, like any program, need to go through testing to verify functionality and expose bugs. However it can be much more difficult to do this for compilers because the bugs might not be in the implementation of compiler itself but the code that it may output. Time can be saved in both testing and development by merging the two; construct a compiler using mathematical rules and you can be more assured of its correctness, this field is called compiler calculation[2], but it's quite an advanced topic; requiring a decent understanding of the formal mathematics behind it.

Bahr and Hutton have built upon these techniques and shown that compiler calculation can be much simpler, at least for compilers that produce code for stack-based virtual machines. The virtual machine takes the place of the computer by executing code with a stack to manipulate arguments, in a real computer this can be likened to memory, in that values are stored and manipulated in this space.

[1] Assembly code can be executed by a computer with the use of an assembler, which translates the assembly into machine code nonetheless.

For a given source language, the Bahr and Hutton method produces a compiler and corresponding virtual machine via equational reasoning; with all of the definitions to compile source expressions and the code to execute them, emerging out of the calculation process. Furthermore calculation process makes use of structural recursion, where the semantics of source expressions are defined by the semantics of their arguments. This allows the use of inductive methods to do calculations efficiently whilst simultaneously guaranteeing their correctness by virtue of *constructive induction* [3].

Through the inductive process we discover and invent definitions for a compiler and corresponding virtual machine without needing to specify how the machine should operate before hand. The end result, is an equational program consisting of definitions of the compile function and virtual machine, and any axillary functions we create in the process.

1.1 Aims and methodology

The aim of this dissertation is to see whether the Bahr and Hutton method can be used to calculate a new and correct compiler with a corresponding virtual machine, not defined in Bahr and Hutton's paper [4]. The compiler and virtual machine we will calculate will be for a source language which extends upon the Arithmetic language example[4, Section 2], and will do so in three stages.

Firstly by summarising the Bahr and Hutton method, using the arithmetic language derivation as described in, as a guide.

Secondly, the language will be gradually extended by calculating new definitions of a more complex nature, and implementing them in Haskell [2]. Each extension of the language will be labelled as it's own language, it's compiler and virtual machine will be provided in it's own .hs file supporting this document. We will see the Bahr and Hutton method used to calculate the language of: conditionals, variable bindings and function definitions.

Each calculation we will be briefly tested using an example expression in three different ways.

1. Compare hand compiled code against the implemented compiler.
2. Compare results of hand executed code against that of the virtual machine.
3. Compare outputs of the virtual machine against the interpreter.

Thirdly, we will use automated testing to verify the definitions in the final and most extended language; by construction, the automated tests should verify all calculations up to and including the should all tests pass, we will be even more confident in our compiler's correctness.

Finally, we will conclude with reflection on using the Bahr and Hutton method, and further work.

[2] Haskell provides curried function application and explicit type declaration which are convenient for defining grammars, as consequence, the implementation closely resembles our calculations

1.2 Roadmap

§2 will review the Bahr and Hutton method and a short literature review on compiler proof. The Bahr and Huttonmethod is a process of constructing a correct compiler from the beginning; its construction serves as it's proof, however compiler proof is a related topic.

In §3 will see the extension of the compiler and virtual machine with definitions for a conditional function

In §4 the compiler and virtual machine will be extended once more with definitions for variable declaration.

In §5 will be a discussion on what would be expected of a compiler and virtual machine that could handle function definition and application. This would have been good to calculate, however this dissertation did not get that far.

In §6 we will discuss the methods of systematically testing the compiler and virtual machine. The automated testing will all be in one section of it's own near the end of this dissertation because it will aim to test the end product compiler and virtual machine. Moreover the language defined by this compiler and virtual machine can be thought of as a set union of every sub-languages, therefore by transitivity, testing the whole set of languages together also tests the individual functionality of every sub language.

§7 is the conclusion where there will be reflection on the use of the Bahr and Hutton method and suggestions for further work.

2 A review of the Bahr and Hutton method

Bahr and Hutton begin their paper §2.1 - §2.4 of Bahr and Hutton's paper describe the unrefined method in detail, only to refine the process in §2.5 [4, Combining the transformation steps], resulting in a much simpler 3 step process, this paper will only be concerned with their refined 3 step method[4, page 12]. Here are the 3 steps:

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

In §2 they are deriving a compiler and virtual machine for the "Arithmetic" language, they begin by defining: a new Haskell data type *Expr*; which contain the set of expressions which belong to their source language, an evaluation function, also referred to as the *interpreter*, which defines their semantics, and a stack of integers where arguments are manipulated.

2.1 Values and addition

$$\begin{aligned}
\mathbf{type}\ \mathit{Stack} &= [\mathit{Int}\ ] \\
\mathbf{data}\mathit{Expr} &= \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\
\mathit{eval} &:: \mathit{Expr}\ \to \mathit{Int} \\
\mathit{eval}\ (\mathit{Val}\ n) &= n \qquad (1) \\
\mathit{eval}\ (\mathit{Add}\ x\ y) &= \mathit{eval}\ x + \mathit{eval}\ y \qquad (2)
\end{aligned}$$

In Haskell **data** creates a new type, here we define *Expr* as either being a *Val* or an *Add*, these tags are called *constructors*, a Val constructor will always be followed by an integer (which is "Int" in Haskell), and an Add will always be followed by two more expressions. It is the constructor *together* with it's arguments that make it of **type** expression, i.e *Val* n or *Add* x y, where n is an Int and x and y are $Expr$, if the constructors are not followed by . However because of curried function application, we cannot simply write $eval\ Val\ n$ or $eval\ Add\ x\ y$, because that applies $eval$to 2 and 3 arguments respectively, thus we package each expression into a single arguments by using parentheses, so long as each "package" is a valid *Expr*, the function application will be type correct and we may continue e.g.

$$\mathit{eval}\ (\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2))$$

On the right hand side of the equations is a description of how to compute the result of what *eval* is applied to. Evaluating a *Val* expression simply returns n on the other hand evaluating an *Add* is recursively defined, as we do not yet know the values of $eval\ x$ and $eval\ y$; Bahr and Hutton are defining the semantics of *Add x y compositionally* by the semantics of each of it's argument expressions, x and y.

Making the semantics compositional allows the use of *inductive* derivations, this means that for a given function applied to an expression we assume it's *type correct* and seek to find a definition for it, which will hold true in all cases regardless of what it's arguments are. Bahr and Hutton explore when this is not possible, but that is beyond the aim of this project [4].

In §2.1 - §2.4 Bahr and Hutton *derived* four components and two correctness equations[4] [page 9]:

- A data type *Code* that represents code for the virtual machine.
- A function $comp\ ::\ Expr\ \rightarrow Code$ that compiles source expressions to code.
- A function $comp'\ ::\ Expr\ \rightarrow\ Code\ \rightarrow\ Code$ that also takes a code continuation as input.
- A function $exec\ ::\ Code\ \rightarrow Stack\ \rightarrow Stack$ that provides a semantics for code by modifying a run-time stack.

$$\begin{aligned} exec\ (comp\ x)\ s &= eval\ x : s & (3) \\ exec\ (comp'\ x\ c)\ s &= exec\ c\ (eval\ x : s) & (4) \end{aligned}$$

Calculations begin in the form the specification of compiler correctness i.e LHS of equation (4), and proceed by constructive induction on expression x, and aim to re-write it into the form $exec\ c'\ s$ for some code c' from which we can then conclude that the new definition for the compile function: $comp\ x\ c = c'$ which by construction is guaranteed to satisfy the specification because the specification was our start point.

It is perhaps simple in appearance, but this equation specifies the correctness of compilation of our entire source code, moreover all of the source code is contained within c and we compile all of it by recursively calling the *comp'* function. Bundling the source code into a single variable may seem optimistic, but a lot of compilers nowadays take entire programs as a (possibly huge) string of characters and it is often up to a preprocessor to collect them together into tokens, and analyse syntax, this process is done a single letter or symbol at a time[5]. We won't be using special syntax in our language and so won't require a parser, instead our functions are Haskell constructors, which state the type of arguments and gives us an *abstract* syntax.

2.2 Calculations

To introduce the Bahr and Hutton method, we will follow the calculation of *comp* and *exec* definitions for *Val* and *Add* expressions[4, §2.5].

2.2.1 Values

Calculations begin with the source expression in question substituted for x in the compiler specification (4). To have values in our language we will make the constructor $Val\ n$, the calculation proceeds as follows[4]:

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

Now there are no further definitions to apply, inventing a definition for *exec* that solves this equation will allow us to proceed:

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

Currently the calculation is the form of the right hand side of this equation, however c and n are now *unbound* along with the new variable c'; they only appear on one side of the equation, so this equation cannot be used as a definition for *exec*. This is similar to declaring unknown variables in algebra, one cannot use an unknown variable to define another without also making it's value unknown, in the same way we can't use expressions with unbound variables to define other expressions, e.g $b = a + c\ |where\ c = 1$, we cannot know the value of y if x is not given.

Therefore to solve this equation we are to define what c' is in terms of the other unbound variables c and n, c' is of type *Code* so with a new instruction that takes n and c as arguments we can bind them both[4, bottom of page 9]

$$\begin{aligned} PUSH \quad & :: \quad Int \rightarrow Code \rightarrow Code \\ exec\ (PUSH\ n\ c)\ s \quad & = \quad exec\ c\ (n : s) \end{aligned} \tag{5}$$

This defines a definition for *exec*, that executing the code $(PUSH\ n\ c)$ pushes the value n onto the stack s then executes the code c by making use of a recursive definition.

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

Our equation is now back to it's original form $exec\ c'\ s$ where $c' = PUSH\ n\ c$. We began from $exec\ (comp'\ (Val\ n)\ c)\ s$, and every equation was valid in the derivation, therefore it is safe to conclude that

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

or more specifically, we have *discovered* a definition for the compiler

$$comp'\ (Val\ n)\ c = PUSH\ n\ c \tag{6}$$

Discovering this definition is the whole point of the calculation process. Just from the definition of the high level semantics of a *Val* source expression, and the specification of compiler correctness, we've come across the exact situation where we need a new instruction to solve a specific problem, and then made that instruction. Furthermore the calculation as a whole serves as a proof for an implementation of these definitions.

2.2.2 Addition

Next Bahr and Hutton calculate definitions for the inductive case, *Add x y*, it's called inductive because the values of x and y are not yet known however it is assumed that they are expressions as well, otherwise there would be a type error.

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

Again the calculation is stuck, however similar to before, we can invent a new definition for *exec* which will allow us to continue. Moreover being an inductive case, we can make use of the induction hypotheses for x and y, these hypotheses are equations in the form of the equation for compiler correctness 4, with specific values for x and s, they formally specify what to compile in order to achieve an assumed stack state.

$$\mathit{exec}\ (\mathit{comp}'\ x\ c)\ s \quad = \quad \mathit{exec}\ c\ (\mathit{eval}\ x : s) \tag{7}$$

$$\mathit{exec}\ (\mathit{comp}'\ y\ c)\ s \quad = \quad \mathit{exec}\ c\ (\mathit{eval}\ y : s) \tag{8}$$

In this case the hypotheses are similar; the hypothesis for x assumes that *eval* x is on top of the stack, whereas the hypothesis for y assumes *eval* y is on top. To be able to use either, we must manipulate our stack into one of the forms on the RHS on the induction hypotheses. The aim is to have both evaluations of x and y to make the addition, therefore both hypotheses need to be used, one after the other. With this in mind, the stack needs both *eval* x and y on top, i.e. $\mathit{eval}\ x : \mathit{eval}\ y : s$ (or vice versa). Just like with *Val*, we can invent a new instruction to get the stack into that state, in doing so, it will solve the equation:

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

So we define ADD that when executed, adds the top two stack elements together and leaves the result on top of the stack.

$$\mathit{ADD} \quad :: \quad \mathit{Code} \rightarrow \mathit{Code}$$

$$\mathit{exec}\ (\mathit{ADD}\ c)\ (m : n : s) \quad = \quad \mathit{exec}\ c\ (n + m : s) \tag{9}$$

Ordering is not important in this case; it is a matter of choice. Bahr and Hutton mention here that their choice is to use left-to-right evaluation by pushing *eval x* on first, for consistency, we will use their definition.

With the new definition for the *exec* we can continue the calculation

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

The final expression is now in the form $\mathit{exec}\ c'\ s$, we started from $\mathit{exec}\ (\mathit{comp}'\ (\mathit{Add}\ x\, y)\ c)\ s$ which means we can conclude that

$$\begin{aligned}
\mathit{exec}\ c'\ s &= \mathit{exec}\ (\mathit{comp}'\ (\mathit{Add}\ x\ y)\ c)\ s\\
\text{where } c' &= \mathit{comp}'\ y\ (\mathit{comp}'\ x\ (\mathit{ADD}\ c))
\end{aligned}$$

In summary, we have discovered a definition for the compiler

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

Again the goal of finding a new definition for the compiler and virtual machine has been achieved, from the definition of the high level semantics of an *Add* source expression, and the specification of compiler correctness. However this time, there were argument expressions x and y in the expression being investigated, *Add*. These expressions needed to be compiled and executed all the same, the induction hypotheses told us what state the stack needed to be in to compile those expressions, so a definitive for *exec* for invented to achieve that stack state. From now on we will continue to use this same method to expand our compiler and virtual machine for more source code to expand our arithmetic language.

In summary Bahr and Hutton calculated the following definitions[3] for the

[3] The instruction HALT simply returns the current state of the stack, I didn't include Bahr and Hutton's derivation of HALT for brevity because it's only a small point

compiler and virtual machine:

$$\begin{array}{rcll}
\textbf{data } \mathit{Code} & = & \mathit{HALT}|\mathit{PUSH\ Int\ Code}|\mathit{ADD\ Code} & \\
\mathit{comp} & :: & \mathit{Expr}\ \rightarrow \mathit{Code} & \\
\mathit{comp}\ x & = & \mathit{comp}'\ x\ \mathit{HALT} & (10) \\
\mathit{comp}' & :: & \mathit{Expr}\ \rightarrow \mathit{Code} \rightarrow \mathit{Code} & \\
\mathit{comp}'\ (\mathit{Val}\ n)\ c & = & \mathit{PUSH}\ n\ c & (11) \\
\mathit{comp}'\ (\mathit{Add}\ x\ y)\ c & = & \mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{ADD}\ c)) & (12) \\
\mathit{exec} & :: & \mathit{Code} \rightarrow \mathit{Stack}\ \rightarrow \mathit{Stack} & \\
\mathit{exec}\ \mathit{HALT} s & = & s & (13) \\
\mathit{exec}\ (\mathit{PUSH}\ n\ c)\ s & = & \mathit{exec}\ c\ (n : s) & (14) \\
\mathit{exec}\ (\mathit{ADD}\ c)\ (m : n : s) & = & \mathit{exec}\ c\ ((n + m) : s) & (15)
\end{array}$$

2.3 Testing

This chapter will be concluded with testing of the definitions for Add and Val, with a somewhat complicated example. This will demonstrate the code that the compiler generates, and how the virtual machine executes it.

To check these calculations, we will: 1) calculate and compare by hand the code that is produced and executed by the compiler and virtual machine definitions, against the results using the Haskell implementation compiled by GHC, 2) compare the result of the executed code against the result given by the interpreter. This result of the interpreter test is arguably the most important because if a counter example is found, then our specification for compiler correctness

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

does not always hold, which means all of our calculations were wrong assuming the test expression is valid.

The test expression, α, will be:

$$\alpha = \mathit{Add}\ (\mathit{Add}\ (\mathit{Val}\ 0)\ (\mathit{Val}\ 1))\ (\mathit{Val}\ 2))$$

This expression, will test that not only each expression is compiled properly, but also each sub-expression, this also makes it a test of the recursive definition of the *comp* and *comp'* functions, should they not recurs properly, then only the first *Add* expression will compile and our output code will still contain source expressions. In future we won't need to use such complicated examples, especially when functions become more complex.

2.3.1 Compilation

These equations demonstrate what happens when the definitions of the compile functions are applied to the expression α:

$$
\begin{aligned}
& \mathit{comp}\ (\mathit{Add}\ (\mathit{Add}\ (\mathit{Val}\ 0)\ (\mathit{Val}\ 1))\ (\mathit{Val}\ 2))\\
= & \{\text{definition of } \mathit{comp}\ e\}\\
& \mathit{comp}'\ (\mathit{Add}\ (\mathit{Add}\ (Val\ 0)\ (Val\ 1))\ (Val\ 2))\ (\mathit{HALT})\\
= & \{\text{definition of } \textit{comp' Add}\}\\
& \mathit{comp}'\ (\mathit{Add}\ (\mathit{Val}\ 0)\ (\mathit{Val}\ 1))\ (\mathit{comp}'\ (\mathit{Val}\ 2)(\mathit{ADD\ HALT}))\\
= & \{\text{definitions of } \textit{comp' Add} \text{ and } \mathit{Val}\}\\
& \mathit{comp}'\ (\mathit{Val}\ 0)\ (\mathit{comp}'\ (\mathit{Val}\ 1)\ (\mathit{ADD}\ (\mathit{PUSH}\ 2\ (\mathit{ADD}\ HALT))))\\
= & \{\text{definition of } \textit{comp' Val} \text{ twice}\}\\
& \mathit{PUSH}\ 0\ (\mathit{PUSH}\ 1\ (\mathit{ADD}\ (\mathit{PUSH}\ 2\ (\mathit{ADD\ HALT}))))
\end{aligned}
$$

This agrees with α being compiled using the Haskell implementation of the compiler. GHC: $\mathit{comp}\ \alpha =$

$$
\begin{aligned}
& \mathit{comp}\ (\mathit{Add}\ (\mathit{Add}\ (\mathit{Val}\ 0)\ (\mathit{Val}\ 1))\ (\mathit{Val}\ 2))\\
= & \{\text{Haskell implementation of } \mathit{comp} \text{ and } \textit{comp'}\}\\
& \mathit{PUSH}\ 0\ (\mathit{PUSH}\ 1\ (\mathit{ADD}\ (\mathit{PUSH}\ 2\ (\mathit{ADD\ HALT}))))
\end{aligned}
$$

2.3.2 Execution

Here are definitions of the virtual machine, *exec*, executing the compiled code for the expression α [4]:

$$
\begin{array}{lll}
 & \mathit{exec}\ (\mathit{PUSH}\ 0\ (\mathit{PUSH}\ 1\ (\mathit{ADD}\ (\mathit{PUSH}\ 2\ (\mathit{ADD}\ HALT))))) & [\,]\\
= & \{\text{definition } \mathit{exec}\ \mathit{PUSH}\ \ \text{twice}\} & \\
 & \mathit{exec}\ (\mathit{ADD}\ (\mathit{PUSH}\ 2\ (\mathit{ADD}\ HALT))) & [1,\ 0]\\
= & \{\text{definition of } \mathit{exec}\ \mathit{Add}\} & \\
 & \mathit{exec}\ (\mathit{PUSH}\ 2\ (\mathit{ADD\ HALT})) & [1]\\
= & \{\text{definition of } \mathit{exec}\ \mathit{PUSH}\ \} & \\
 & \mathit{exec}\ (\mathit{ADD\ HALT}) & [2,\ 1]\\
= & \{\text{definition } \mathit{exec}\ \mathit{Add}\} & \\
 & \mathit{exec}\ HALT & [3]\\
= & \{\text{definition } \mathit{exec}\ \mathit{HALT}\ \} & \\
 & [3] &
\end{array}
$$

This agrees with the same expression being executed using the Haskell im-

[4] The left column contains the function followed by the code to be executed, and the right column is the current state of the run-time stack

plementation of the virtual machine. GHC: $exec\ \alpha =$

$$\begin{aligned}
&\quad exec\ (PUSH\ 0\ (PUSH\ 1\ (ADD\ (PUSH\ 2\ (ADD\ HALT)))))\ [\,]\\
&= \{\text{Haskell implementation of } exec\}\\
&\quad [3]
\end{aligned}$$

2.3.3 Interpretation

Applying the definitions of the evaluation function (the interpreter) to the expression $Add\ (Add\ (Val\ 0)\ (Val\ 1))\ (Val\ 2))$:

$$\begin{aligned}
&\quad eval\ (Add\ (Add\ (Val\ 0)\ (Val\ 1))\ (Val\ 2))\\
&= \{\text{definition } eval\ Add\}\\
&\quad eval\ (Add\ (Val\ 0)\ (Val\ 1))\ +\ eval\ (Val\ 2)\\
&= \{\text{definition } eval\ Add \text{ and } Val\}\\
&\quad eval\ (Val\ 0)\ +\ eval\ (Val\ 1)\ +\ 2\\
&= \{\text{definition of } eval\ Val\text{, twice}\}\\
&\quad 0\ +\ 1\ +\ 2\\
&= \{\text{arithmetic}\}\\
&\quad 3
\end{aligned}$$

α being interpreted using the Haskell implementation of the the interpreter GHC: $eval\ \alpha =$

$$\begin{aligned}
&\quad eval\ (Add\ (Add\ (Val\ 0)\ (Val\ 1))\ (Val\ 2))\\
&= \{\text{Haskell implementation of } eval\}\\
&\quad 3
\end{aligned}$$

The results of the execution and interpretation test agree with each other on the same test expression. using the Haskell implementation of, the virtual machine and interpreter, given that the virtual machine outputs a list and the interpreter an integer. GHC: [3], 3

2.4 Summary

We have seen how the basic concepts of the Bahr and Hutton method and how they can be applied to derive definitions for a compiler and virtual machine for a couple simple source expressions, for example, Bahr and Hutton made and applied induction hypotheses (in §2.2.2) to induce the state of the stack just after the point where the calculation was halted. At this stopping point is where they invented a new code constructor and definition for the virtual machine almost effortlessly because they had the equation to solve right in front of them, and they used Haskell code, just a "+" in this case, to define exactly what the virtual machine was to do.

In the previous section, we verified these expressions, at least for the example source expression α. The results of testing the interpreter and virtual machine on α support the compiler correctness specification, because together, they have been shown to satisfy the equation:

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

Where c is just $HALT$, and s is empty [ ].

These definitions of the compiler and virtual machine so far, form the Arithmetic language, which will be referred to as L_a.

The next sections will investigate how far the method can be applied to more source expressions to expand the source language, and what new problems will be encountered.

3 Conditionals, L_c

From now on this dissertation will report on an investigation into applying the Bahr and Hutton method to develop a compiler with definitions not defined in their paper[4].

The first calculation is a derivation definitions for a conditional operator, the purpose of this was to practise the method on an operation only slightly more complicated than the addition operation that Bahr and Hutton derived.

Step 1:

> "define an evaluation function in a compositional manner".

The evaluation function remains the same as before, but we need to define the semantics of our new expression.

Haskell conditionals concrete syntax "if x then y else z", however without a parser to do lexical analysis[5, chapter 2.2] of the lexemes[5], our *source* language cannot use this syntax, so we define ours abstractly. In general conditionals are formed out of three parts: a condition, a true case, and a false case. In our language these will be three expressions that follow an "*Ite*" constructor.

$$\textbf{data}\ \mathit{Expr} = ...|\mathit{Ite}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr}$$

The semantics of *Ite* will be:

$$\mathit{eval}\ (\mathit{Ite}\ x\ y\ z) = \mathit{if}\ \ \mathit{eval}\ x \neq 0\ \mathit{then}\ \mathit{eval}\ y\ \mathit{else}\ \mathit{eval}\ z \tag{16}$$

The condition $\mathit{eval}\ x\ \ \neq 0$ is very basic, and we may benefit more from having variable conditions which we could define at source level, that could be done if we had an evaluation function that could return boolean values, however for the purpose of this calculation this fixed condition will do.

More importantly, the semantics of *Ite* are compositional, again because we have defined it's semantics in terms of the semantics of it's arguments so calculations about *Ite* expressions will be *inductive.*

step 2:

> "Define equations that specify the correctness of the compiler".

The *exec* and *comp* functions still take the same type of arguments as before, so there is no need to update the specifications yet

$$\begin{aligned} \mathit{exec}\ (\mathit{comp}\ x)\ s &= \mathit{eval}\ x : s & (17) \\ \mathit{exec}\ (\mathit{comp}'\ x\ c)\ s &= \mathit{exec}\ c\ (\mathit{eval}\ x : s) & (18) \end{aligned}$$

[5] "if", "then" and "else" in this case

3.1 Calculation

Step 3: "Calculate definitions that satisfy these specifications"

In order to satisfy the specification (18), we begin with it's LHS where x is our *Ite* expression.

$$\begin{aligned}
& \mathit{exec}\ (\mathit{comp}'\ (\mathit{Ite}\ x\ y\ z)\ c)\ s \\
=\ & \{\text{specification of compiler}\} \\
& \mathit{exec}\ c\ (\mathit{eval}\ (\mathit{Ite}\ x\ y\ z) : s) \\
=\ & \{\text{defintion of } \mathit{eval}\} \\
& \mathit{exec}\ c\ (\mathit{if}\ \ \mathit{eval}\ x \neq 0\ \mathit{then}\ \mathit{eval}\ y\ \mathit{else}\ \mathit{eval}\ z : s)
\end{aligned}$$

There are no more definitions to apply from here, it's clear that we required to create a new definition for *exec*, and because this is an inductive calculation we can use the inductive hypotheses just like with Bahr and Hutton's calculation of *Add*. The inductive hypotheses are:

$$\begin{aligned}
\mathit{exec}\ (\mathit{comp}'\ x\ c)\ s &\ =\ \mathit{exec}\ c\ (\mathit{eval}\ x : s) \\
\mathit{exec}\ (\mathit{comp}'\ y\ c)\ s &\ =\ \mathit{exec}\ c\ (\mathit{eval}\ y : s) \\
\mathit{exec}\ (\mathit{comp}'\ z\ c)\ s &\ =\ \mathit{exec}\ c\ (\mathit{eval}\ z : s)
\end{aligned}$$

However, to be able to use them, the stack must have $\mathit{eval}\ x, y, z$ on top in some order, a new definition of *exec* is needed to solve the generalised equation:

$$\mathit{exec}\ c'\ (k : m : n : s) = \mathit{exec}\ c\ (\mathit{if}\ \ k \neq 0\ \mathit{then}\ m\ \mathit{else}\ n : s)$$

Our code constructor to solve this will be

$$\mathit{ITE}\ :: \mathit{Code}\ \rightarrow \mathit{Code}$$

and it's definition for the virtual machine

$$\mathit{exec}\ (\mathit{ITE}\ c)\ (k : m : n : s) = \mathit{exec}\ c\ (\mathit{if}\ \ k \neq 0\ \mathit{then}\ m\ \mathit{else}\ n : s)$$

i.e executing and ITE instruction checks the top of the stack for the condition $k \neq 0$ and if so, then k and n are removed, else k and m are removed. Using this to continue the calculation, we have

$$\begin{aligned}
& \mathit{exec}\ c\ (\mathit{if}\ \ \mathit{eval}\ x\ \ \neq 0\ \mathit{then}\ \mathit{eval}\ y\ \mathit{else}\ \mathit{eval}\ z : s) \\
=\ & \{\text{defintion of } \mathit{exec}\} \\
& \mathit{exec}\ (\mathit{ITE}\ c)\ (\mathit{eval}\ x : \mathit{eval}\ y : \mathit{eval}\ z : s) \\
=\ & \{\text{induction hypothesis } for\ x\} \\
& \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{ITE}\ c))\ (\mathit{eval}\ y : \mathit{eval}\ z : s) \\
=\ & \{\text{induction hypothesis } for\ y\} \\
& \mathit{exec}\ (\mathit{comp}'\ y\ (\mathit{comp}'\ x\ (\mathit{ITE}\ c)))\ (\mathit{eval}\ z : s) \\
=\ & \{\text{induction hypothesis } for\ z\} \\
& \mathit{exec}\ (\mathit{comp}'\ z\ (\mathit{comp}'\ y\ (\mathit{comp}'\ x\ (\mathit{ITE}\ c))))\ s
\end{aligned}$$

We may conclude from this calculation these two new definitions for the compiler and virtual machine:

$$\begin{aligned} \mathit{comp}'\ (\mathit{Ite}\ x\ y\ z)\ c &= \mathit{comp}'\ z\ (\mathit{comp}'\ y\ (\mathit{comp}'\ x\ (\mathit{ITE}\ c))) && (19) \\ \mathit{exec}\ (\mathit{ITE}\ c)\ (k : m : n : s) &= \mathit{exec}\ c\ ((\mathit{if}\ \ k \neq 0\ \mathit{then}\ m\ \mathit{else}\ n) : s) && (20) \end{aligned}$$

Both of these definitions strike a glaring resemblance to the compiler and virtual machine counterparts for an *Add* expression; the *comp'* is almost identical, and the *exec* is just an operation using some Haskell code “if k then m else n“ as opposed to “m + n”. Clearly the nature of *Add* and *Ite* are more similar than immediately meets the eye.

In this language addition is a prefix operator which is followed by two arguments. Both of these arguments are compiled and pushed to the stack in some order which is up to choice. Likewise, *Ite*is a prefix operator also followed by arguments which end up on the stack in some order of choosing, making the only real difference that there are three of them.

In both expressions an operation is applied involving all of the arguments involved on the stack. It might be worth investigating that for any n argument operation of a similar nature, that the compile rule is also similar and the execution rule only differs by the Haskell code to perform the operation.

3.2 Testing

For the testing, the example *Ite* expression β will be used, where:

$$\beta = \mathit{Ite}\ (\mathit{Val}\ 1)(\mathit{Add}\ (\mathit{Val}\ 2)(\mathit{Val}\ 3))(\mathit{Add}\ (\mathit{Val}\ 4)(\mathit{Val}\ 5))))$$

This expression would test that each of the sub-expressions (*Add* and *Val*) compile properly first, and that condition will correctly choose what sub expression's code to execute. For completeness we should need to test both True and False outcomes of the condition, and such tests are included in the supporting Haskell files for L_c, however the by hand calculations for these tests are omitted for brevity because they would not add much new information to these series of tests.

3.2.1 Compilation

These equations demonstrate what code is generated when the definitions of the compile functions are applied to β:

$$
\begin{aligned}
& \mathit{comp}(\mathit{Ite}\ (\mathit{Val}\ 1)(\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))(\mathit{Add}\ (\mathit{Val}\ 4)\ (\mathit{Val}\ 5)))\\
= & \{\text{definition of } \mathit{comp}\ e\}\\
& \mathit{comp}'\ (\mathit{Add}\ (\mathit{Val}\ 4)\ (\mathit{Val}\ 5))\\
& (\mathit{comp}'\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))\\
& (\mathit{comp}'\ (\mathit{Val}\ 1)\ (\mathit{ITE}\ \mathit{HALT})))\\
= & \{\text{definitions } \textit{comp'Add} \text{ twice, and } \mathit{comp}\ \ \mathit{Val} \text{ once}\}\\
& \mathit{comp}'\ (\mathit{Val}\ 4)\ (\mathit{comp}'\ (\mathit{Val}\ 5)\\
& (\mathit{ADD}\ (\mathit{comp}'\ (\mathit{Val}\ 2)\ (\mathit{comp}'\ (\mathit{Val}\ 3)\\
& (\mathit{ADD}\ (\mathit{PUSH}\ 1\ (\mathit{ITE}\ \mathit{HALT})))))))\\
= & \{\text{definition of } \textit{comp' Val}\text{, 4 times}\}\\
& \mathit{PUSH}\ 4\ (\mathit{PUSH}\ 5\ (\mathit{ADD}\ (\mathit{PUSH}\ 2\ (\mathit{PUSH}\ 3\ (\mathit{ADD}\ (\mathit{PUSH}\ 1\ (\mathit{ITE}\ \mathit{HALT})))))))
\end{aligned}
$$

When the expression β is compiled using the Haskell implementation of the compiler GHC: *comp* β =

$$
\begin{aligned}
& \mathit{comp}\ (\mathit{Ite}\ (\mathit{Val}\ 1)(\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))(\mathit{Add}\ (\mathit{Val}\ 4)\ (\mathit{Val}\ 5)))\\
= & \{\text{Haskell implementation of } \mathit{comp} \text{ and } \textit{comp'}\}\\
& \mathit{PUSH}\ 4\ (\mathit{PUSH}\ 5\ (\mathit{ADD}\ (\mathit{PUSH}\ 2\ (\mathit{PUSH}\ 3\ (\mathit{ADD}\ (\mathit{PUSH}\ 1\ (\mathit{ITE}\ \mathit{HALT})))))))
\end{aligned}
$$

3.2.2 Execution

Here are definitions of the virtual machineapplied to the compiled code for the expression β:

$$
\begin{aligned}
&\quad \mathit{exec}\ (\mathit{PUSH}\ \ 4\,(\mathit{PUSH}\ \ 5\,(\mathit{ADD}\ ,(\mathit{PUSH}\ \ 2\,(\mathit{PUSH}\ \ 3\,(\mathit{ADD}\ ,(\mathit{PUSH}\ \ 1\,(\mathit{ITE}\ \ \mathit{HALT}))))))))\ \ [\,]\\
&= \{\text{definition } \mathit{exec}\ \mathit{PUSH}\ \ \text{twice}\}\\
&\quad \mathit{exec}\ (\mathit{ADD}\ (\mathit{PUSH}\ 2\ (\mathit{PUSH}\ 3\ (\mathit{ADD}\ (\mathit{PUSH}\ 1\ (\mathit{ITE}\ \mathit{HALT}))))))\ \ [5,4]\\
&= \{\text{definition } \mathit{exec}\ \mathit{ADD}\ \}\\
&\quad \mathit{exec}\ (\mathit{PUSH}\ 2\ (\mathit{PUSH}\ 3\ (\mathit{ADD}\ (\mathit{PUSH}\ 1\ (\mathit{ITE}\ \mathit{HALT})))))\ \ [9]\\
&= \{\text{definition } \mathit{exec}\ \mathit{PUSH}\ \ \text{twice}\}\\
&\quad \mathit{exec}\ (\mathit{ADD}\ (\mathit{PUSH}\ 1\ (\mathit{ITE}\ \mathit{HALT})))\ \ [3,2,9]\\
&= \{\text{definition } \mathit{exec}\ \mathit{ADD}\ \}\\
&\quad \mathit{exec}\ (\mathit{PUSH}\ 1\ (\mathit{ITE}\ \mathit{HALT}))\ \ [5,9]\\
&= \{\text{definition } \mathit{exec}\ \mathit{PUSH}\ \}\\
&\quad \mathit{exec}\ (\mathit{ITE}\ \mathit{HALT})\ \ [1,5,9]\\
&= \{\text{definition } \mathit{exec}\ \mathit{ITE}\ \}\\
&\quad \mathit{exec}\ \mathit{HALT}\ \ [5]\\
&= \{\text{definition } \mathit{exec}\ \mathit{HALT}\ \}\\
&\quad [5]
\end{aligned}
$$

Which agrees with compiled code for β being executed using the Haskell implementation of the virtual machine. GHC:

$$
\begin{aligned}
&\quad \mathit{exec}\ (\mathit{PUSH}\ 0\ (\mathit{PUSH}\ 1\ (\mathit{ADD}\ (\mathit{PUSH}\ 2\ (\mathit{ADD}\ \mathit{HALT})))))\ [\,]\\
&= \{\text{Haskell implementation of } \mathit{exec}\}\\
&\quad [3]
\end{aligned}
$$

It is apparent that the compilation rule produces a lot of code to execute which is then thrown away; all the effort (or execution time) in executing $(\mathit{PUSH}\ \ 4\,(\mathit{PUSH}\ \ 5\,(\mathit{ADD}\ c)))$, at the start was eventually wasted because when it came to executing the ITE instruction, the result of this part of the code (9), was thrown away without using it in any other way, which means we didn't need this code to be compiled in the first place. Clearly there is room for improvement in efficiency of the compiler.

3.2.3 Interpretation

Applying the definitions of the evaluation function (the interpreter) to the expression β:

$$\begin{aligned}
& \mathit{eval}\ (\mathit{Ite}\ (\mathit{Val}\ 1)\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))\ (\mathit{Add}\ (\mathit{Val}\ 4)\ (\mathit{Val}\ 5))) \\
= & \{\text{definition of } \mathit{eval}\ \mathit{Add}\} \\
& \mathit{eval}\ (\mathit{Add}\ (\mathit{Val}\ 0)\ (\mathit{Val}\ 1))\ +\ \mathit{eval}\ (\mathit{Val}\ 2) \\
= & \{\text{definition of } \mathit{eval}\ \mathit{Add} \text{ and } \mathit{eval}\ \mathit{Val}\} \\
& \mathit{eval}\ (\mathit{Val}\ 0)\ +\ \mathit{eval}\ (\mathit{Val}\ 1)\ +\ 2 \\
= & \{\text{definition of } \mathit{eval}\ \mathit{Val} \text{ twice}\} \\
& 0\ +\ 1\ +\ 2 \\
= & \{\text{arithmetic}\} \\
& 3
\end{aligned}$$

Interpreting β using the Haskell implementation of the the interpreter GHC: $\mathit{eval}\ \beta =$

$$\begin{aligned}
& \mathit{eval}\ (\mathit{Ite}\ (\mathit{Val}\ 1)\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))\ (\mathit{Add}\ (\mathit{Val}\ 4)\ (\mathit{Val}\ 5))) \\
= & \{\text{Haskell implementation of } \mathit{eval}\} \\
& 3
\end{aligned}$$

The tests by hand and with the implemented evaluation function have passed, at least with β. Comparing the results of implemented interpreter and virtual machine on β GHC: [3], 3.

3.3 Summary

The Arithmetic language L_a has now been extended slightly to include a conditional operator, this new language will be referred to as L_c.

The results of testing the compiler, virtual machine and interpreter on an example conditional expression all work out correctly. Moreover the virtual machine and and interpret agree, and so have been shown to satisfy the equation for β:

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

Where c is just HALT , and s is empty [ ].

In summary, from this series of calculations we saw more arguments being pushed to the stack and learnt that there is a lot of choice in the order in which we compile arguments with the Bahr and Hutton method. For instance the first argument k of the *Ite* did not necessarily need to be on top of the other two arguments in the stack, because the *exec* could have been defined as

$$\mathit{exec}\ (\mathit{ITE}\ c)\ (n : m : k : s) = \mathit{exec}\ c\ (\mathit{if}\ \ k \neq 0\ \mathit{then}\ m\ \mathit{else}\ n : s)$$

where *comp'*would similarly change to

$$comp'\ (Ite\ x\ y\ z)\ c = comp'\ x\ (comp'\ y\ (comp'\ z\ (ITE\ c)))$$

and thanks to Haskell's pattern matching, it would work all the same. When it came to testing we also saw that the method of compiling a conditional produced more code than was necessary which resulted in a lot of wasted execution time. This is because the semantics of an *Ite* were taken literally when it came to the calculation, whereby all arguments are evaluated and therefore also compiled and executed.

This kind of evaluation of the condition is eager; in that both cases are compiled and executed while only one piece of code needs to be. This can avoided at if we instead separate the code for $comp'\ y$ and $comp'\ z$ into two separate branches and throw away the branch for the case that we don't need to execute, which is determined by the condition.

3.4 Lazy evaluation

First we define the semantics in a compositional manner. The "Lazy if then else" function will be called *Lite*, and it's semantics are:

$$\begin{aligned} \textbf{data}\ Expr &= ...\mid Lite\ Expr\ Expr\ Expr \\ eval\ (Lite\ x\ y\ z) &= if\ \ eval\ x \neq 0 then\ \ eval\ y\ \ else\ \ eval\ z \end{aligned} \tag{21}$$

Expressions cannot be lazily evaluated, because lazy evaluation means evaluating something only when the evaluation function is called on it, therefore being able to lazily evaluate something by calling the evaluator on it, would be contradictory. Thus *Lite* has the same semantics as *Ite*.

3.4.1 Calculation

Step 2:

> "Define equations that specify the correctness of the compiler"

The specification for the compiler has not changed, but here they are for reference

$$exec\ (comp\ x)\ s \ = \ eval\ x : s \tag{22}$$

$$exec\ (comp'\ x\ c)\ s \ = \ exec\ c\ (eval\ x : s) \tag{23}$$

Beginning by applying the specification and evaluation function to our new source expression:

$$\begin{aligned} & exec\ (comp'\ (Lite\ \ x\ y\ z)\ c)\ s \\ = & \{\text{specification of the compiler}\} \\ & exec\ c\ (eval\ (Lite\ x\ y\ z) : s) \\ = & \{\text{defintion of } eval\} \\ & exec\ c\ (if\ \ eval\ x\ \ \neq 0\ then\ \ eval\ y\ \ else\ \ eval\ z : s) \end{aligned}$$

Like with *Ite* our calculation halts here, but our aim with this time, is that rather than deciding upon what value throw away we instead decide upon two branches of code, *ct* and *ce*, containing complied code of the *y* and *z* expressions.

$$\begin{aligned} LITE\ \ &::\ \ Code\ \to Code\ \to Code \\ exec\ (LITE\ ct\ ce)\ (eval\ x : s)\ \ &=\ \ exec\ c\ (if\ \ eval\ x\ \ \neq 0\ then\ \ eval\ y\ \ else\ \ eval\ z : s) \end{aligned} \tag{24}$$

We cannot use this equation as a definition of exec because *c*, *y* and *z* are unbound in the body of the expression[4, page 10]. However we can bind them in *ct* and *ce*. But what is contained within each of these arguments?

In order to be type correct both *ct* and *ce* must be of type code, code is produced by the compile function which is defined such that any expression is followed by continuation code, therefore both *ct* and *ce* also contain the continuation code *c* for expressions *y* and *z*.

$$\begin{aligned} ct &= comp'\ y\ c \\ ce &= comp'\ z\ c \end{aligned}$$

In summary our generalised formal partial specification is

$$exec\ (LITE\ (comp'\ y\ c)\ (comp'\ z\ c))\ (k : s) = exec\ (if\ \ k \neq 0\ then\ comp'\ y\ c\ else\ comp'\ z\ c)\ s$$

which makes the rest of our calculation straightforward

$$\begin{aligned} & exec\ c\ (if\ \ eval\ x\ \ \neq 0\ then\ eval\ y\ else\ eval\ z : s) \\ = & \{\text{defintion of } exec\} \\ & exec\ (LITE\ (comp'\ y\ c)\ (comp'\ z\ c))\ (eval\ x : s) \\ = & \{\text{induction hypothesis for } x\} \\ & exec\ (comp'\ x\ ((LITE\ (comp'\ y\ c)\ (comp'\ z\ c)))\ s \end{aligned}$$

From which we may deduce

$$comp'\ (Lite\ x\ y\ z)\ c = comp'\ x\ (LITE\ (comp'\ y\ c)\ (comp'\ z\ c))$$

that is compiling a *Lite* means compiling the condition expression *x* followed by the *LITE*constructor and two branches of code each containing the same continuation code. This method poses a problem; it makes double use of the continuation code, this not only doubles in length of the fully compiled code, but doubles in *compile time.* This causes an exponential compile time complexity $\mathcal{O}(2^n)$ and equally space complexity $\mathcal{O}(2^n)$ where n is the number of *Lite* expressions in the source code. Surely there must be a way to avoid this.

The problem comes from the double use of *c*, at the moment this is necessary because *comp'* takes an *Expr* and *Code* as arguments and is in each branch of code, however they could instead *share* a code continuation if we used a different compile function which would allow a single expression (and all sub-expressions contained within) to be compiled without continuation code of it's own, unlike *comp'*, also LITE would need 3 code arguments in it's constructor and we would need a way of reuniting the condition's code back with the rest of the code *c* as "$:\ \ cons''$ would not work,

$$\begin{aligned} f\ (Lite\ \ x\ y\ z)\ c &= f'\ x\ (LITE\ \ (f\ \ y)\ (f\ \ z)\ c) \\ exec\ (LITE\ \ (f\ \ y)\ (f\ \ z)\ c)(k : s) &= exec\ ((if \neq 0\ then\ (f\ \ y)\ else\ (f\ \ z)) : c)\ s \end{aligned}$$

but this is an optimisation problem out of the scope of this dissertation.

In summary, we have discovered the following definitions for the compiler and virtual machine.

$$\begin{aligned} comp'\ (Lite\ x\ y\ z)\ c &= comp'\ x\ (LITE\ (comp'\ y\ c)\ (comp'\ z\ c)) && (25)\\ exec\ (LITE\ ct\ ce)\ (eval\ x : s) &= exec\ c\ (if\ \ eval\ x\ \ \neq 0\ then\ eval\ y\ else\ \ eval\ z : s) && (26)\end{aligned}$$

3.4.2 Testing

This series of tests looks to not only validate the functionality of these definitions, but also determine if the argument expressions are actually lazily evaluated and produce more efficient code than with the eager *Ite* function.

A fair test would be to apply *Lite* to the same arguments as *Ite* was applied to in §3.2. Our test expression δ is

$$\delta = Lite\ (Val\ 1)\ (Add\ (Val\ 2)\ (Val\ 3))\ (Add\ (Val\ 4)\ (Val\ 5))$$

3.4.3 Compilation

Applying compile functions to δ should yield code that is significantly longer as discussed previously:

$$\begin{aligned} & comp\ (Lite\ (Val\ 1)\ (Add\ (Val\ 2)\ (Val\ 3))\ (Add\ (Val\ 4)(Val\ 5)))\\ = & \{\text{defintion of } comp\ e\}\\ & comp'\ (Lite\ (Val\ 1)\ (Add\ (Val\ 2)\ (Val\ 3))\ (Add\ (Val\ 4)(Val\ 5)))\ HALT\\ = & \{\text{defintion of } comp\text{'}\ Lite\}\\ & comp'\ (Val\ 1)(LITE\ (comp'\ (Add\ (Val\ 2)\ (Val\ 3))\ HALT)\\ & (comp'\ (Add\ (Val\ 4)\ (Val\ 5))\ HALT))\\ = & \{\text{definition of } comp\text{'}\ Val \text{ and } comp\text{'}\ Add \text{ twice}\}\\ & PUSH\ 1\ (LITE\ (comp'\ (Val\ 2)\ (comp'\ (Val\ 3)\ (ADD\ HALT))\\ & (comp'\ (Val\ 4)\ (comp'\ (Val\ 5)\ (ADD\ HALT))))\\ = & \{\text{definition of } comp\text{'}\ Val \text{ four times}\}\\ & PUSH\ 1\ (LITE\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ HALT)))\\ & (PUSH\ 4\ (PUSH\ 5(ADD\ HALT))))\end{aligned}$$

Using the Haskell implementation of the compile function on δ yields the same result $comp\ \delta =$:

$$\begin{aligned} & comp\ (Lite\ (Val\ 1)\ (Add\ (Val\ 2)\ (Val\ 3))\ (Add\ (Val\ 4)\ (Val\ 5))))\\ = & \{\text{Haskell implementation of } comp \text{ and } comp\text{'}\}\\ & PUSH\ 1\ (LITE\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ HALT)))\\ & (PUSH\ 4\ (PUSH\ 5(ADD\ HALT))))\end{aligned}$$

3.4.4 Execution

$$
\begin{aligned}
& \mathit{exec}\ (\mathit{PUSH}\ 1\ (\mathit{LITE}\ (\mathit{PUSH}\ 2\ (\mathit{PUSH}\ 3\ (\mathit{ADD}\ \mathit{HALT})))(\mathit{PUSH}\ 4\ (\mathit{PUSH}\ 5(\mathit{ADD}\ \mathit{HALT}))))) \ [\,] \\
& = \{\mathit{equation}\ 14\} \\
& \mathit{exec}\ (\mathit{LITE}\ (\mathit{PUSH}\ 2\ (\mathit{PUSH}\ 3\ (\mathit{ADD}\ \mathit{HALT})))(\mathit{PUSH}\ 4\ (\mathit{PUSH}\ 5(\mathit{ADD}\ \mathit{HALT})))) \ [1] \\
& = \{\mathit{equation}\ 26\} \\
& \mathit{exec}\ (\mathit{PUSH}\ 2\ (\mathit{PUSH}\ 3\ (\mathit{ADD}\ \mathit{HALT})))\ [\,] \\
& = \{\mathit{equation}\ 14\ \mathit{twice}\} \\
& \mathit{exec}\ (\mathit{ADD}\ \mathit{HALT})\ [3,2] \\
& = \{\mathit{equation}\ 15\} \\
& \mathit{exec}\ \mathit{HALT}[5] \\
= & \{\mathit{equation}\ 13\} \\
& [5]
\end{aligned}
$$

Using the Haskell implementation of the virtual machine.

$$
\begin{aligned}
& \mathit{exec}\ (\mathit{PUSH}\ 1\ (\mathit{LITE}\ (\mathit{PUSH}\ 2\ (\mathit{PUSH}\ 3\ (\mathit{ADD}\ \mathit{HALT})))(\mathit{PUSH}\ 4\ (\mathit{PUSH}\ 5(\mathit{ADD}\ \mathit{HALT}))))) \\
& = [5]
\end{aligned}
$$

3.4.5 Interpretation

$$
\begin{aligned}
& \mathit{eval}\ (\mathit{Lite}\ (\mathit{Val}\ 1)(\mathit{Add}\ (\mathit{Val}\ 2)(\mathit{Val}\ 3))(\mathit{Add}\ (\mathit{Val}\ 4)(\mathit{Val}\ 5))) \\
= & \{\mathit{equation}\ 21\}
\end{aligned}
$$

In the Haskell implementation of the interpreter:

$$\mathit{eval}\ (\mathit{Lite}\ (\mathit{Val}\ 1)(\mathit{Add}\ (\mathit{Val}\ 2)(\mathit{Val}\ 3))(\mathit{Add}\ (\mathit{Val}\ 4)(\mathit{Val}\ 5))) = 5$$

Which agrees with the execution of the compiled code of the expression.

3.4.6 Proof

Finally we come to proving our calculation. We aim to show that our definitions for *Lite*satisfy the specification

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

$$
\begin{aligned}
& \quad \mathit{exec}\ c\ (\mathit{eval}\ (\mathit{Lite}\ x\ y\ z) : s) \\
& = \{\text{defintion of } \mathit{eval}\} \\
& \quad \mathit{exec}\ c\ ((\mathit{if}\ \ \mathit{eval}\ x\ \ \neq 0\ \mathit{then}\ \mathit{eval}\ y\ \mathit{else}\ \mathit{eval}\ z) : s) \\
& = \{\mathit{define}\ \mathit{exec}\ (\mathit{LITE}\ (\mathit{comp}'\ x\ c)(\mathit{comp}'\ y\ c)(k : s) \\
& \quad\ = \mathit{exec}\ (\mathit{if}\ \ k \neq 0\ \mathit{then}\ (\mathit{comp}'\ y\ c)\ \mathit{else}\ (\mathit{comp}'\ z\ c))\ s\} \\
& \quad \mathit{exec}\ (\mathit{LITE}\ (\mathit{comp}'\ y\ c)\ (\mathit{comp}'\ z\ c))\ (\mathit{eval}\ x : s) \\
& = \{\text{defintion of } \mathit{comp}'\} \\
& \quad \mathit{exec}\ (\mathit{comp}'\ x\ (\mathit{LITE}\ (\mathit{comp}'\ y\ c)\ (\mathit{comp}'\ z\ c)))\ s \\
& = \{\text{defintion of } \mathit{comp}'\} \\
& \quad\ = \mathit{exec}(\mathit{comp}'\ (\mathit{Lite}\ x\ y\ z)c)\ s
\end{aligned}
$$

QED

3.5 Summary

In conclusion we have calculated the following definitions [4, page 11]:

$$
\begin{array}{rcl}
\textbf{data}\ \mathit{Code} & = & ...\mathit{ITE}\ \ \mathit{Code}\ |\mathit{LITE}\ \ \mathit{Code}\ \ \mathit{Code} \\
\mathit{comp} & :: & \mathit{Expr}\ \to \mathit{Code} \\
\mathit{comp}\ x & = & \mathit{comp}'\ x\ \mathit{HALT} \\
\mathit{comp}' & :: & \mathit{Expr}\ \to \mathit{Code}\ \to \mathit{Code} \\
\mathit{comp}'\ (\mathit{Ite}\ x\ y\ z) & = & \mathit{comp}'\ z\ (\mathit{comp}'\ y\ (\mathit{comp}'\ x\ (\mathit{ITE}\ c))) \\
\mathit{comp}'\ (\mathit{Lite}\ x\ y\ z) & = & \mathit{comp}'\ x\ (\mathit{LITE}\ (\mathit{comp}'\ y\ c)\ (\mathit{comp}'\ z\ c)) \\
\mathit{exec}\ (\mathit{ITE}\ c)\ (k : m : n : s) & = & \mathit{exec}\ c\ ((\mathit{if}\ \ k \neq 0\ \mathit{then}\ m\ \mathit{else}\ n) : s) \\
\mathit{exec}\ (\mathit{LITE}\ \mathit{ct}\ \mathit{ce})\ (k : s) & = & \mathit{exec}\ (\mathit{if}\ \ k \neq 0\ \mathit{then}\ \mathit{ct}\ \mathit{else}\ \mathit{ce})\ s
\end{array}
$$

We have seen that via induction on the arguments of the virtual machine we can not only manipulate stack elements but also code. But our language is still very basic. Could we introduce more structures to make the language more complicated; with more features that resemble an actual programming language? Bahr and Hutton certainly do, by using multiple code continuations they implement exception handling and the “compilation techniques arising naturally” through calculations[4, page 24].

4 Bindings, L_b

Variables are a key component of a lot of programming languages; they allow users to easily reference an object without needing to recompute. Computers use memory to store information which programs and programmers a like may take advantage of. Variables may be declared by *binding* a pair of two pieces of information: a name, and a value. Our *eval* function as of yet cannot do such an operation because it does not manipulate any kind of data structure of it's own, it only iterates through expressions and interprets them. *eval* would require atleast one more argument containing a set of bindings which it can manipulate.

4.1 Semantics

step 1: define an evaluation function in a compositional manner.

Our bindings structure will be called an environment, it is a stack of name-value pairs $(i,\ j)$ where a string i paired to an integer j.

$$\textbf{type}\, Env = [(String,\ Int)]$$

The evaluation function needs to be updated to take an *Env* as an argument as well as an expression.

$$\begin{array}{cl}
eval & ::\ Expr \rightarrow Env \rightarrow Int \\
eval\ (Val\ n)\ bs & = n \\
eval\ (Add\ x\ y)\ bs & = eval\ x\ bs\ +\ eval\ y\ bs \\
eval\ (Ite\ x\ y\ z)\ bs & = if\ (eval\ x\ bs) \neq 0\ then\ (eval\ y\ bs)\ else\ (eval\ z\ bs) \\
eval\ (Lite\ x\ y\ z)\ bs & = if\ (eval\ x\ bs) \neq 0\ then\ (eval\ y\ bs)\ else\ (eval\ z\ bs)
\end{array}$$

All of our functions have been calculated without need of environments, therefore we can be reasonably sure that simply adding in the *Env*argument won't affect them[6].

Now the expression that will make a binding, we'll call "*Let*". *Let* has the concrete syntax: $Let\ v = x\ in\ y$, again without a parser to do lexical analysis, we need to use abstract syntax, and our constructor for it

$$\textbf{data}\ Expr = ...|\ Let\ String\ \ Expr\ Expr$$

Let creates a new binding, by pushing the String-Int pair onto the *Env*, to reference a variable we will use a "Var" constructor

$$\textbf{data}\ Expr = ...|\ Var\ String$$

The String is taken directly from the source String part of the *Let* expression, however the value it's paired to needs to be computed inductively.

[6] brackets have been added around the expressions for ease of reading

Therefore our semantics of Let and Var

$$\begin{aligned}
eval\ (Let\ v\ x\ y)\ bs &= eval\ y\ ((v,\ eval\ x\ bs) : bs)\\
eval\ (Var\ v)\ bs &= valueOf\ v\ bs\\
valueOf &:: String \rightarrow Env \rightarrow Int\\
valueOf\ s\ [\,] &= error\ \text{“}Binding\ out\ of\ scope?\text{”}\\
valueOf\ s\ ((v,\ n) : bs) &= if\ s == v\ then\ n\ else\ valueOf\ s\ bs
\end{aligned}$$

valueOf is an auxiliary function, it takes a string as input, iterates through an environment, and attempts to match the string to the strings in each binding. It returns the value of the *first*[7] binding to have a matching string.

Sub-expressions inherit environments from their parent expressions, and therefore the variables within them have the same *scope*, except in the case of *Let* where each sub-expression x and y has a different scope. To illustrate this, the following equations have the resulting environment included on the RHS. Our evaluator actually empties the environment after it's computation, but it's helpful to think of it like this

$$eval\ (Add\ (Var\ \text{“}a'')\ (Val\ 2))\ [(\text{“}a'',\ 2)] \qquad = 4,\ [\text{“}a'',\ 2]$$

$$eval\ (Let\ \text{“}a''\ (Val\ 2)\ (Add\ (Var\ a)\ (Val\ 2)))\ [\,] \qquad = 4,\ [(\text{“}a'',\ 2)]$$

$$\begin{aligned}
&eval\ (Let\text{“}b''\\
&(Let\ \text{“}a''\ (Val\ 2)\ (Add\ (Var\ a)\ (Val\ 2)))\\
&(Add\ (Var\ b)\ (Val\ 2)))\ [\,] \qquad = 6,\ [(\text{“}b'',\ 4),\ (\text{“}a'',\ 2)]
\end{aligned} \tag{27}$$

$$\begin{aligned}
&eval\ (Let\text{“}a''\\
&(Let\ \text{“}b''\ (Val\ 2)\ (Add\ (Var\ a)\ (Val\ 2)))\\
&(Add\ (Var\ b)\ (Val\ 2)))\ [\,] \qquad = \text{“}Binding\ b\ out\ of\ scope''
\end{aligned} \tag{28}$$

In equation (27) the second *Add* inherits the scope of *Let* sub-expression preceding it, because it evaluates *within scope* of it. Conversely with equation (28) the first sub-expression tries to reference a variable that is *out of scope*, and our interpreter throws an error.

4.2 Compiler Correctness

Step 2: Define equations that specify the correctness of the compiler.

Our compiler specifications have been:

[7] Should a variable name be bound to twice in a source expression and in the same scope, only the latter binding will be in effect

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

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

However, these no longer hold because our eval function has changed with the introduction of environments, therefore we need to update these equations. Our *Env* specifies parings of variable names to values, the compiler may be able to handle names, but cannot compute any values on its own but it can produce code that will produce the same effect once executed. Breaking down what the interpreter does can indicate what the compiler and virtual machine should do.

$$eval\ (Let\ v\ x\ y)\ bs = eval\ y\ ((v,\ eval\ x\ bs) : bs)$$

NB: bs, although a new type it is just a list of pairs, we could re-write it in equations as [(vars, vals)] where vars are the variable names and vals, their values, but it is simpler to keep it as bs.

To evaluate a *Let*, the interpreter must do three things:

1. Evaluate x in the current environment

2. Bind the variable v to that value

3. Evaluate y in the modified environment

Clearly the compiler and virtual machine must have some kind of environment of their own to reflect changes in the environment. The compiler cannot compute the value parts of each pair, it can however, store what variables are called and in what order. The structure that will do this, will be a stack of strings called a "*Context* " or "*Cxt* ".

$$\begin{aligned}
\textbf{type}\ Context &= [String] \\
comp &:: Expr \to Code \\
comp\ e &= comp'\ e\ [\,]\ HALT(*) \\
comp' &:: Expr \to Cxt \to Code
\end{aligned}$$

**comp* stays much the same except it's cxt is initially empty.

Remember, our aim at the moment is to relate the compiler to the semantics via a virtual machine. If we tried to do update our compiler specifications now; without the proper definitions for the virtual machine, we'd have

$$\begin{aligned}
exec\ (comp\ x)\ s &= exec\ (comp'\ x\ [\,]\ c)\ s \\
exec\ (comp'\ x\ cxt\ c)\ s &= exec\ c\ ((eval\ x\ bs) : bs) : s
\end{aligned}$$

Which cannot be used because bs is still unbound. Bindings are just name-value pairs, at the moment we have variable names that don't have paired values, because the compiler cannot compute values but this can be left to the virtual machine.

We can use[8] a new stack to manipulate these variable values. *exec*should take as input the a pair of: it's current run-time stack, and the values stack, and then output the modified versions of both, that is

$$\begin{aligned} &\textbf{type}\ Memory && = (Stack,\ Stack) && (29)\\ &exec && ::\ Code \rightarrow Memory \rightarrow Memory &&\\ &exec\ (comp'\ x\ cxt\ c)(s,\ vs) && = exec\ c\ ((eval\ x\ bs) : s,\ vs) && (30) \end{aligned}$$

We now have a way of storing variable names and their values, just in two different places. To update the compiler correctness equations, we need a function to pair up the names to values. That is the "Zip" function in Haskell

$$exec\ (comp\ e)\ (s,\ vs) = exec\ (comp'\ e\ [\,]\ HALT)\ (s,\ vs)$$

$$exec\ (comp'\ e\ cxt\ c)\ (s,\ vs) = exec\ c\ ((eval\ e\ (Zip\ cxt\ vs) : s,\ vs)$$

Because our equations for
evalf use the *bs* symbol for environments, it will be useful to formally state:

$$\begin{aligned} &Zip\ (x : xs)\ (y : ys) && = (x,\ y) : (Zip\ xs\ ys) && (31)\\ &Zip\ cxt\ vs && = bs && (32) \end{aligned}$$

These equations satisfy the full description of step 2 in their Bahr and Hutton's General methodology [4, page 42].

4.3 Calculation

Step 3: Calculate definitions that satisfy the correctness of the compiler

Now that we have our compiler equations, we can calculate definitions that satisfy them by constructive rule induction starting from the LHS of 18[4, pg 42].

$$\begin{aligned} & exec\ (comp'\ (Let\ v\ x\ y)\ cxt\ c)\ (s,\ vs)\\ = & \{specification\ 30\}\\ & exec\ c\ (eval\ (Let\ v\ x\ y)\ (Zip\ cxt\ vs) : s,\ vs)\\ = & \{\text{defintion of } Zip,\ \text{defintion of } eval\ \}\\ & exec\ c\ (eval\ y\ (\ (v,\ eval\ x\ bs) : bs) : s,\ vs) \end{aligned}$$

There are no more definitions to apply.

[8] There may be a way to have the variable values on the run-time stack, but it's simpler to use a new one

We aim to apply the inductive hypotheses for x and y, however our original ones will not do because they won't tell us anything about changes of context. We know, by the definition of our interpreter, that x needs to be evaluated first and bound to v in the environment, so it can be referenced by any subexpression in y. The Zip function connects the environment to our context and values stacks. To update an environment we can use the definition of zip (32), where x and y are the new v and χ.

$$\mathit{Zip}\ (v : xs)\ (\chi : ys) = (v,\ \chi) : (\mathit{Zip}\ xs\ ys) = (v,\ \chi) : bs$$

y is evaluated with this environment. Making the new inductive hypothesis for x and y

$$\begin{aligned}
\mathit{exec}\ (\mathit{comp}'\ x\ \mathit{cxt}\ c')\ (s,\ vs) &= \mathit{exec}\ c'\ (\mathit{eval}\ x\ bs : s,\ vs)\\
\mathit{exec}\ (\mathit{comp}'\ y\ (v : \mathit{cxt})\ c'')\ (s,\ \chi : vs) &= \mathit{exec}\ c''\ (\mathit{eval}\ y\ ((v,\ \chi) : \mathit{Zip}\ \mathit{cxt}\ vs) : s,\ vs)
\end{aligned}$$

NB:The code arguments are c' and c'' here because we know we need a code instruction to perform the binding, making it different to c, and c'' is the code after the binding has been made.

To better fit the induction hypothesis for y, apply the definition of Zip to the last step in the calculation where $\chi = \mathit{eval}\ x\ bs,\ bs = \mathit{Zip}\ \mathit{cxt}\ vs$

$$(v,\ \mathit{eval}\ x\ bs) : bs = (v,\ \chi) : bs = (v,\ \chi) : (\mathit{Zip}\ \mathit{cxt}\ vs)$$

To be able to use the induction hypothesis for y, we need to have some value on vs to take the place of χ, this value is unknown but is definitely an integer[9].

$$\mathit{exec}\ c''\ (s,\ \chi : vs) = \mathit{exec}\ c\ (s,\ vs)$$

Substituting the specific value χ for the general value n

$$\begin{aligned}
TEL\ &::\ \mathit{Code} \to \mathit{Code}\\
\mathit{exec}\ (TEL\ c)\ (s,\ n : vs)\ &= \mathit{exec}\ c\ (s,\ vs)
\end{aligned}$$

That is, TEL removes the top the of the values stack. A variable would have been bound to it as we will see, but we will never be in a situation where we refer the wrong value to a variable, because by construction a

Using this to continue the calculation

[9] at the moment it's type is the only thing that matters, not it's value, but it will always be χ because of the next step

$$
\begin{aligned}
&\quad exec\ c\ (eval\ y\ (\ (v,\ eval\ x\ bs) : bs) : s,\ vs)\\
&= \{\text{defintion of } Zip\}\\
&\quad exec\ c\ (eval\ y\ (v, \chi) : (Zip\ cxt\ vs) : s,\ vs)\\
&= \{\text{defintion of } exec\ TEL\}\\
&\quad exec\ (TEL\ c)\ (v, \chi) : (Zip\ cxt\ vs) : s,\ \chi : vs)\\
&= \{\text{induction hypothesis } for\ y\}\\
&\quad exec\ (comp'\ y\ (v : cxt)\ (TEL\ c))\ (s,\ \chi : vs)
\end{aligned}
$$

Now to be able to use the induction hypothesis for x we need a value to not be on vs but rather on s. We solve the equation

$$exec\ c'\ (\chi : s,\ vs) = exec\ c\ (s,\ \chi : vs)$$

Substituting the specific value χ for the general value n

$$
\begin{aligned}
LET &:: Code \rightarrow Code\\
exec\ (LET\ c)\ (n : s,\ vs) &= exec\ c\ (s,\ n : vs)
\end{aligned}
$$

continuing the calculation

$$
\begin{aligned}
&\quad exec\ (comp'\ y\ (v : cxt)\ (TEL\ c))\ (s,\ \chi : vs)\\
&= \{\text{defintion of } exec\ LET\}\\
&\quad exec\ (LET\ (comp'\ y\ (v : cxt)\ (TEL\ c)))\ (\chi : s,\ vs)\\
&= \{\chi = eval\ x\ bs\}\\
&\quad exec\ (LET\ (comp'\ y\ (v : cxt)\ (TEL\ c)))\ ((eval\ x\ bs) : s,\ vs)\\
&= \{\text{induction hypothesis } for\ x\}\\
&\quad exec\ (comp'\ x\ cxt\ (LET\ (comp'\ y\ (v : cxt)\ (TEL\ c))))\ (s,\ vs)
\end{aligned}
$$

From this we conclude

$$comp'\ (Let\ v\ x\ y)\ cxt\ c = comp'\ x\ cxt\ (LET\ (comp'\ y\ (v : cxt)\ (TEL\ c)))$$

4.4 Testing

5 Function definition, L_f

6 Automated testing and calculation checking

7 Conclusion

References

[1] R. Hunter, *COMPILERS: Their Design and Construction Using Pascal.* John Wiley and Sons, Ltd, 1985.

[2] E. Meijer, "Calculating compilers," Ph.D. dissertation, Katholieke Universiteit Nijmegen, 1992.

[3] R. C. Backhouse, *Program Construction: Calculating Implementations from Specificiations.* John Wiley and Sons, Inc, 2003.

[4] P. Bahr and G. Hutton, "Calculating correct compilers," *Journal of Functional Programming*, 2015.

[5] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing company, 1988.