Calculating a correct compiler using the Bahr and Hutton method

Marco Jones

April 22, 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 operations which can be used 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 inefficient on a large scale, so programmers have developed higher level programming languages by abstracting away from the details of how a computer works, these languages are easier for programmers to interpret than machine code.

A programming language is defined by a compiler and virtual machine. A compiler translates the high level *source language* into a lower level *target language*. The end result of compilation is the program translated into assembly language, from here an assembler translates the assembly code into machine code for the computer to execute.

Compilers are written in a programming language just like any other program and developed by a similar process; a programmer has an idea of what features a program is to have, implements, and performs tests to verify it's correctness.

Bahr and Hutton have developed a simple technique, for calculating a compiler and virtual machine via equational reasoning for a given source language and its' semantics [1], whilst simultaneously guaranteeing it's correctness by virtue of *constructive induction* [2].

Through the inductive process we discover and invent definitions of our compiler and virtual machine without needing to wholly specify how the machine should operate before hand. The 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 their paper [1]. The compiler and virtual machine we will calculate will be for a source language which extends upon the Arithmetic language example[1, 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 [1]. Each extension of the language will be labelled as it's own language, it's compiler and

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

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.

1.2 Roadmap

Section 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 sections 3 and 4 we will derive compiler and virtual machine definitions for: a conditional function, and variable declaration function.

In section 6 we will discuss the method of systematically testing the compiler and virtual machine.

2 A review of the Bahr and Hutton method

Sections 2.1 - 2.4 of Bahr and Hutton's paper describe the unrefined method in detail, only to refine the process in section 2.5 [1, 2.5 Combining the transformation steps], resulting in a much simpler 3 step process, this paper will only be concerned with their refined 3 step method[1, 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 section 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.

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

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.

$$eval\ (Add\ (Val\ 1)\ (Val\ 2))$$

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 [1].

In sections 2.1 - 2.4 Bahr and Hutton *derived* four components and two correctness equations[1] [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.

$$exec\ (comp\ x)\ s \quad = \quad eval\ x : s \tag{3}$$
$$exec\ (comp'\ x\ c)\ s \quad = \quad exec\ c\ (eval\ x : s) \tag{4}$$

Calculations begin in the form the specification of compiler correctness i.e 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 satisfies the specification[1].

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[3]. 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.

2.1 Example calculations

To introduce the Bahr and Hutton method, we will follow the calculation of *comp*and *exec* definitions for *Val* and *Add* expressions[1, section 2.5].

Starting from the compiler specification (3) where the expression x is the base case $Val\ n$, the calculation proceeds as follows[1]:

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

Now there are no further definitions that we can apply, but we must invent a definition for *exec* which allows us to proceed; by solving the equation:

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

Currently our 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 we cannot use this equation as a definition for *exec* or *comp*. 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[1, bottom of page 9],

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

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.

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

Our equation is now in the 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

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

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

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

Next Bahr and Hutton calculate definitions for the inductive case, $Add\ x\ y$, we call it inductive because we don't yet know the values of x and y however we are assuming that they are expressions as well, otherwise there would be a type error. Starting from a similar point,

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

Again we are stuck, however similar to before, we can make 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 tell us what to compile in order to achieve a certain stack state.

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

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

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 we need to use both hypotheses one after the other. With this in mind, we need to have stack with both *eval* x and y on top, i.e. *eval* x : *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, we will solve the equation:

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

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

$$\begin{array}{rcl} ADD & :: & Code \rightarrow Code \\ exec\ (ADD\ c)\ (m : n : s) & = & exec\ c\ (n + m : s) \end{array} \tag{9}$$

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.

Now we have the operation definition for the virtual machine we continue the calculation

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

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

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

In summary, we have discovered a definition for the compiler

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

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 it. After that we applied induction hypotheses to find definitions for the compiler.

Proceeding like this, we can define more source code to expand our arithmetic language.

An important aspect of the Bahr and Huttonmethod is that the equations that construct the compiler serve as proof of it's correctness, and so they explain that a derivation such as this one can be read as a proof [1, page 14]. This will not show us much at the moment, but will be used later on.

In summary Bahr and Hutton calculated the following definitions[2] for the compiler and virtual machine:

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

2.2 Testing Add and Val expressions

To conclude this section I will test the functionality of Add and Val, with a somewhat complicated example. This will give us a chance to look at the code that their compiler generates and how the virtual machine executes it.

In checking these calculations, we can do several tests: Calculate and compare by hand the code that is produced or executed by the compiler definitions, against the results using the Haskell implementation in GHCi. Compare the result of the executed code against the result given by the interpreter. This result is arguably the most important because if a counter example is found, then our specification for compiler correctness

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

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

The equation we'll be looking at is:

$$Add\ (Add\ (Val\ 0)\ (Val\ 1))\ (Val\ 2))$$

This expression was chosen because it will test that each sub-expression is compiled properly, this is more of a general test of the *comp* and *comp'*

[2]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

functions. In future we won't need to use such complicated examples, especially when functions become more complicated.

2.2.1 Compilation

$$
\begin{aligned}
& comp\ (Add\ (Add\ (Val\ 0)\ (Val\ 1))\ (Val\ 2)) \\
& = \{definition\ 10\} \\
& comp'\ (Add\ (Add\ (Val\ 0)\ (Val\ 1))\ (Val\ 2))\ (HALT) \\
= & \{definition\ 11\} \\
& comp'\ (Add\ (Val\ 0)\ (Val\ 1))\ (comp'\ (Val\ 2)(ADD\ HALT)) \\
= & \{definitions\ 11\ and\ 6\ \} \\
& comp'\ (Val\ 0)\ (comp'\ (Val\ 1)\ (ADD\ (PUSH\ 2\ (ADD\ HALT)))) \\
= & \{definition\ 6\ twice\ \} \\
& PUSH\ 0\ (PUSH\ 1\ (ADD\ (PUSH\ 2\ (ADD\ HALT))))
\end{aligned}
$$

This agrees with the same expression being compiled using the Haskell implementation of the compiler. GHCi: $PUSH\ 0\ (PUSH\ 1\ (ADD\ (PUSH\ 2\ (ADD\ HALT))))$

2.2.2 Execution

The LHS contains the function representing our virtual machine and the RHS is the current state of the run-time stack

$$
\begin{aligned}
& exec\ (PUSH\ 0\ (PUSH\ 1\ (ADD\ (PUSH\ 2\ (ADD\ HALT))))) && [\,] \\
= & \{definition\ 13\ twice\} \\
& exec\ (ADD\ (PUSH\ 2\ (ADD\ HALT))) && [1,\ 0] \\
= & \{definition\ 14\} \\
& exec\ (PUSH\ 2\ (ADD\ HALT)) && [1] \\
= & \{definition\ 13\} \\
& exec\ (ADD\ HALT) && [2,\ 1] \\
= & \{definition\ 14\} \\
& exec\ HALT && [3] \\
= & \{definition\ 12\} \\
& [3]
\end{aligned}
$$

2.2.3 Interpretation

$$
\begin{aligned}
eval(Val\ n) &= n \\
eval(Add\ x\ y) &= eval\ x + eval\ y
\end{aligned}
$$

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

The results of the execution and interpretation test agree with each other and with the same tests using the Haskell implementation of the virtual machine and interpreter. GHCi: [3], 3

3 Conditionals, L_c

From now on this dissertation will report on my investigation into applying the Bahr and Hutton method to develop a compiler with definitions not defined in CCC.

To start off with, we will derive 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 is: "define an evaluation function in a compositional manner". We are still using the same *eval* function from before, but we need to define a new expression and it's semantics.

Haskell conditionals concrete syntax "if x then y else z", however without a parser to do lexical analysis[3, chapter 2.2] of the lexemes[3], 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.

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

the semantics of it will be

$$eval(Ite\ x\ y\ z) = if\ eval\ x \neq 0 then\ eval\ y\ else\ eval\ z \tag{15}$$

The condition $eval\ x \neq 0$ 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.

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

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

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

3.1 Calculation

Step 3: "Calculate definitions that satisfy these specifications"

In order to satisfy specification 2, we begin with it's LHS where x is our Ite expression.

$$\begin{aligned} & exec\ (comp'\ (Ite\ x\ y\ z)\ c)\ s \\ =& \{specification\ (2)\} \\ & exec\ c\ (eval\ (Ite\ x\ y\ z) : s) \\ =& \{\text{defintion of } eval\} \\ & exec\ c\ (if\ eval\ x \neq 0\ then\ eval\ y\ else\ eval\ z : s) \end{aligned}$$

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:

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

However, to be able to use them, we must push $eval\ x, y, z$ onto the stack in some order of our choice. So we must solve the generalised equation:

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

Our code constructor to solve this will be

$$ITE :: Code \to Code$$

and it's definition for the virtual machine

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

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

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

I conclude from this calculation these new definitions for the compiler and virtual machine:

$$comp'\ (Ite\ x\ y\ z)\ c = comp'\ z\ (comp'\ y\ (comp'\ x\ (ITE\ c))) \quad (16)$$

$$exec\ (ITE\ c)\ (k : m : n : s) = exec\ c\ ((if\ k \neq 0\ then\ m\ else\ n) : s) \quad (17)$$

3.2 Testing

For the testing, we can create an example Ite expression

$$Ite\ (Val\ 1)(Add\ (Val\ 2)(Val\ 3))(Add\ (Val\ 4)(Val\ 5))))$$

This expression would test that each of the sub-expressions (*Add* and *Val*) compile properly first, and that the result of the condition is correct [4].

[4]For completeness we would need to test both True and False outcomes of the condition, such tests are included in the supporting Haskell files, however the by hand calculations are omitted for brevity because they would not add much new information to these series of tests

3.2.1 Compilation

$$
\begin{aligned}
& comp'(Ite\ (Val\ 1)(Add\ (Val\ 2)\ (Val\ 3))(Add\ (Val\ 4)\ (Val\ 5))) \\
= & \{equation\ 16\} \\
& comp'\ (Add\ (Val\ 4)\ (Val\ 5)) \\
& (comp'\ (Add\ (Val\ 2)\ (Val\ 3)) \\
& (comp'\ (Val\ 1)\ (ITE\ HALT))) \\
= & \{equations\ 11\ twice,\ and\ 6\ once\} \\
& comp'\ (Val\ 4)\ (comp'\ (Val\ 5) \\
& (ADD\ (comp'\ (Val\ 2)\ (comp'\ (Val\ 3) \\
& (ADD\ (PUSH\ 1\ (ITE\ HALT))))))) \\
= & \{equation\ 3,\ 4\ times\} \\
& PUSH\ 4\ (PUSH\ 5\ (ADD\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ (PUSH\ 1\ (ITE\ HALT)))))))
\end{aligned}
$$

This agrees with the same expression being compiled using the Haskell implementation of the compiler. GHCi:

$$PUSH\ 4\ (PUSH\ 5\ (ADD\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ (PUSH\ 1\ (ITE\ HALT)))))))$$

3.2.2 Execution

In executing this code with an empty stack, we get: [5]

By hand, this calculation is as follows:

$$
\begin{aligned}
& exec\,(PUSH\,4\,(PUSH\,5\,(ADD\,(PUSH\,2\,(PUSH\,3\,(ADD\,(PUSH\,1\,(ITE\,HALT)))))))) && [\,] \\
= & \{equation\ 13,\ twice\} \\
& exec\ (ADD\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ (PUSH\ 1\ (ITE\ HALT)))))) && [5,4] \\
= & \{equation\ 14\} \\
& exec\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ (PUSH\ 1\ (ITE\ HALT))))) && [9] \\
= & \{equation\ 13, twice\} \\
& exec\ (ADD\ (PUSH\ 1\ (ITE\ HALT))) && [3,2,9] \\
= & \{equation\ 14\} \\
& exec\ (PUSH\ 1\ (ITE\ HALT)) && [5,9] \\
= & \{equation\ 13\} \\
& exec\ (ITE\ HALT) && [1,5,9] \\
= & \{equation\ 17\} \\
& exec\ HALT && [5] \\
= & \{equation\ 12\} \\
& [5]
\end{aligned}
$$

Interpreting the expression[5]

[5]Recall that the interpreter returns an Int rather than a stack

$$eval\ (Ite\ (Val\ 1)\ (Add\ (Val\ 2)\ (Val\ 3))\ (Add\ (Val\ 4)\ (Val\ 5))) = 5$$

As the virtual machine and interpreter are in agreement, we have shown an example of the compiler satisfying the compiler correctness specification

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

3.2.3 Interpretation

In retrospect, this kind of evaluation of the condition is eager; both cases regardless of the condition's result, are executed while only one branch of code needs to be. This can avoided at if we can instead evaluate the condition at compile time and throw away the code for the case that we don't need to execute.

3.3 Lazy evaluation

Our "Lazy if then else" function will be called *Lite*.

$$\textbf{data}\ Expr = ...|Lite\ Expr\ Expr\ Expr$$

and it's semantics are

$$eval(Lite\ x\ y\ z) = if\ eval\ x \neq 0 then\ eval\ y\ else\ eval\ z \tag{18}$$

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

3.3.1 Calculation

Our inductive calculation of *Lite*, begins in the same way

$$\begin{aligned}
&exec\ (comp'\ (Lite\ x\ y\ z)\ c)\ s \\
=&\{specification\ (2)\} \\
&exec\ c\ (eval\ (Lite\ x\ y\ z) : s) \\
=&\{\text{defintion of } eval\} \\
&exec\ c\ (if\ eval\ x\ \neq 0\ then\ eval\ y\ else\ eval\ z : s)
\end{aligned}$$

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

$$LITE :: Code \to Code \to Code$$

$$exec\ (LITE\ ct\ ce)\ (eval\ x : s) = exec\ c\ (if\ eval\ x\ \neq 0\ then\ eval\ y\ else\ eval\ z : s)$$

We cannot use this equation as a definition of exec because c, y and z are unbound in the body of the expression[1, page 10]. However we can bind them in *ct* and *ce*. The compile function *comp* requires that any code is followed by more code (unless it is a HALT), so not only do *ct* and *ce* contain the code for expressions y and z but also the continuation code c

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

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

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

This method poses a problem; on the right hand side of this equation, c appears twice, meaning the code not only doubles in length, but doubles in *compile time*. This would cause a compile time-complexity of $T(n) = \mathcal{O}(2^n)$ where n is the number of *Lite* expressions. 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.

$$comp'\ (Lite\ x\ y\ z)\ c$$
$$= comp'\ x\ (LITE\ (comp'\ y\ c)\ (comp'\ z\ c)) \tag{19}$$
$$exec\ (LITE\ ct\ ce)\ (eval\ x : s)$$
$$= exec\ c\ (if\ eval\ x\ \neq 0\ then\ eval\ y\ else\ eval\ z : s) \tag{20}$$

3.3.2 Testing

To test our definitions we can use the Haskell compiler to compile and execute an example *Lite*expression, which will be the same expression as we used on it's the eager twin for comparison.

$$exec\ (comp\ (Lite\ (Val\ 1)\ (Add\ (Val\ 2)\ (Val\ 3))\ (Add\ (Val\ 4)\ (Val\ 5))))\ [\,]$$

3.3.3 Compilation

Compiling by hand:

$comp\ (Lite\ (Val\ 1)\ (Add\ (Val\ 2)\ (Val\ 3))\ (Add\ (Val\ 4)(Val\ 5)))$
$= \{$defintion of $comp\ followed\ by\ equation\ 19\ \}$
$comp'\ (Val\ 1)(LITE\ (comp'\ (Add\ (Val\ 2)\ (Val\ 3))\ HALT)\ (comp'\ (Add\ (Val\ 4)\ (Val\ 5))))$
$= \{equation\ 6, 11\}$
$PUSH\ 1\ (LITE\ (comp'\ (Add\ (Val\ 2)\ (Val\ 3))\ HALT)\ (comp'\ (Add\ (Val\ 4)\ (Val\ 5))))$
$= \{equation\ 8\ twice\}$
$PUSH\ 1\ (LITE\ (comp'\ (Val\ 2)\ (comp'\ (Val\ 3)\ (ADD\ HALT))\ (comp'\ (Val\ 4)\ (comp'\ (Val\ 5)\ (ADD$
$= \{equation\ 9\ four\ times\}$
$PUSH\ 1\ (LITE\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ HALT)))(PUSH\ 4\ (PUSH\ 5(ADD\ HALT))))$

Using the compile function
Comp on this expression in the Haskell implementation yields the same result:

$comp\ (Lite\ (Val\ 1)\ (Add\ (Val\ 2)\ (Val\ 3))\ (Add\ (Val\ 4)\ (Val\ 5))))$
$= PUSH\ 1\ (LITE\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ HALT)))(PUSH\ 4\ (PUSH\ 5(ADD\ HALT))))$

3.3.4 Execution

$$
\begin{aligned}
&exec\ (PUSH\ 1\ (LITE\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ HALT)))(PUSH\ 4\ (PUSH\ 5(ADD\ HALT))))))\ [\,] \\
&= \{equation\ 13\} \\
&exec\ (LITE\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ HALT)))(PUSH\ 4\ (PUSH\ 5(ADD\ HALT))))\ [1] \\
&= \{equation\ 20\} \\
&exec\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ HALT)))\ [\,] \\
&= \{equation\ 13\ twice\} \\
&exec\ (ADD\ HALT)\ [3,2] \\
&= \{equation\ 14\} \\
&exec\ HALT\ [5] \\
=&\{equation\ 12\} \\
&[5]
\end{aligned}
$$

Using the Haskell implementation of the virtual machine.

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

3.3.5 Interpretation

$$
\begin{aligned}
&eval\ (Lite\ (Val\ 1)(Add\ (Val\ 2)(Val\ 3))(Add\ (Val\ 4)(Val\ 5))) \\
=&\{equation\ 18\}
\end{aligned}
$$

In the Haskell implementation of the interpreter:

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

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

3.3.6 Proof

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

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

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

3.4 Summary

In conclusion we have calculated the following definitions [1, page 11]:

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

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[1, 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.

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

All of our functions have been calculated without need of environments, therefore we can be reasonably sure that simply adding in the Envargument 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.

Therefore our semantics of *Let* and Var

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

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

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

$$
\begin{aligned}
eval\ (Add\ (Var\ \text{“}a''\text{)}\ (Val\ 2))\ [(\text{“}a''\text{, } 2)] &= 4,\ [\text{“}a''\text{, } 2] \\
eval\ (Let\ \text{“}a''\ (Val\ 2)\ (Add\ (Var\ a)\ (Val\ 2)))\ [\,] &= 4,\ [(\text{“}a''\text{, } 2)] \\
eval\ (Let\text{“}b''\ (Let\ \text{“}a''\ (Val\ 2)\ (Add\ (Var\ a)\ (Val\ 2)))\ (Add\ (Var\ b)\ (Val\ 2)))\ [\,] &= 6,\ [(\text{“}b''\text{, } 4),\ (\text{“}a''\text{, } 2)] \qquad (21) \\
eval\ (Let\text{“}a''\ (Let\ \text{“}b''\ (Val\ 2)\ (Add\ (Var\ a)\ (Val\ 2)))\ (Add\ (Var\ b)\ (Val\ 2)))\ [\,] &= \text{“}Binding\ b\ out\ of\ scope'' \qquad (22)
\end{aligned}
$$

In equation (21) the second *Add* inherits the scope of *Let* sub-expression preceding it, because it evaluates *within scope* of it. Conversely with equation (22) 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:

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

[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

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

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 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 when and what variables are called.

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

*

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

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

Which cannot be used because bs is still unbound. bs is just a set of name-value pairs, now we have variable names that don't have paired values, 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

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

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 *eval* use the *bs* symbol for environments, it will be useful to formally state:

$$Zip\ (x : xs)\ (y : ys) = (x,\ y) : (Zip\ xs\ ys) \tag{23}$$

$$Zip\ cxt\ vs = bs \tag{24}$$

These equations satisfy the full description of step 2 in their Bahr and Hutton's General methodology [1, 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 ?THM? **??**[1, page 42].

$$\begin{aligned} & exec\ (comp'\ (Let\ v\ x\ y)\ cxt\ c)\ (s,\ vs) \\ =& \{specification\ 4\} \\ & exec\ c\ (eval\ (Let\ v\ x\ y)\ (Zip\ cxt\ vs) : s,\ vs) \\ =& \{\text{defintion of } Zip,\ \text{defintion of } eval\} \\ & exec\ c\ (eval\ y\ (\ (v,\ eval\ x\ bs) : bs) : s,\ vs) \end{aligned}$$

There are no more definitions to apply.

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.

[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 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 sub-expression 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 (24), where x and y are the new v and χ.

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

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

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

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 $\chi = eval\ x\ bs,\ bs = Zip\ cxt\ vs$

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

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].

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

Substituting the specific value χ for the general value n

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

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

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

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

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

continuing the calculation

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

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

References

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

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

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