

Calculating a correct compiler using the Bahr and Hutton method

Marco Jones

Contents

1	Introduction	1
1.1	Methodology	2
1.2	The Bahr and Hutton method	2
1.3	Example calculations	4
1.4	Testing Add and Val expressions	6
2	Conditionals	7
2.1	Calculation	7
2.1.1	Testing	9
2.2	Lazy evaluation	11
2.2.1	Calculation	12
2.3	Summary	15
3	Bindings	15
3.1	Semantics	16
3.2	Calculation	17

1 Introduction

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].

Traditionally compilers are derived by

However through the inductive process we discover and invent definitions of our compiler and virtual machine without needing consideration of an abstract syntax tree. The result, is an equational program mainly consisting of definitions of a pair functions, “comp” and “exec”, representing the compiler and virtual machine, respectively.

The aim of this dissertation is to document the application of this method to a source language which extends upon the examples given in Calculating Correct Compilers (CCC), and will do so in three stages.

1.1 Methodology

Firstly by summarising the Bahr and Hutton method, using the arithmetic language derivation as described in CCC [Section 2], 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.

After each of calculations we will verify them, using an example expression and few tests:

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.
4. Prove the equations meet the specification[1, page 14]

Thirdly, we will use automated testing, 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 The Bahr and Hutton method

Sections 2.1 - 2.4 of CCC describe steps 1 - 4 of the method, only to have steps 2 - 4 combined in section 2.5 [1, 2.5 Combining the transformation steps], resulting in a much simpler 3 step process, thus we will use their refined method[1, page 12].

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 "to make the manipulation of argument values explicit".

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

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

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

type *Stack* = [*Int*]

data in Haskell creates a new type, here we define *Exprs* either being a *Val* or an *Add*, these tags are called *constructors*, a *Val* constructor will always precede and integer (which is “Int” in Haskell), and an *Add* will always precede 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*. 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 is a valid *Expr*, the function application will be type correct and we may continue.

On the right hand side of the equations is a description of how to compute the result when *eval* is applied to *Val* and *Add* expressions respectively. *eval Val n* simply returns *n* on the other hand *eval Add x y* is recursively defined as we do not yet know the values of *x* and *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 proofs and definitions *partial* in the Bahr and Hutton method. Although Bahr and Hutton explore when this is not possible, 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* → *Code* that compiles source expressions to code.
- A function *comp'* :: *Expr* → *Code* → *Code* that also takes a code continuation as input.
- A function *exec* :: *Code* → *Stack* → *Stack* that provides a semantics for code by modifying a run-time stack.

Specification 1 (Bahr and Hutton 3)

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

Specification 2 (Bahr and Hutton 4)

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

NB: Bahr and Hutton have labelled them 3 and 4 respectively, and will appear as 3 and 4 in quotations.

“Calculations begin with equations of the form *exec (comp' x c) s* as in 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 definition $comp\ x\ c = c'$ satisfies the specification in this case.”[1]

It is perhaps strange 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 can manipulate all of it by induction. A lot of compilers nowadays take entire programs as a (possibly huge) string of characters and it is up to a preprocessor to collect them together into tokens [3]; $comp$, as we will see, translates the source code into a list of instructions of type *code* for the virtual machine to execute.

1.3 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 equation 4 and the expression x being the base case $Val\ n$, the calculation proceeds as follows[1]: (recall: $eval(Val\ x) = n$)

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

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

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

Currently we have the right hand side of this equation, however c' is a new variable only on one side of the equation so we say it is *unbound*. 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 $y = x + z$ | *where* $z = 1$, we cannot know the value of y if x is not given.

“The solution is to package these two variables up in the code argument c' by means of a new constructor in the *Code* data type that takes these two variables as arguments,”

$$PUSH :: Int \rightarrow Code \rightarrow Code$$

“and define a new equation for $exec$ as follows:”

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

“executing the code $PUSH\ n\ c$ proceeds by pushing the value n onto the stack and then executing the code c ”, furthermore we can see that we have n and c on both sides of the equation and therefore no longer unbound.

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

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

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

or more specifically, we have *discovered* that
 $comp'\ (Val\ n)\ c = PUSH\ n\ c$

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, because otherwise there would be a type error. Starting from a similar point, (recall: $eval(Add\ x\ y) = eval\ x + eval\ y$)

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

Again we are stuck, however using a similar process as before, we can make a new definition for $exec$. Moreover being an inductive case, we can make use of the induction hypotheses for x and y [1].

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

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

Although all that is on top of the stack on the RHS of this equation is $eval\ x$ or $eval\ y$, but we want to use both values to make the addition. The solution is to do one after the other, but nonetheless we know we can have the top two stack elements to be $eval\ x$ or $eval\ y$. Therefore we can use a new *Code* constructor that when executed, adds the top two stack elements together and puts the result on top of the stack.

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

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 n or (as we will see) $eval\ x$ on first, again for consistency with CCC, I have used their definition.

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

$$\begin{aligned}
&= \{\text{definition of } exec\} \\
&exec\ (ADDc)\ (eval\ y : eval\ x : s) \\
&= \{\text{induction hypothesis for } y\} \\
&exec\ (comp'y(comp'x(ADDc)))s
\end{aligned}$$

We can conclude from this

$$exec\ (comp'\ (Add\ x\ y)\ c)\ s = exec\ (comp'y(comp'x(ADDc)))s$$

An important aspect of the Bahr and Hutton method is that the equations that construct the compiler serve as proof of its 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 definitions² for the compiler and virtual machine:

data <i>Code</i>	$= HALT PUSHIntCode ADDCode$
<i>comp</i>	$:: Expr \rightarrow Code$
<i>comp</i> x	$= comp'\ x\ HALT$
<i>comp'</i>	$:: Expr \rightarrow Code \rightarrow Code$
<i>comp'</i> $(Val\ n)\ c$	$= PUSH\ n\ c$
<i>comp'</i> $(Add\ x\ y)\ c$	$= comp'\ x\ (comp'\ y\ (ADD\ c))$
<i>exec</i>	$:: Code \rightarrow Stack \rightarrow Stack$
<i>exec</i> $HALT\ s$	$= s$
<i>exec</i> $(PUSH\ n\ c)\ s$	$= exec\ c\ (n : s)$
<i>exec</i> $(ADD\ c)\ (m : n : s)$	$= exec\ c\ ((n + m) : s)$

1.4 Testing Add and Val expressions

To conclude this section I will test the functionality of Add and Val, with some example expressions. These expressions will be nested with more expressions to be somewhat more complicated than what we have derived. This will also give us a chance to look at the code that our compiler generates and how the virtual machine executes it.

²The 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 is only a small point

2 Conditionals

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 lexemes³, 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.

data *Expr* = ...|*Ite Expr Expr Expr*

the semantics of it will be

$$eval(Ite\ x\ y\ z) = if\ eval\ x\ \neq\ 0\ then\ eval\ y\ else\ eval\ z$$

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.

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

$$exec\ (comp\ x)\ s \quad =\ eval\ x : s \tag{1}$$

$$exec\ (comp'\ x\ c)\ s \quad =\ exec\ c\ (eval\ x : s) \tag{2}$$

2.1 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.

³“if”, “then” and “else” in this case

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

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

$$ITE :: Code \rightarrow Code$$

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

$$\begin{aligned}
& exec c (if eval x \neq 0 then eval y else eval z : s) \\
& = \{definition of exec\} \\
& exec (ITE c) (eval x : eval y : eval z : s) \\
& = \{inductionhypothesisforx\} \\
& exec (comp' x (ITE c)) (eval y : eval z : s) \\
& = \{inductionhypothesisfory\} \\
& exec (comp' y (comp' x (ITE c))) (eval z : s) \\
& = \{inductionhypothesisforz\} \\
& exec (comp' z (comp' y (comp' x (ITE c)))) s
\end{aligned}$$

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

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

2.1.1 Testing

To check these calculations, we can do several tests⁴:

1) Calculate by hand the code that should be produced by the compiler, and compare it to the code that is produced by the compiler which uses the derived definition. 2) Execute this code by hand and with GHCi and compare results. 3) Compare the result of the executed code against the result given by the interpreter. After these tests we can prove the equations meet the specification by the method that Bahr and Hutton describe in CCC [1, page 14; derivation vs proof]

For the testing, we can create an example *Ite* expression

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

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

The resulting code after applying the *comp* function is

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

Which is correct, as given by the following induction:

$$\text{comp}' (\text{Ite } x \ y \ z) \ c = \text{comp}' z(\text{comp}' y (\text{comp}' x (\text{ITE } c))) \quad (3)$$

$$\text{comp}' (\text{Add } x \ y) \ c = \text{comp}' x(\text{comp}' y(\text{ADD } c)) \quad (4)$$

$$\text{comp}' (\text{Val } n) \ c = \text{PUSH } n \ c \quad (5)$$

$$\text{comp}'(\text{Ite } (\text{Val } 1)(\text{Add } (\text{Val } 2)(\text{Val } 3))(\text{Add } (\text{Val } 4)(\text{Val } 5)))$$

$$= \{\text{equation 1}\}$$

$$\text{comp}'(\text{Add } (\text{Val } 4)(\text{Val } 5))(\text{comp}'(\text{Add } (\text{Val } 2)(\text{Val } 3))(\text{comp}' (\text{Val } 1) (\text{ITE } \text{HALT})))$$

$$= \{\text{equation 2, twice}\}$$

$$\text{comp}'(\text{Val } 4)(\text{comp}'(\text{Val } 5)(\text{ADD}(\text{comp}'(\text{Val } 2)(\text{comp}'(\text{Val } 3)(\text{ADD}(\text{comp}'(\text{Val } 1)(\text{ITE } \text{HALT})))))))$$

$$= \{\text{equation 3, 5 times}\}$$

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

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

⁴We do need to test both True and False outcomes of the condition, for completeness, such tests are included in the Haskell files, however the details of their workings are omitted for brevity because they would not add much new information to this dissertation

By hand, this calculation is as follows:

$$exec (PUSH\ n\ c)s = exec\ c\ (n : s) \quad (6)$$

$$exec (ADD\ c)\ s = exec\ c\ ((n + m) : s) \quad (7)$$

$$exec (ITE\ c)\ (k : m : n : s) = execc((if\ k \neq 0\ then\ m\ else\ n) : s) \quad (8)$$

$exec (PUSH\ 4(PUSH\ 5(ADD(PUSH\ 2(PUSH\ 3(ADD(PUSH\ 1(ITE\ HALT))))))) \square$
 $= \{equation\ 4, twice\}$
 $exec (ADD (PUSH\ 2 (PUSH\ 3 (ADD (PUSH\ 1 (ITE\ HALT))))) [5, 4]$
 $= \{equation\ 5\}$
 $exec (PUSH\ 2 (PUSH\ 3 (ADD (PUSH\ 1 (ITE\ HALT))))) [9]$
 $= \{equation\ 4, twice\}$
 $exec (ADD (PUSH\ 1 (ITE\ HALT))) [3, 2, 9]$
 $= \{equation\ 5\}$
 $exec (PUSH\ 1 (ITE\ HALT)) [5, 9]$
 $= \{equation\ 4\}$
 $exec (ITE\ HALT) [1, 5, 9]$
 $= \{equation\ 6\}$
 $exec\ HALT [5]$
 $= \{definition\ of\ exec\}$
 $[5]$

Applying the interpreter to the expression⁵

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

As the virtual machine and interpreter are in agreement, now we can prove that the definition is correct, by showing that it satisfies the compiler correctness specification, i.e we show

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

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

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

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 be avoided 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.

2.2 Lazy evaluation

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

data *Expr* = ... | *Lite Expr Expr Expr*

and its semantics are

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

which are the same as the semantics of an *Ite* expression as expressions cannot be lazily evaluated, 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.

2.2.1 Calculation

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

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

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 compiled y and z expressions.

$$LITE :: Code \rightarrow Code \rightarrow Code$$

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

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

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

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

Testing

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

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

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.

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

the result of comp:

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

$$comp' \ (Lite \ x \ y \ z) \ c = comp' \ x(LITE \ (comp' \ y \ c) \ (comp' \ z \ c)) \quad (9)$$

$$comp' \ (Add \ x \ y) \ c = comp' \ x(comp' \ y(ADD \ c)) \quad (10)$$

$$comp' \ (Val \ n) \ c = PUSH \ n \ c \quad (11)$$

Compiling by hand:

$comp\ (Lite\ (Val\ 1)(Add\ (Val\ 2)(Val\ 3))(Add\ (Val\ 4)(Val\ 5)))$
 $= \{definition\ of\ comp\ followed\ by\ equation\ 7\}$
 $comp'\ (Val\ 1)(LITE\ (comp'\ (Add\ (Val\ 2)(Val\ 3))\ HALT)\ (comp'\ (Add\ (Val\ 4)(Val\ 5))))$
 $= \{equation\ 9\}$
 $PUSH\ 1\ (LITE\ (comp'\ (Add\ (Val\ 2)(Val\ 3))\ HALT)\ (comp'\ (Add\ (Val\ 4)(Val\ 5))))$
 $= \{equation\ 8\ twice\}$
 $PUSH\ 1\ (LITE\ (comp'\ (Val\ 2)(comp'\ (Val\ 3)(ADD\ HALT))\ (comp'\ (Val\ 4)(comp'\ (Val\ 5)(ADD\ HALT))))$
 $= \{equation\ 9\ four\ times\}$
 $PUSH\ 1\ (LITE\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ HALT)))(PUSH\ 4\ (PUSH\ 5(ADD\ HALT))))$

So yet again, the expression has compiled correctly.

the result of exec: [5]

Which agrees with the interpretation of the expression:

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

So we can move onto checking the execution by hand

$$exec\ (LITE\ ct\ ce)\ (k : s) = exec(if\ k \neq 0\ then\ ct\ else\ ce)\ s \quad (12)$$

$$exec\ (PUSH\ n\ c)s = exec\ c\ (n : s) \quad (13)$$

$$exec\ (ADD\ c)\ s = exec\ c\ ((n + m) : s) \quad (14)$$

$exec\ (PUSH\ 1\ (LITE\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ HALT)))(PUSH\ 4\ (PUSH\ 5(ADD\ HALT)))) \ []$
 $= \{equation\ 11\}$
 $exec(LITE\ (PUSH\ 2\ (PUSH\ 3\ (ADD\ HALT)))(PUSH\ 4\ (PUSH\ 5(ADD\ HALT))))\ [1]$
 $= \{equation\ 10\}$
 $exec(PUSH\ 2\ (PUSH\ 3\ (ADD\ HALT)))\ []$
 $= \{equation\ 11\ twice\}$
 $exec(ADD\ HALT)\ [3, 2]$
 $= \{equation\ 12\}$
 $exec\ HALT\ [5]$
 $= \{definition\ of\ exec\}$
 $[5]$

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

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

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

QED

2.3 Summary

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

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

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].

3 Bindings

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 alike 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 its own, it only iterates through expressions and interprets them. *eval* would require at least one more argument containing a set of bindings which it can manipulate.

3.1 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 .

type $Env = [(String, Int)]$

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

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

Because all of our functions have been calculated without need of environments, it means they do not interact with environments at all therefore we can be reasonably sure that simply adding in the Env argument won't affect them⁶.

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

data $Expr = \dots | Let String Expr Expr$

Let creates a new binding, by pushing the String-Int pair onto the Env . The String is taken from the source String part of the *Let* expression, however the value it's paired to needs to be computed inductively. If expressions are now interpreted in the context of a set of bindings, then any sub expression also is, however the *scope* is different.

eval (Let "v" (eval (Let "v" x y)) z)

In this situation the second let should inherit whatever scope the first let was in, because it evaluates *before* the first binding to x happens.

eval (Let "v" x (eval (Let "v" x y)))

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

Conversely when evaluating after a binding, the second let would *overwrite* the binding to x

Therefore our semantics of *Let*

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

But that is only half of the semantics of variables, creating a binding is useless unless we can access the value of “v”. Like *Val*, we need an expression that outputs the value of a given variable when evaluated, and an error is thrown if the variable is not in scope.

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

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.

3.2 Compiler Correctness

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.

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

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

We can learn two things from this: 1) When a variable is declared, it’s value will be available. 2) Clearly the compiler must have some kind of environment of it’s own to reflect changes in the environment. The compiler cannot compute the value parts of each pair, it can however, store the variable names.

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

```

type Context    = [String]
      comp       :: Expr → Code
      comp e     = comp' e [] HALT
      comp'      :: Expr → Context → Code
      comp' x ctx c = comp' c' ctx'(**)

```

****Note:** this is not a definition the the compiler.

But now we have variable names that don't have paired values, this can be left to the virtual machine.

We can use⁸ 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

```

type Memory    = (Stack, Stack)
      exec       :: Code → Memory → Memory
      exec c (s, vs) = exec c' (s', vs')

```

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

Specification 3

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

Specification 4

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

3.3 Calculation

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

References

- [1] P. Bahr and G. Hutton, “Calculating correct compilers,” *Journal of Functional Programming*, 2015.
- [2] R. C. Backhouse, *Program Construction: Calculating Implementations from Specifications*. 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.

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