Calculating a correct compiler with the Bahr and Hutton method

Marco Jones

1 A summary of Calculating Correct Compilers

Bahr and Hutton have developed a simple technique, for calculating a compiler and virtual machinea via equational reasoning for a given source language and its' semantics [1], whilst simultaneously guanteeing 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.

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 similiar nature, and implementing them in Haskell [1]. Testing will occur at the end of each significant set of additions to the language to verify the calculations, even though the calculations themselves can be read as proofs[1, page 14, derivation vs proof].

Thirdly, define a new source language. Finally I will conclude with comments on space and time complexity of the implementation, and further work. Thirdly, describe the tests that I carried out and discuss the results.

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 the refined method.

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

In section 2 they are deriving a compiler and virtual machine for the "Arithmetic" language, they begin by defining: a new Haskell datatype $Expr$; which contain the set of expressions which belong to their source language, an evaluation function, also reffered to as the *interpreter*, which defines their semantics, and a stack of integers "to make the manipulation of argument values explicit".

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

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

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

data in Haskell 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 preceed and integer (which is "Int" in Haskell), and an Add will always preceed two more expressions. It is the constructor *together* with it's argumets that make it of **type** expresssion, 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 parendtheses, 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 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 datatype $Code$ that represents code for the virtual machine.
- A function $comp :: Expr \to Code$ that compiles source expressions to code.
- A function $comp' :: Expr \to Code \to Code$ that also takes a code continuation as input.
- A function $exec :: Code \to Stack \to Stack$ that provides a semantics for code.

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.

"These equations capture the relationships between the semantics, compiler and virtual machine"[1, page 9], because *comp* turns source expressions into code, *exec* turns code together with a stack into another stack, and *eval*give us the semantics of the expressions.

"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 defines the correctness of compilation of our entire source code, moreover all of the source code is contained within c'; however this should be unsurprising as any traditional compiler takes the entire program 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.

2.1 Example calculations

Before I begin my own calculations, I will use the calculations of the compilation and execution of value and addition expressions in section 2.5 of CCC, as simple example calculations, just as they have.

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 $Code$ datatype that takes these two variables as arguments,"

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

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

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

"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 aswell, 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 notheless 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

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

We can conclude from this $exec\ (comp'\ (Add\ x\ y)\ c)\ s = exec\ (comp'y(comp'x(ADDc)))s$

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

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

$$
\begin{array}{rl}
\textbf{data}\ Code & = HALT|PUSHIntCode|ADDCode \\
comp & :: Expr \rightarrow Code \\
comp\ x & = comp'\ x\ HALT \\
comp' & :: Expr \rightarrow Code \rightarrow Code \\
comp'\ (Val\ n)\ c & = PUSH\ n\ c \\
comp'\ (Add\ x\ y)\ c & = comp'\ x\ (comp'\ y(ADD\ c)) \\
exec & :: Code \rightarrow Stack \rightarrow Stack \\
exec\ HALT\ s & = s \\
exec\ (PUSH\ n\ c)\ s & = exec\ c\ (n : s) \\
exec\ (ADD\ c)\ (m : n : s) & = exec\ c\ ((n + m) : s)
\end{array}
$$

2.2 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.

3 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 I derived 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.

Haskell conditionals are formed using an "if then else" line, therefore I've chosen the constructor to be it's abbreviation "*Ite*". Conditionals are formed out of three parts: a condition, a true case, and a false case. In our language these will be three expressions; the first being the condition, secondly the true case, and thirdly the false case.

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

the semantics of it are

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

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, making our calculation *inductive*.

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

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:

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

To check my calculations, we can do several tests: 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 Huttondescribe in CCC [1, page 14; derivation vs proof]

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 the result of the condition is correct.

The resulting code after applying the *comp* funtion is
PUSH 4 (PUSH 5 (ADD (PUSH 2 (PUSH 3 (ADD (PUSH 1 (ITE HALT)))))))
Which is correct, as given by the following induction:

$$
\begin{aligned}
comp'\ (Ite\ x\ y\ z)\ c &= comp'\ z(comp'\ y\ (comp'\ x\ (ITE\ c))) && (1) \\
comp'\ (Add\ x\ y)\ c &= comp'x(comp'y(ADDc)) && (2) \\
comp'\ (Val\ n)\ c &= PUSH\ n\ c && (3)
\end{aligned}
$$

$comp'(Ite\ (Val\ 1)(Add\ (Val\ 2)(Val\ 3))(Add\ (Val\ 4)(Val\ 5)))$
$= \{equation\ 1\}$
$comp'(Add\ (Val\ 4)(Val\ 5))(comp'(Add\ (Val\ 2)(Val\ 3))(comp'\ (Val\ 1)\ (ITE\ c)))$
$= \{equation\ 2,\ twice\}$
$comp'(Val\ 4)(comp'(Val\ 5)(ADD(comp'(Val\ 2)(comp'(Val\ 3)(ADD(comp'\ (Val\ 1)\ (ITE\ c)))))))$
$= \{equation\ 3,\ 5\ times\}$
$PUSH\ 4(PUSH\ 5(ADD(PUSH2(PUSH3(ADD(PUSH\ 1\ (ITE\ c)))))))$

In executing this code with an empty stack, we get: [5]
By hand, this calculation is as follows:

$$\begin{aligned} exec\ (PUSH\ n\ c)s &= exec\ c\ (n:s) && (4)\\ exec\ (ADD\ c)\ s &= exec\ c\ ((n+m):s) && (5)\\ exec\ (ITE\ c)\ (k:m:n:s) &= execc((if\ k\ \neq 0\ then\ m\ else\ n):s) && (6) \end{aligned}$$

$exec\ (PUSH\ 4(PUSH\ 5(ADD(PUSH2(PUSH3(ADD(PUSH\ 1\ (ITE\ c)))))))) []$
$= \{equation\ 4, twice\}$
$exec\ (ADD(PUSH2(PUSH3(ADD(PUSH\ 1\ (ITE\ c))))))\ [5,4]$
$= \{equation\ 5\}$
$exec\ (PUSH2(PUSH3(ADD(PUSH\ 1\ (ITE\ c)))))\ [9]$
$= \{equation\ 4, twice\}$
$exec\ (ADD(PUSH\ 1\ (ITE\ c)))\ [3,2,9]$
$= \{equation\ 5\}$
$exec\ (PUSH\ 1\ (ITE\ c))\ [5,9]$
$= \{equation\ 4\}$
$exec\ (ITE\ c)\ [1,5,9]$
$= \{equation\ 6\}$
$[5]$

Applying the interepreter to the expresssion[3]

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

As the virtual machineand interpreter are in agreement, we can show the proof of the definition meeting the specification. i.e we show $exec\ c\ (eval\ (Ite\ x\ y\ z) : s) = exec(comp'\ (Ite\ x\ y\ z)c)\ s$

[3]Recall that the interpreter returns and Int rather than a stack

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

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 we don't want to execute.

3.1 Lazy evaluation

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

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

and it's semantics are

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

which are the same as the semantics of an Ite expression because there's no code to be lazily evaluated at the interpreter level.

This inductive calculation begins in the same way

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

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.

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

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

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 coninuation code c

$$\begin{aligned} ct &= comp'yc \\ ce &= comp'zc \end{aligned}$$

In summary our generalised formal partial specfication is

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

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 porblem 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 resst 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.

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:

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

and exec: [5]

Again, we can check this by hand

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.