Calculating compilers with the Bahr and Hutton method

Marco Jones

1 A summary of Calculating Correct Compilers (Bahr and Hutton , 2014)

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, whilst simultaneously guanteeing it's correctness by virtue of *constructive induction [1].*

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 other than the examples given in Calculating Correct Compilers (CCC).

Firstly this paper will start 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]

Thirdly, define a new source language. Finally I will conclude with comments on space and time complexity of the implementation, and further work.

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 [2][2.5 Combining the transformation steps], resulting in a much simpler 3 step process. Thus we will use the refined method.

Bahr and Hutton begin by defining a datatype *Expr*, which represents the syntax of the source language, a function which defines their semantics, and a

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

Stack "which corresponds to a stack of integers" [2][section 2.5, pg 9].

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

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

These equations describe a function *eval* which takes a single input, an expression of type $Expr$, and outputs an integer of type Int.

For comparison, this way of defining semantics is similar to key words programming languages, in that they are compiled into machine code and executed by the virtual machine, to create a specific output. E.g *eval* in our case, has a similar effect to "return" in Python.

Then they *derived* four additional components and two correctness equations [2][page 9]:

- A datatype $Code$ that represents code for the virtual machine.
- A function $comp :: Expr \to Code$ that compiles expressions to code.
- A function $comp' :: Expr \to Code \to Code$ that also takes a continuation.
- A function $exec :: Code \to Stack \to Stack$ that provides a semantics for code.

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

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

Note: this segment was from CCC, *specifications* 3 and 4 are numbered as such, and so will any other equations taken from CCC.

These equations capture the relationships between the semantics, compiler and virtual machine[2][page 9], because *comp*turns an expression into code, and *exec*turns code together with a stack into another stack; to reiterrate, this can be interpreted intuitively as *'the relation between execution of compiled code and the semantics of that code is defined by equation 3'*

"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." [2].

It is perhaps strange in appearance but this axiom 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 tokens and it is up to a parser to collect them together *comp*as we will see, interprets the source code and builds up a list of instructions of type *code*

expression e, and a stack s, as it's input, and output s', where $s' = e' : s$ where e' is the result of evaluating the expression and $: s$ means that e' is pushed to the top of s. [2] From this *partial specification* [3]

$$\begin{aligned} evals \quad & :: Expr \to Stack \to Stack \\ & evals(Val\, n)s \\ = \quad & \{\text{specification(1)}\} \\ & eval(Val\, n) : s \\ = \quad & \{\text{definition of eval}\} \\ & n : s \\ = \quad & \{\text{define } push_S\, n\, s == n : s\} \\ & push_S\, n\, s \end{aligned}$$

From this calculation we can conclude that $evals(Val\, n)s = push_S\, n\, s$
Next the inductive case, $Add(xy)$

$$\begin{aligned} & evals(add\, x\, y)s \\ = \quad & \{\text{definition of } eval_S\} \\ & eval(add\, x\, y) : s \\ = \quad & \{\text{define: } add_S\, n : m : s = (eval\, m + eval\, n) : s\} \\ & (eval\, x + eval\, y) : s \\ = \quad & \{\text{induction hypothesis for } y\} \\ & add_S(evals\, y : (eval\, x : s)) \\ = \quad & \{\text{induction hypothesis for } x\} \\ & add_S(evals\, y : (evals\, x\, s)) \end{aligned}$$

$add_S\, n : m : s == (eval\, m + eval\, n) : s$ Note how evaluation order has been maintained, allbeit subtely. x is pushed onto the stack first as x corresponds to m in the above equation, the value of $evalm + evaln$ is unaffected because

[2] ':' (read as 'cons') in Haskell, is the list constructor operator, meaning our stack will be implemented as a list with values being appeneded to, and removed from the head of the list.

[3] Partial specifications avoid the need to "predetermine implementation decisions", furthermore they give us freedom in our calculations to define auxillary functions

addition is commutative, however this is an important point that left to right pushes of arguments, corresponds to left to right evaluation for non-commutative operators. This is demonstrates how design decisions are still apparent even though the calculation process is inductive, because we can deifne our own funcitons to meet our requirements. Later we will see how our calculations hit dead ends unless we define new operations.

In conclusion, Bahr and Hutton have calculated the following definitions:

$$eval_C :: Expr \to Cont \to Cont$$

such that:

Specification 1

$$eval_C\, x\, c\, s = c\ (evalsx\ s)$$

The base case is very simple here. Recalling the definition we found in the previous section: $evals(Val\, n)s = push_S\, n\, s$

$$\begin{array}{ll} cl & eval_C(Addxy)cs \\ = & \{\text{specification 2}\} \\ & c(evals(add_S)s) \\ = & \{\text{definition of } evals\} \\ & c(add_S(evalsy(evalsx\ s))) \\ = & \{\text{function composition}\} \\ = & eval_C\, x(eval_C\, y(c\, o\, add_S))s \end{array}$$

3 Extending the compilers

From now on this dissertation will be about using the Bahr and Hutton method to calculate a compiler with definitions not defined in CCC.

To begin with 4.1

3.1 Conditionals

References

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

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