Under consideration for publication in the Journal of Functional Programming

FUNCTIONAL PEARL
Compiling a Fifty Year Journey

GRAHAM HUTTON
School of Computer Science, University of Nottingham, UK
(email: gmh@cs.nott.ac.uk*)*

PATRICK BAHR
Department of Computer Science, IT University of Copenhagen, Denmark
(email: paba@itu.dk*)*

Abstract

Fifty years ago, John McCarthy and James Painter (1967) published the first paper on compiler verification, in which they showed how to formally prove the correctness of a compiler that translates arithmetic expressions into code for a register-based machine. In this article, we revisit this example in a modern context, and show how such a compiler can now be calculated directly from a specification of its correctness using simple equational reasoning techniques.

1 Preliminaries

The source language for our compiler comprises arithmetic expressions that are built up from integer values using an addition operator, whose semantics is given by a recursive function that evaluates an expression to its integer value:

$$\begin{array}{l}\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}\\ \mathit{eval} \quad :: \mathit{Expr} \to \mathit{Int}\\ \mathit{eval}\ (\mathit{Val}\ n) \quad = n\\ \mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y\end{array}$$

In turn, the target for our compiler is a virtual machine with an infinite sequence of registers that can be used to store intermediate results, each of which is identified by a unique name and can store a single integer value. The current value of each register is contained in the memory of the machine. For the purposes of simplifying our calculations, we use abstract types for both register names and the memory:

$$\begin{array}{l}\textbf{type}\ \mathit{Reg} \ = \cdots\\ \textbf{type}\ \mathit{Mem} = \cdots\end{array}$$

We further assume that the machine has a special register called the accumulator, which is used to store the result of evaluating an expression. Rather than using a specific register in

the memory for this purpose, we factor out the accumulator as a component of the machine configuration, which comprises the current accumulator value and memory:

type $Conf = (Int, Mem)$

Given the assumptions above, our goal now is to calculate three additional components that together complete the definition of a compiler for expressions:

- A datatype *Code* that represents code for the virtual machine;
- A function $compile :: Expr \to Code$ that compiles expressions to code;
- A function $exec :: Code \to Conf \to Conf$ that provides a semantics for code.

Intuitively, *Code* will comprise a suitable set of primitive machine operations for evaluating expressions, *compile* will translate an expression into a sequence of such operations, and *exec* will execute code starting from an initial configuration of the machine to give a final configuration. Moreover, the desired relationship between the source language, compiler and virtual machine is captured by the following correctness property:

$$exec\ (compile\ e)\ (a, empty) \quad = \quad (eval\ e, empty) \tag{1}$$

That is, compiling an expression and executing the resulting code starting with any accumulator value and an empty memory gives a final configuration in which the accumulator contains the value of the expression and the memory is empty. In practice, the fact that the memory must be returned to an empty state in the final configuration means that if the compiled code for an expression uses any registers to store intermediate results, it must free them up afterwards to ensure that the memory is returned to its original state. Once again, the rationale for this decision is that it simplifies our calculations.

Equation (1) captures the correctness of the compiler, but on its own isn't suitable as a basis for calculating the three undefined components. In particular, our methodology is based on the use of induction, and as is often the case we first need to generalise the property we are considering. To this end, we generalise equation (1) from the empty memory to an arbitrary memory whose registers are free from a given point onwards. In turn, we generalise the compiler to take two extra arguments: the name of the first free register that can be used, and additional code to be executed after the compiled code. Using these ideas, the correctness of a generalised compilation function

$$comp :: Expr \to Reg \to Code \to Code$$

can then be specified by the following implication:

$$freeFrom\ r\ m \quad \Rightarrow \quad exec\ (comp\ e\ r\ c)\ (a, m) \ = \ exec\ c\ (eval\ e, m) \tag{2}$$

That is, if the memory is assumed to be free from a given register onwards, then compiling an expression and executing the resulting code starting from a given configuration gives the same result as executing the additional code in a configuration with the value of the expression in the accumulator and the same initial memory.

In summary, (1) and (2) provide specifications for the undefined components, and our goal is to calculate definitions that satisfy these properties. Given that the specifications have four unknowns (*Code*, *compile*, *exec* and *comp*), this may seem like an impossible

task. However, as we shall see, it can be achieved using simple equational reasoning. All of our calculations have been mechanically checked using the Coq proof assistant.

2 Memory model

Before proceeding to the calculation itself, we need to formalise our assumptions about the abstract types *Reg* for register names and *Mem* for the memory that we introduced. First of all, we assume the following primitive operations on these types:

$$\begin{array}{ll} \mathit{empty} & :: \mathit{Mem} \\ \mathit{set} & :: \mathit{Reg} \to \mathit{Int} \to \mathit{Mem} \to \mathit{Mem} \\ \mathit{get} & :: \mathit{Reg} \to \mathit{Mem} \to \mathit{Int} \\ \mathit{first} & :: \mathit{Reg} \\ \mathit{next} & :: \mathit{Reg} \to \mathit{Reg} \\ \mathit{free} & :: \mathit{Reg} \to \mathit{Mem} \to \mathit{Mem} \end{array}$$

Intuitively, *empty* is the initial memory in which all registers are empty (contain no value), while *set* and *get* respectively change and fetch the integer value of a given register in the memory. In turn, *first* is the name of the first register, *next* gives the name of the next register, and *free* makes a register empty (contain no value). Note that *freeFrom* is not included in the above list, as it is a meta-level predicate for reasoning purposes rather than being an operation on the memory of the virtual machine.

A simple way to realise the above memory model is to represent a register name as a natural number, the memory as a function from register names to their current value, and use a special value to represent a register that is empty. For the purposes of our calculations, however, we only require the following properties of the primitive operations, and don't need to be concerned with precisely how they are defined.

$$\begin{array}{rcll} & & \mathit{freeFrom}\ \mathit{first}\ \mathit{empty} & \text{(EMPTY MEMORY)} \\ & & \mathit{get}\ r\ (\mathit{set}\ r\ n\ m) = n & \text{(SET/GET)} \\ \mathit{freeFrom}\ r\ m & \Rightarrow & \mathit{free}\ r\ (\mathit{set}\ r\ n\ m) = m & \text{(SET/FREE)} \\ \mathit{freeFrom}\ r\ m & \Rightarrow & \mathit{freeFrom}\ (\mathit{next}\ r)\ (\mathit{set}\ r\ n\ m) & \text{(SET/FREEFROM)} \end{array}$$

These properties state in turn that: every register from the first onwards is free in the empty memory; setting a register and then getting its value gives the expected result; setting the first free register and then freeing it up returns the memory to its previous state; and finally, setting the first free register leaves all subsequent registers free.

3 Compiler calculation

To calculate the compiler we proceed directly from specification (2) by structural induction on the expression argument e, using the desire to apply the induction hypotheses as the driving force for the calculation process. In each case, we aim to rewrite the left-hand side $\mathit{exec}\ (\mathit{comp}\ e\ r\ c)\ (a,m)$ of the equation into the form $\mathit{exec}\ c'\ (a,m)$ for some code c', from which we can then conclude that the definition $\mathit{comp}\ e\ r\ c = c'$ satisfies the specification in

this case. In order to do this, we will find that we need to introduce new constructors into the *Code* type, along with their interpretation by the function *exec*.

In the base case, $e = Val\ n$, we assume *freeFrom r m*, and calculate as follows:

$$\begin{array}{ll} & exec\ (comp\ (Val\ n)\ r\ c)\ (a,m) \\ = & \quad \{\ \text{specification (2)}\ \} \\ & exec\ c\ (eval\ (Val\ n),m) \\ = & \quad \{\ \text{applying } eval\ \} \\ & exec\ c\ (n,m) \end{array}$$

Now we appear to be stuck, as no further definitions can be applied. However, we are aiming to end up with a term of the form $exec\ c'\ (a,m)$ for some code c'. That is, in order to complete the calculation we need to solve the equation:

$$exec\ c'\ (a,m) \quad = \quad exec\ c\ (n,m)$$

Note that we can't simply use this equation as a definition for *exec*, because n and c would be unbound in the body of the definition. The solution is to package these two variables up in the code argument c' (which can freely be instantiated as it is existentially quantified, whereas the other variables in the equation are universally quantified), by adding a new constructor to the *Code* datatype that takes these two variables as arguments,

$$LOAD :: Int \to Code \to Code$$

and defining a new equation for the function *exec* as follows:

$$exec\ (LOAD\ n\ c)\ (a,m) = exec\ c\ (n,m)$$

That is, executing the code $LOAD\ n\ c$ proceeds by loading the integer n into the accumulator and then executing the code c, hence the choice of name for the new constructor. Using these ideas, it is now straightforward to complete the calculation:

$$\begin{array}{ll} & exec\ c\ (n,m) \\ = & \quad \{\ \text{definition of } exec\ \} \\ & exec\ (LOAD\ n\ c)\ (a,m) \end{array}$$

The final term now has the form $exec\ c'\ (a,m)$ where $c' = LOAD\ n\ c$, from which we conclude that the following definition satisfies specification (2) in the base case:

$$comp\ (Val\ n)\ r\ c = LOAD\ n\ c$$

That is, the code for an integer value simply loads the value into the accumulator and then continues with the extra code that is supplied. Note that for this case we didn't need to use the register argument r or the *freeFrom* assumption.

For the inductive case, $e = Add\ x\ y$, we assume *freeFrom r m* and begin in the same way as above by first applying the specification and the evaluation function:

$$\begin{array}{ll} & exec\ (comp\ (Add\ x\ y)\ r\ c)\ (a,m) \\ = & \quad \{\ \text{specification (2)}\ \} \\ & exec\ c\ (eval\ (Add\ x\ y),m) \\ = & \quad \{\ \text{applying } eval\ \} \\ & exec\ c\ (eval\ x + eval\ y,m) \end{array}$$

At this point no further definitions can be applied. However, as we are performing an inductive calculation, we can make use of the induction hypotheses for the expressions x and y. To use the induction hypothesis for y, that is, $\mathit{freeFrom}\ r'\ m' \Rightarrow \mathit{exec}\ (\mathit{comp}\ y\ r'\ c')\ (a', m') = \mathit{exec}\ c'\ (\mathit{eval}\ y, m')$, we must satisfy the precondition for some register r' and memory m', and rewrite the term being manipulated into the form $\mathit{exec}\ c'\ (\mathit{eval}\ y, m')$ for some code c'. That is, we need to satisfy the precondition $\mathit{freeFrom}\ r'\ m'$ and solve the equation:

$$\mathit{exec}\ c'\ (\mathit{eval}\ y, m') \quad = \quad \mathit{exec}\ c\ (\mathit{eval}\ x + \mathit{eval}\ y, m)$$

We are free to instantiate r', m' and c' in order to achieve these goals. First of all, we generalise from the specific values $\mathit{eval}\ x$ and $\mathit{eval}\ y$ in the equation to give:

$$\mathit{exec}\ c'\ (a, m') \quad = \quad \mathit{exec}\ c\ (b + a, m)$$

We can't use this equation as a definition for *exec*, because the variables c, b and m would be unbound in the body of the definition. However, we are free to instantiate c' and m' in order to solve the equation. We consider each unbound variable in turn:

- For c, the simplest option is to put it into the argument c' as we did in the base case, by adding a new constructor. If we attempted to put c into m', this would require storing code in the memory, which is not supported by our memory model.
- For b, the simplest option is to put it into the memory m', by assuming that it is stored in a register. If we attempted to put b into the code c', this would require evaluating the argument expression x at compile-time to produce this value, whereas for a compiler we normally expect evaluation to take place at run-time.
- For m, the simplest option is also to put it into m', by assuming that m can be derived from m' in some way. If we attempted to put m into c', this would require storing the entire memory in the code, which is not what we expect from a compiler.

How should we satisfy the above requirements for m and b? One might try simply equating the memories and assuming the value is stored in the first free register, that is take $m' = m$ and assume $\mathit{get}\ r\ m = b$. However, the latter assumption is not satisfiable as it conflicts with our top-level assumption $\mathit{freeFrom}\ r\ m$ that all registers from r onwards are free in m.

The simplest way to resolve this problem is to equate the two memories for all registers *except* register r, which in the case of m' should contain the value b to satisfy our requirement, and in the case of m should be free to satisfy our top-level assumption. The desired relationship between the two memories can then be captured by two assumptions, $\mathit{get}\ r\ m' = b$ and $\mathit{free}\ r\ m' = m$, using which the equation to be solved now has the form:

$$\mathit{exec}\ c'\ (a, m') \quad = \quad \mathit{exec}\ c\ (\mathit{get}\ r\ m' + a, \mathit{free}\ r\ m')$$

The variable r is now unbound on the right-hand side of the equation, but can readily be packaged up along with the variable c in the code argument c' by adding a new constructor to the *Code* datatype that takes these two variables as arguments,

$$\mathit{ADD} :: \mathit{Reg} \to \mathit{Code} \to \mathit{Code}$$

and defining a new equation for the function *exec*:

$$\mathit{exec}\ (\mathit{ADD}\ r\ c)\ (a, m) = \mathit{exec}\ c\ (\mathit{get}\ r\ m + a, \mathit{free}\ r\ m)$$

That is, executing the code *ADD r c* proceeds by adding the value of register r to the accumulator and then freeing up this register in the memory, hence the choice of name for the new constructor. Using our three local assumptions

(a) $\mathit{freeFrom}\ r'\ m'$

(b) $\mathit{get}\ r\ m' = \mathit{eval}\ x$

(c) $\mathit{free}\ r\ m' = m$

we then continue the calculation as follows:

$$
\begin{array}{ll}
 & \mathit{exec}\ c\ (\mathit{eval}\ x + \mathit{eval}\ y, m) \\
= & \quad \{\ \text{assumptions (b) and (c)}\ \} \\
 & \mathit{exec}\ c\ (\mathit{get}\ r\ m' + \mathit{eval}\ y,\ \mathit{free}\ r\ m') \\
= & \quad \{\ \text{definition of } \mathit{exec}\ \} \\
 & \mathit{exec}\ (\mathit{ADD}\ r\ c)\ (\mathit{eval}\ y, m') \\
= & \quad \{\ \text{induction hypothesis for } y\text{, assumption (a)}\ \} \\
 & \mathit{exec}\ (\mathit{comp}\ y\ r'\ (\mathit{ADD}\ r\ c))\ (a', m')
\end{array}
$$

We are now free to chose m' and r' to satisfy the assumptions (a), (b) and (c). The simplest approach is to define m' by setting register r in memory m to the value $\mathit{eval}\ x$, and define r' as the next free register after r. That is, we take $m' = \mathit{set}\ r\ (\mathit{eval}\ x)\ m$ and $r' = \mathit{next}\ r$. It is then easy to verify that these assignments discharge the assumptions:

(a):

$$
\begin{array}{ll}
 & \mathit{freeFrom}\ r'\ m' \\
\Leftrightarrow & \quad \{\ \text{applying } r' \text{ and } m'\ \} \\
 & \mathit{freeFrom}\ (\mathit{next}\ r)\ (\mathit{set}\ r\ (\mathit{eval}\ x)\ m) \\
\Leftrightarrow & \quad \{\ \textsc{set/freeFrom}\ \text{property}, \mathit{freeFrom}\ r\ m\ \} \\
 & \mathit{True}
\end{array}
$$

(b):

$$
\begin{array}{ll}
 & \mathit{get}\ r\ m' \\
= & \quad \{\ \text{applying } m'\ \} \\
 & \mathit{get}\ r\ (\mathit{set}\ r\ (\mathit{eval}\ x)\ m) \\
= & \quad \{\ \textsc{set/get}\ \text{property}\ \} \\
 & \mathit{eval}\ x
\end{array}
$$

(c):

$$
\begin{array}{ll}
 & \mathit{free}\ r\ m' \\
= & \quad \{\ \text{applying } m'\ \} \\
 & \mathit{free}\ r\ (\mathit{set}\ r\ (\mathit{eval}\ x)\ m) \\
= & \quad \{\ \textsc{set/free}\ \text{property}, \mathit{freeFrom}\ r\ m\ \} \\
 & m
\end{array}
$$

In summary, using the assignments for m' and r' that we have determined above, the term that we are manipulating now has the following form:

$$\mathit{exec}\ (\mathit{comp}\ y\ (\mathit{next}\ r)\ (\mathit{ADD}\ r\ c))\ (a',\ \mathit{set}\ r\ (\mathit{eval}\ x)\ m)$$

We are free to chose the new accumulator value a' at this point. With a view to now applying the induction hypothesis for x, which requires that the accumulator contains *eval x*, we simply take $a' = eval\ x$, resulting in the following term:

$$exec\ (comp\ y\ (next\ r)\ (ADD\ r\ c))\ (eval\ x,\ set\ r\ (eval\ x)\ m)$$

We could now apply the induction hypothesis for x, because the memory satisfies the precondition for the register *next r*. However, doing so would yield the term

$$exec\ (comp\ x\ (next\ r)\ (comp\ y\ (next\ r)\ (ADD\ r\ c)))\ (a,\ set\ r\ (eval\ x)\ m)$$

which cannot be rewritten into the desired form $exec\ c'\ (a,m)$, as there is no way of retrieving the value *eval x* from the machine configuration. Therefore, we have to transform the configuration from $(eval\ x, set\ r\ (eval\ x)\ m)$ into $(eval\ x, m)$ before applying the induction hypothesis. That is, we need to solve the equation

$$exec\ c'\ (eval\ x,m) \quad = \quad exec\ (comp\ y\ (next\ r)\ (ADD\ r\ c))\ (eval\ x,\ set\ r\ (eval\ x)\ m)$$

As with the case for y, we first generalise the equation, in this case by abstracting over the value *eval x* and the code $comp\ y\ (next\ r)\ (ADD\ r\ c)$, to give the following:

$$exec\ c'\ (a,m) \quad = \quad exec\ c\ (a,\ set\ r\ a\ m)$$

We can't use this equation as a definition for *exec*, because the variables c and r would be unbound in the body of the definition. However, we are free to instantiate c' in order to solve the equation. As previously we can package r and c up in the code argument c' by adding a new constructor to the *Code* datatype,

$$STORE :: Reg \to Code \to Code$$

and defining a new equation for the function *exec*:

$$exec\ (STORE\ r\ c)\ (a,m) = exec\ c\ (a,\ set\ r\ a\ m)$$

That is, executing the code *STORE r c* proceeds by storing the accumulator value in register r, hence the choice of name for the new constructor. We then continue the calculation:

$$\begin{array}{ll} & exec\ (comp\ y\ (next\ r)\ (ADD\ r\ c))\ (eval\ x,\ set\ r\ (eval\ x)\ m) \\ = & \quad \{\ \text{definition of } exec\ \} \\ & exec\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c)))\ (eval\ x, m) \\ = & \quad \{\ \text{induction hypothesis for } x, \text{ assuming } freeFrom\ r'\ m\ \} \\ & exec\ (comp\ x\ r'\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c))))\ (a', m) \end{array}$$

We are now free to choose the register r' and the new accumulator value a' to satisfy the inductive assumption $freeFrom\ r'\ m$. The simplest approach is just to take $r' = r$ and $a' = a$, under which the inductive assumption reduces to our top-level assumption *freeFrom r m* and the machine configuration has the desired form (a,m). The resulting term

$$exec\ (comp\ x\ r\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c))))\ (a,m)$$

now has the form $exec\ c'\ (a,m)$ for some code c', from which we conclude that the following definition satisfies specification (2) in the inductive case:

$$comp\ (Add\ x\ y)\ r\ c = comp\ x\ r\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c)))$$

That is, the code for addition first computes the value of x and stores the resulting value in the first free register r, and then computes the value of the expression y and adds the resulting value to the contents of register r. Note that when compiling y the next free register becomes *next r*, because r itself is used to store the value of x.

Finally, we consider the top-level function $compile :: Expr \to Code$, whose correctness was specified by equation (1). In a similar manner to (2), we aim to rewrite the left-hand side $exec\ (compile\ e)\ (a, empty)$ into the form $exec\ c\ (a, empty)$ for some code c, from which we can then conclude that the definition $compile\ e = c$ satisfies the specification. In this case there is no need to use induction as simple calculation suffices, during which we introduce a new constructor $HALT :: Code$ to transform the term being manipulated into the required form so that specification (2) can then be applied.

$$
\begin{array}{ll}
 & exec\ (compile\ e)\ (a, empty) \\
= & \quad \{ \text{ specification (1) } \} \\
 & (eval\ e, empty) \\
= & \quad \{ \text{ define: } exec\ HALT\ (a,m) = (a,m) \} \\
 & exec\ HALT\ (eval\ e, empty) \\
= & \quad \{ \text{ specification (2), EMPTY MEMORY property } \} \\
 & exec\ (comp\ e\ first\ HALT)\ (a, empty)
\end{array}
$$

In summary, we have calculated the following definitions:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{data}\ Code = LOAD\ Int\ Code \mid STORE\ Reg\ Code \mid ADD\ Reg\ Code \mid HALT} \\
compile & :: Expr \to Code \\
compile\ e & = comp\ e\ first\ HALT \\
comp & :: Expr \to Reg \to Code \to Code \\
comp\ (Val\ n)\ r\ c & = LOAD\ n\ c \\
comp\ (Add\ x\ y)\ r\ c & = comp\ x\ r\ (STORE\ r\ (comp\ y\ (next\ r)\ (ADD\ r\ c))) \\
exec & :: Code \to Conf \to Conf \\
exec\ (LOAD\ n\ c)\ (a,m) & = exec\ c\ (n,m) \\
exec\ (STORE\ r\ c)\ (a,m) & = exec\ c\ (a,\ set\ r\ a\ m) \\
exec\ (ADD\ r\ c)\ \ (a,m) & = exec\ c\ (get\ r\ m + a,\ free\ r\ m) \\
exec\ HALT\ \ \ \ \ \ (a,m) & = (a,m)
\end{array}
$$

This compiler is essentially the same as McCarthy and Painter's (1967) except that i) our compiler has been calculated directly from a high-level specification of its correctness, with all the above compilation machinery falling naturally out of the calculation process; and ii) their source language also includes variables, which haven't been considered here for simplicity, but don't pose any difficulties for our methodology.

4 Reflection

The original compiler correctness proof of McCarthy and Painter (1967) is rather complex, using many lemmas. A methodology that can be used for calculating a compiler has to be simpler, otherwise one becomes lost in the technical details. Three main ideas form the foundation of the simplified methodology that made our calculation possible: using

pattern matching rather than deconstructors, strengthening the induction hypothesis using additional code, and using a specification that requires freeing up unused registers.

The use of pattern matching improves the clarity of the reasoning by virtue of providing a more compact notation than deconstructors. But pattern matching is also indispensable for the methodology for calculating the virtual machine. As part of the calculation we need to solve equations in order to make progress, and solving such an equation directly yields a case for the definition of the virtual machine by pattern matching.

The desire to apply induction hypotheses is the main driving force in our calculation process. Strengthening the induction hypothesis by introducing an additional code argument to the compiler means that the induction hypothesis becomes directly applicable without the need for additional lemmas (Hutton, 2016, chapter 16). The need for such lemmas would obscure the goal to which the calculation is targeted. Instead, the more general induction hypothesis allows us to make progress in the calculation by solving simple equations.

McCarthy and Painter use reasoning modulo unused registers. Rather than using normal equality, they reason with an equivalence relation $=_r$ on the memory of their machine, under which $m =_r m'$ when m and m' coincide on all registers prior to r. This relaxed equality allows the compiler to use registers from r onwards to store intermediate results. However, this approach makes reasoning with additional code more difficult. In particular, the specification for *comp* then becomes $exec\ (comp\ e\ r\ c)\ (a,m) = exec\ c\ (eval\ e, m')$, where the final memory m' is existentially quantified with the side condition that $m =_r m'$. Calculations with existentially quantified variables that are subject to side conditions are difficult to manage and prone to errors. Instead, our approach is to demand the equality of m and m'. As a consequence, the virtual machine has to 'clean up after itself' by using *free* to restore memory that has been used to store intermediate results.

One might think that this requirement may result in a less efficient implementation. However, all uses of *free* can safely be removed from the virtual machine. Or equivalently, we can instantiate the memory model with an implementation where *free r* is the identity function. We only use *free* to impose structure on the memory to simplify the reasoning. This structure, in the form of the *freeFrom* predicate, is not used in the virtual machine itself. Importantly, however, having a virtual machine that cleans up after itself allowed us to adapt the compiler calculation methodology that we developed for stack-based machines (Bahr & Hutton, 2015), which relies on the intrinsic structure of the stack.

Acknowledgements

Graham Hutton was funded by EPSRC grant EP/P00587X/1, *Mind the Gap: Unified Reasoning About Program Correctness and Efficiency*.

References

Bahr, Patrick, & Hutton, Graham. (2015). Calculating Correct Compilers. *Journal of Functional Programming*, **25**(Sept.).

Hutton, Graham. (2016). *Programming in Haskell*. Cambridge University Press.

McCarthy, John, & Painter, James. (1967). Correctness of a Compiler for Arithmetic Expressions. *Pages 33–41 of: Mathematical Aspects of Computer Science*. Proceedings of Symposia in Applied Mathematics, vol. 19. American Mathematical Society.