Functional Pearl

Cutting Out Continuations

Graham Hutton[1] and Patrick Bahr[2]

[1] University of Nottingham, UK
[2] IT University of Copenhagen, Denmark

Abstract. In the field of program transformation, one often transforms programs into continuation-passing style to make their flow of control explicit, and then immediately removes the resulting continuations using defunctionalisation to make the programs first-order. In this article, we show how these two transformations can be fused together into a single transformation step that cuts out the need to first introduce and then eliminate continuations. Our approach is calculational, uses standard equational reasoning techniques, and is widely applicable.

1 Introduction

In his seminal work on definitional interpreters for higher-order programming languages, Reynolds [12] popularised two techniques that have become central to the field of program transformation: i) transforming programs into *continuation-passing style* (CPS) to make their flow of control explicit; and ii) transforming higher-order programs into first-order style using *defunctionalisation*. These two transformations are often applied together in sequence, for example to transform high-level denotational semantics for programming languages into low-level implementations such as abstract machines or compilers [12,13,11,1,2].

However, there is something rather unsatisfactory about applying these two transformations in sequence: the CPS transformation introduces continuations, only for these to immediately be removed by defunctionalisation. This seems like an ideal opportunity for applying *fusion*, which in general aims to combine a series of transformations into a single transformation that achieves the same result, but without the overhead of using intermediate structures between each step. In our case, the intermediate structures are continuations.

Despite CPS transformation and defunctionalisation being obvious candidates for being fused into a single transformation, this does not appear to have been considered before. In this article, we show how this can be achieved in a straightforward manner using an approach developed in our recent work on calculating compilers [5]. Our approach starts with a specification that captures the intended behaviour of the desired output program in terms of the input program. We then calculate definitions that satisfy the specification by *constructive induction* [4], using the desire to apply induction hypotheses as the driving force for the calculation process. The resulting calculations only require

standard equational reasoning techniques, and achieve the combined effect of CPS and defunctionalisation without the intermediate use of continuations.

We illustrate our approach by means of two examples: a minimal language of arithmetic expressions to introduce the basic ideas in a simple manner, and a call-by-value lambda calculus to show how it can be applied to a language with variable binding. More generally, the approach scales up to a wide range of programming language features and their combination, including exceptions, state, loops, non-determinism, interrupts, and different orders of evaluation.

The article is aimed at a general functional programming audience rather than specialists, and all the programs and calculations are written in Haskell. The calculations have also been mechanically verified using the Coq proof assistant, and the proof scripts for our two examples, together with a range of other applications, are freely available online via GitHub [7].

2 Arithmetic Expressions

Consider a simple language of arithmetic expressions built up from integers and addition, whose syntax and denotational semantics are defined as follows:

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

Now suppose that we wish to calculate an abstract machine for this language, which is given by a set of first-order, tail-recursive functions that are together semantically equivalent to the function *eval*. We initially perform this calculation using the standard two step approach [1]: transformation into continuation-passing style, followed by defunctionalisation. Then in section 3 we show how these two transformation steps can be combined into a single step.

Step 1 - Add Continuations

The first step is to transform the evaluation function into continuation-passing style. More specifically, we seek to derive a more general evaluation function

$$\mathit{eval}' :: \mathit{Expr} \to (\mathit{Int} \to \mathit{Int}) \to \mathit{Int}$$

that takes a *continuation* of type $\mathit{Int} \to \mathit{Int}$ as additional argument, which is applied to the result of evaluating the expression. More precisely, the desired behaviour of eval' is specified by the following equation:

$$\mathit{eval}'\ x\ c = c\ (\mathit{eval}\ x) \tag{1}$$

Rather than first defining the new function eval' and then separately proving by induction that it satisfies the above equation, we aim to *calculate* a definition

for $eval'$ that satisfies this equation by *constructive induction* on the expression argument x. In each case, we start with the term $eval'\ x\ c$ and gradually transform it by equational reasoning, aiming to arrive at a term t that does not refer to the original semantic function $eval$, such that we can then take $eval'\ x\ c = t$ as a defining equation for $eval'$ in this case. For the base case, when the expression has the form $Val\ n$, the calculation is trivial:

$$
\begin{array}{ll}
 & eval'\ (Val\ n)\ c \\
= & \quad \{ \text{ specification (1) } \} \\
 & c\ (eval\ (Val\ n)) \\
= & \quad \{ \text{ applying } eval \} \\
 & c\ n
\end{array}
$$

By means of this calculation, we have *discovered* the definition for $eval'$ in the base case, namely: $eval'\ (Val\ n)\ c = c\ n$. In the inductive case, when the expression has the form $Add\ x\ y$, we start in the same manner as above:

$$
\begin{array}{ll}
 & eval'\ (Add\ x\ y)\ c \\
= & \quad \{ \text{ specification (1) } \} \\
 & c\ (eval\ (Add\ x\ y)) \\
= & \quad \{ \text{ applying } eval \} \\
 & c\ (eval\ x + eval\ y)
\end{array}
$$

Now we appear to be stuck, as no further definitions can be applied. However, as we are performing an inductive calculation, we can use the induction hypotheses for x and y, that is, $eval'\ x\ c' = c'\ (eval\ x)$ and $eval'\ y\ c'' = c''\ (eval\ y)$. To use these hypotheses, we must rewrite the term being manipulated into the form $c'\ (eval\ x)$ and $c''\ (eval\ y)$ for some continuations c' and c''. This can be readily achieved by abstracting over $eval\ x$ and $eval\ y$ using lambda expressions. Using this idea, the rest of the calculation is then straightforward:

$$
\begin{array}{ll}
 & c\ (eval\ x + eval\ y) \\
= & \quad \{ \text{ abstracting over } eval\ y \} \\
 & (\lambda m \to c\ (eval\ x + m))\ (eval\ y) \\
= & \quad \{ \text{ induction hypothesis for } y \} \\
 & eval'\ y\ (\lambda m \to c\ (eval\ x + m)) \\
= & \quad \{ \text{ abstracting over } eval\ x \} \\
 & (\lambda n \to eval'\ y\ (\lambda m \to c\ (n + m)))\ (eval\ x) \\
= & \quad \{ \text{ induction hypothesis for } x \} \\
 & eval'\ x\ (\lambda n \to eval'\ y\ (\lambda m \to c\ (n + m)))
\end{array}
$$

The same result can also be achieved by applying the induction hypotheses in the opposite order to the above, but for the purposes of the later fused calculation it is important that we apply the induction hypothesis on y first if we want to ensure that the resulting abstract machine evaluates the arguments of addition from left-to-right. In conclusion, we have calculated the following definition:

$$
\begin{array}{lll}
\mathit{eval}' & :: \mathit{Expr} \to (\mathit{Int} \to \mathit{Int}) \to \mathit{Int} \\
\mathit{eval}'\ (\mathit{Val}\ n) & c = c\ n \\
\mathit{eval}'\ (\mathit{Add}\ x\ y)\ c & = \mathit{eval}'\ x\ (\lambda n \to \mathit{eval}'\ y\ (\lambda m \to c\ (n + m)))
\end{array}
$$

Finally, our original evaluation function can now be redefined in terms of the new version by taking the identity function as the continuation in specification (1), which results in the definition $\mathit{eval}\ x = \mathit{eval}'\ x\ (\lambda n \to n)$.

Step 2 - Defunctionalise

The second step is to transform the new evaluation function back into first-order style, using defunctionalisation. The basic idea is to replace the higher-order type $\mathit{Int} \to \mathit{Int}$ of continuations by a first-order datatype whose values represent the specific forms of continuations that we actually need.

Within the definitions for the functions *eval* and eval', there are only three forms of continuations that are used, namely one to terminate the evaluation process, one to continue once the first argument of an addition has been evaluated, and one to add two integer results together. We begin by defining an abbreviation *Cont* for the type of continuations, together with three combinators *halt*, *next* and *add* for constructing the required forms of continuations:

$$
\begin{array}{lll}
\textbf{type}\ \mathit{Cont} & = \mathit{Int} \to \mathit{Int} \\
\mathit{halt} & :: \mathit{Cont} \\
\mathit{halt} & = \lambda n \to n \\
\mathit{next} & :: \mathit{Expr} \to \mathit{Cont} \to \mathit{Cont} \\
\mathit{next}\ y\ c & = \lambda n \to \mathit{eval}'\ y\ (\mathit{add}\ n\ c) \\
\mathit{add} & :: \mathit{Int} \to \mathit{Cont} \to \mathit{Cont} \\
\mathit{add}\ n\ c & = \lambda m \to c\ (n + m)
\end{array}
$$

Using these definitions, our evaluator can now be rewritten as:

$$
\begin{array}{lll}
\mathit{eval} & :: \mathit{Expr} \to \mathit{Int} \\
\mathit{eval}\ x & = \mathit{eval}'\ x\ \mathit{halt} \\
\mathit{eval}' & :: \mathit{Expr} \to \mathit{Cont} \to \mathit{Int} \\
\mathit{eval}'\ (\mathit{Val}\ n) & c = c\ n \\
\mathit{eval}'\ (\mathit{Add}\ x\ y)\ c & = \mathit{eval}'\ x\ (\mathit{next}\ y\ c)
\end{array}
$$

The next stage in the process is to define a first-order datatype whose constructors represent the three combinators:

$$
\begin{array}{l}
\textbf{data}\ \mathit{CONT}\ \textbf{where} \\
\quad \mathit{HALT} :: \mathit{CONT} \\
\quad \mathit{NEXT} :: \mathit{Expr} \to \mathit{CONT} \to \mathit{CONT} \\
\quad \mathit{ADD} :: \mathit{Int} \to \mathit{CONT} \to \mathit{CONT}
\end{array}
$$

Note that the constructors for the new type have the same names and types as the combinators for the *Cont* type, except that all the items are now capitalised.

The fact that values of type *CONT* represent continuations of type *Cont* is formalised by the following denotational semantics:

$$\begin{array}{lll} exec & :: & CONT \to Cont \\ exec\ HALT & = & halt \\ exec\ (NEXT\ y\ c) & = & next\ y\ (exec\ c) \\ exec\ (ADD\ n\ c) & = & add\ n\ (exec\ c) \end{array}$$

In the literature this function is typically called *apply* [12]. The reason for using the name *exec* in our setting will become clear shortly. Using these ideas, our aim now is to derive a new evaluation function

$$eval'' :: Expr \to CONT \to Int$$

that behaves in the same way as our continuation semantics $eval' :: Expr \to Cont \to Int$, except that it uses values of type *CONT* rather than continuations of type *Cont*. The desired behaviour of $eval''$ is specified by the equation:

$$eval''\ x\ c = eval'\ x\ (exec\ c) \tag{2}$$

We can now calculate a definition for $eval''$ from this specification, and in turn new definitions for *exec* and *eval* in terms of the new function $eval''$. The calculations are entirely straightforward, and can be found in Hutton and Wright [8]. The end result is the following set of definitions:

$$\begin{array}{lll} eval'' & :: & Expr \to CONT \to Int \\ eval''\ (Val\ n)\ \ c & = & exec\ c\ n \\ eval''\ (Add\ x\ y)\ c & = & eval''\ x\ (NEXT\ y\ c) \\ \\ exec & :: & CONT \to Int \to Int \\ exec\ HALT\ \ n & = & n \\ exec\ (NEXT\ y\ c)\ n & = & eval''\ y\ (ADD\ n\ c) \\ exec\ (ADD\ n\ c)\ \ m & = & exec\ c\ (n + m) \\ \\ eval & :: & Expr \to Int \\ eval\ x & = & eval''\ x\ HALT \end{array}$$

Together with the new type, these definitions form an abstract machine for evaluating expressions: *CONT* is the type of *control stacks* for the machine, which specify how it should continue after the current evaluation has concluded; $eval''$ evaluates an expression in the context of a control stack; *exec* executes a control stack in the context of an integer argument; and finally, *eval* evaluates an expression by invoking $eval''$ with the empty control stack *HALT*. The fact that the machine operates by means of two mutually recursive functions, $eval''$ and *exec*, reflects the fact that it has two modes of operation, depending on whether it is being driven by the structure of the expression or the control stack.

3 Fusing the Transformation Steps

We now show how the two separate transformation steps that were used to derive the abstract machine for evaluating expressions can be combined into a single

step that avoids the use of continuations. In the previous section, we started off by defining a datatype $Expr$ and a function $eval :: Expr \to Int$ that respectively encode the syntax and semantics for arithmetic expressions. Then in two steps we derived an abstract machine comprising four components:

- A datatype $CONT$ that represents control stacks;
- A function $eval'' :: Expr \to CONT \to Int$ that evaluates expressions;
- A function $exec :: CONT \to Int \to Int$ that executes control stacks;
- A new definition for $eval :: Expr \to Int$ in terms of $eval''$.

By combining specifications (1) and (2), the relationship between the three functions is captured by the following equation:

$$eval''\ x\ c = exec\ c\ (eval\ x) \tag{3}$$

That is, evaluating an expression in the context of a control stack gives the same result as executing the control stack in the context of the value of the expression. The key to combining the CPS and defunctionalisation steps is to use this equation *directly* as a specification for the four additional components, from which we then aim to calculate definitions that satisfy the specification. Given that the equation involves two known definitions ($Expr$ and the original $eval$) and four unknown definitions ($CONT$, $eval''$, $exec$, and the new $eval$), this may seem like an impossible task. However, with the benefit of the experience gained from our earlier calculations, it turns out to be straightforward.

We proceed from specification (3) by constructive induction on the expression x. In each case, we aim to rewrite the term $eval''\ x\ c$ into a term t that does not refer to the original semantics $eval$, such that we can then take $eval''\ x\ c = t$ as a defining equation for $eval''$. In order to do this we will find that we need to introduce new constructors into the $CONT$ type, along with their interpretation by the function $exec$. The base case is once again trivial:

$$\begin{array}{ll} & eval''\ (Val\ n)\ c \\ = & \quad \{\ \text{specification (3)}\ \} \\ & exec\ c\ (eval\ (Val\ n)) \\ = & \quad \{\ \text{applying } eval\ \} \\ & exec\ c\ n \end{array}$$

The inductive case begins in the same way:

$$\begin{array}{ll} & eval''\ (Add\ x\ y)\ c \\ = & \quad \{\ \text{specification (3)}\ \} \\ & exec\ c\ (eval\ (Add\ x\ y)) \\ = & \quad \{\ \text{applying } eval\ \} \\ & exec\ c\ (eval\ x + eval\ y) \end{array}$$

At this point no further definitions can be applied. However, we can make use of the induction hypotheses for the argument expressions x and y. In order to use the induction hypothesis for y, that is, $eval''\ y\ c' = exec\ c'\ (eval\ y)$, we

must rewrite the term that is being manipulated into the form $exec\ c'\ (eval\ y)$ for some control stack c'. That is, we need to solve the equation:

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

First of all, we generalise $eval\ x$ and $eval\ y$ to give:

$$exec\ c'\ m = exec\ c\ (n + m)$$

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 control stack argument c' by adding a new constructor to the $CONT$ type that takes these two variables as arguments,

$$ADD :: Int \to CONT \to CONT$$

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

$$exec\ (ADD\ n\ c)\ m = exec\ c\ (n + m)$$

That is, executing the command $ADD\ n\ c$ in the context of an integer argument m proceeds by adding the two integers and then executing the remaining commands in the control stack c, hence the choice of the name for the new constructor. Using these ideas, we continue the calculation:

$$\begin{array}{ll} & exec\ c\ (eval\ x + eval\ y) \\ = & \quad \{\ \text{define: } exec\ (ADD\ n\ c)\ m = exec\ c\ (n + m)\ \} \\ & exec\ (ADD\ (eval\ x)\ c)\ (eval\ y) \\ = & \quad \{\ \text{induction hypothesis for } y\ \} \\ & eval''\ y\ (ADD\ (eval\ x)\ c) \end{array}$$

No further definitions can be applied at this point, so we seek to use the induction hypothesis for x, that is, $eval''\ x\ c' = exec\ c'\ (eval\ x)$. In order to do this, we must rewrite the term $eval''\ y\ (ADD\ (eval\ x)\ c)$ into the form $exec\ c'\ (eval\ x)$ for some control stack c'. That is, we need to solve the equation:

$$exec\ c'\ (eval\ x) = eval''\ y\ (ADD\ (eval\ x)\ c)$$

As with the case for y, we first generalise $eval\ x$ to give

$$exec\ c'\ n = eval''\ y\ (ADD\ n\ c)$$

and then package the free variables y and c up in the argument c' by adding a new constructor to $CONT$ that takes these variables as arguments

$$NEXT :: Expr \to CONT \to CONT$$

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

$$exec\ (NEXT\ y\ c)\ n = eval''\ y\ (ADD\ n\ c)$$

That is, executing the command $NEXT\ y\ c$ in the context of an integer argument n proceeds by evaluating the expression y and then executing the control stack $ADD\ n\ c$. Using this idea, the calculation can now be completed:

$$
\begin{array}{ll}
 & eval''\ y\ (ADD\ (eval\ x)\ c) \\
= & \quad \{ \text{ define: } exec\ (NEXT\ y\ c)\ n = eval''\ y\ (ADD\ n\ c) \ \} \\
 & exec\ (NEXT\ y\ c)\ (eval\ x) \\
= & \quad \{ \text{ induction hypothesis for } x \ \} \\
 & eval''\ x\ (NEXT\ y\ c)
\end{array}
$$

Finally, we conclude the development of the abstract machine by aiming to redefine the original evaluation function $eval :: Expr \to Int$ in terms of the new evaluation function $eval'' :: Expr \to CONT \to Int$. In this case there is no need to use induction as simple calculation suffices, during which we introduce a new constructor $HALT :: CONT$ to transform the term being manipulated into the required form in order that specification (3) can then be applied:

$$
\begin{array}{ll}
 & eval\ x \\
= & \quad \{ \text{ define: } exec\ HALT\ n = n \ \} \\
 & exec\ HALT\ (eval\ x) \\
= & \quad \{ \text{ specification (3) } \} \\
 & eval''\ x\ HALT
\end{array}
$$

In summary, we have calculated the following definitions:

$$
\begin{array}{l}
\textbf{data}\ CONT\ \textbf{where} \\
\quad HALT :: CONT \\
\quad NEXT :: Expr \to CONT \to CONT \\
\quad ADD\ \ :: Int \to CONT \to CONT \\
eval'' :: Expr \to CONT \to Int \\
eval''\ (Val\ n)\ \ c = exec\ c\ n \\
eval''\ (Add\ x\ y)\ c = eval''\ x\ (NEXT\ y\ c) \\
exec :: CONT \to Int \to Int \\
exec\ HALT\ \ n = n \\
exec\ (NEXT\ y\ c)\ n = eval''\ y\ (ADD\ n\ c) \\
exec\ (ADD\ n\ c)\ \ m = exec\ c\ (n + m) \\
eval :: Expr \to Int \\
eval\ x = eval''\ x\ HALT
\end{array}
$$

These are precisely the same definitions as in the previous section, except that they have now been calculated directly from a specification for the correctness of the abstract machine, rather than indirectly using two transformation steps.

In a similar manner to the first calculation step in section 2, we could have reversed the order in which we apply the induction hypotheses in the case of *Add*, which would result in an abstract machine that evaluates the arguments of addition right-to-left rather than left-to-right.

Reflection

The fundamental drawback of the two step approach is that we have to find the right specification for the first step in order for the second step to yield the desired result. This is non-trivial. How would we change the specification for the CPS transformation such that subsequent defunctionalisation yields a compiler rather than an abstract machine? Why did the specification we used yield an abstract machine? By combining the two transformation steps into a single transformation, we avoid this problem altogether: we write one specification that directly relates the old program to the one we want to calculate. As a result, we have an immediate link between the decisions we make during the calculations and the characteristics of the resulting program. Moreover, by avoiding CPS and defunctionalisation, the calculations become conceptually simpler. Because the specification directly relates the input program and the output program, it only requires the concepts and terminology of the domain we are already working on: no continuations or higher-order functions are needed.

4 Lambda Calculus

For our second example, we consider a call-by-value variant of the untyped lambda calculus. To this end, we assume for the sake of simplicity that our meta-language is strict. For the purposes of defining the syntax for the language, we represent variables using de Bruijn indices:

$$\textbf{data}\ \mathit{Expr} = \mathit{Var}\ \mathit{Int} \mid \mathit{Abs}\ \mathit{Expr} \mid \mathit{App}\ \mathit{Expr}\ \mathit{Expr}$$

Informally, $\mathit{Var}\ i$ is the variable with de Bruijn index $i \geqslant 0$, $\mathit{Abs}\ x$ constructs an abstraction over the expression x, and $\mathit{App}\ x\ y$ applies the abstraction that results from evaluating the expression x to the value of the expression y.

In order to define the semantics for the language, we will use an *environment* to interpret the free variables in an expression. Using de Bruijn indices we can represent an environment simply as a list of values, in which the value of variable i is given by indexing into the list at position i:

$$\textbf{type}\ \mathit{Env} = [\,\mathit{Value}\,]$$

In turn, we will use a value domain for the semantics in which functional values are represented as *closures* [10] comprising an expression and an environment that captures the values of its free variables:

$$\textbf{data}\ \mathit{Value} = \mathit{Clo}\ \mathit{Expr}\ \mathit{Env}$$

Using these ideas, it is now straightforward to define an evaluation function that formalises the semantics of lambda expressions:

$$\begin{array}{ll} \mathit{eval} & :: \mathit{Expr} \to \mathit{Env} \to \mathit{Value} \\ \mathit{eval}\ (\mathit{Var}\ i)\ e & = e \mathbin{!!} i \end{array}$$

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Abs}\ x)\ e &&= \mathit{Clo}\ x\ e\\
&\mathit{eval}\ (\mathit{App}\ x\ y)\ e &&= \mathbf{case}\ \mathit{eval}\ x\ e\ \mathbf{of}\\
&&&\qquad \mathit{Clo}\ x'\ e' \to \mathit{eval}\ x'\ (\mathit{eval}\ y\ e : e')
\end{aligned}
$$

Keep in mind that although we use Haskell syntax, we now assume a strict semantics for the meta language. In particular, in the case of $\mathit{App}\ x\ y$ above, this means that the argument expression y is evaluated before x'.

Unlike our first example, however, the above semantics is not compositional. That is, it is not structurally recursive: in the case for $\mathit{App}\ x\ y$ we make a recursive call $\mathit{eval}\ x'$ on the expression x' that results from evaluating the argument expression x. As a consequence, we can no longer use simple structural induction to calculate an abstract machine, but must use the more general technique of *rule induction* [14]. To this end, it is convenient to first reformulate the semantics in an explicit rule-based manner. We define an evaluation *relation* $\Downarrow \subseteq \mathit{Expr} \times \mathit{Env} \times \mathit{Value}$ by means of the following set of inference rules, which are obtained by rewriting the definition of *eval* in relational style:

$$
\frac{e\ !!\ i \ \text{ is defined}}{\mathit{Var}\ i\ \Downarrow_e\ e\ !!\ i} \qquad\qquad \frac{}{\mathit{Abs}\ x\ \Downarrow_e\ \mathit{Clo}\ x\ e}
$$

$$
\frac{x\ \Downarrow_e\ \mathit{Clo}\ x'\ e' \qquad y\ \Downarrow_e\ u \qquad x'\ \Downarrow_{u:e'}\ v}{\mathit{App}\ x\ y\ \Downarrow_e\ v}
$$

Note that *eval* is not a total function because i) evaluation may fail to terminate, and ii) looking up a variable in the environment may fail to return a value. The first cause of partiality is captured by the $\Downarrow$ relation: if $\mathit{eval}\ x\ e$ fails to terminate, then there is no value v with $x\ \Downarrow_e\ v$. The second cause is captured by the side-condition "$e\ !!\ i$ is defined" on the inference rule for $\mathit{Var}\ i$.

Specification

For the purposes of calculating an abstract machine based on the above semantics, the types for the desired new evaluation and execution functions remain essentially the same as those for arithmetic expressions,

$$
\begin{aligned}
&\mathit{eval}'' :: \mathit{Expr} \to \mathit{CONT} \to \mathit{Int}\\
&\mathit{exec}\ \ :: \mathit{CONT} \to \mathit{Int} \to \mathit{Int}
\end{aligned}
$$

except that eval'' now requires an environment to interpret free variables in its expression argument, and the value type is now *Value* rather than *Int*:

$$
\begin{aligned}
&\mathit{eval}'' :: \mathit{Expr} \to \mathit{Env} \to \mathit{CONT} \to \mathit{Value}\\
&\mathit{exec}\ \ :: \mathit{CONT} \to \mathit{Value} \to \mathit{Value}
\end{aligned}
$$

For arithmetic expressions, the desired behaviour of the abstract machine was specified by the equation $\mathit{eval}''\ x\ c = \mathit{exec}\ c\ (\mathit{eval}\ x)$. For lambda expressions,

this equation needs to be modified to take account of the presence of an environment e, and the fact that the semantics is expressed by a relation $\Downarrow$ rather than a function *eval*. The resulting specification is as follows:

$$x \Downarrow_e v \quad \Rightarrow \quad eval'' \; x \; e \; c = exec \; c \; v \tag{4}$$

Note that there is an important qualitative difference between the specification above and equation (3) for arithmetic expressions. The latter expresses soundness and completeness of the abstract machine, whereas the above only covers completeness, that is, every result produced by the semantics is also produced by the machine. But the specification does not rule out that *exec* terminates with a result for an expression that diverges according to the semantics. A separate argument about soundness has to be made. Bahr and Hutton [5] discuss this issue in more detail in the context of calculating compilers.

Calculation

Based upon specification (4), we now calculate definitions for the functions $eval''$ and *exec* by constructive rule induction on the assumption $x \Downarrow_e v$. In each case, we aim to rewrite the left-hand side $eval'' \; x \; e \; c$ of the equation into a term t that does not refer to the evaluation relation $\Downarrow$, from which we can then conclude that the definition $eval'' \; x \; e \; c = t$ satisfies the specification in this case. As in our first example, along the way we will find that we need to introduce new constructors into the *CONT* type, together with their interpretation by *exec*. The base cases for variables and abstractions are trivial.

Assuming $Var \; i \Downarrow_e e \mathbin{!!} i$, we have that

$$\begin{array}{ll} & eval'' \; (Var \; i) \; e \; c \\ = & \quad \{ \text{ specification (4) } \} \\ & exec \; c \; (e \mathbin{!!} i) \end{array}$$

and assuming $Abs \; x \Downarrow_e Clo \; x \; e$, we have that

$$\begin{array}{ll} & eval'' \; (Abs \; x) \; e \; c \\ = & \quad \{ \text{ specification (4) } \} \\ & exec \; c \; (Clo \; x \; e) \end{array}$$

In the case for $App \; x \; y \Downarrow_e v$, we may assume $x \Downarrow_e Clo \; x' \; e'$, $y \Downarrow_e u$ and $x' \Downarrow_{u:e'} v$ by the inference rule that defines the behaviour of $App \; x \; y$, together with induction hypotheses for the three component expressions x, y and x'. The calculation then proceeds by aiming to rewrite the term being manipulated into a form to which these induction hypotheses can be applied. Applying the induction hypothesis for x', i.e. $eval'' \; x' \; (u : e') \; c' = exec \; c' \; v$, is straightforward:

$$\begin{array}{ll} & eval'' \; (App \; x \; y) \; e \; c \\ = & \quad \{ \text{ specification (4) } \} \\ & exec \; c \; v \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ induction hypothesis for } x' \ \} \\
& \mathit{eval}''\ x'\ (u : e')\ c
\end{array}
$$

In turn, to apply the induction hypothesis for y, i.e. $\mathit{eval}''\ y\ e\ c' = \mathit{exec}\ c'\ u$, we need to rewrite the term $\mathit{eval}''\ x'\ (u : e')\ c$ into the form $\mathit{exec}\ c'\ u$ for some control stack c', i.e. we need to solve the equation:

$$\mathit{exec}\ c'\ u = \mathit{eval}''\ x'\ (u : e')\ c$$

As in our first example, the solution is to package the free variables x', e' and c in this equation up in the control stack argument c' by adding a new constructor to the CONT type that takes these variables as arguments

$$\mathit{FUN} :: \mathit{Expr} \to \mathit{Env} \to \mathit{CONT} \to \mathit{CONT}$$

adding a new equation for exec

$$\mathit{exec}\ (\mathit{FUN}\ x'\ e'\ c)\ u = \mathit{eval}''\ x'\ (u : e')\ c$$

and then continuing the calculation:

$$
\begin{array}{ll}
& \mathit{eval}''\ x'\ (u : e')\ c \\
= & \{ \text{ define: } \mathit{exec}\ (\mathit{FUN}\ x'\ e'\ c)\ u = \mathit{eval}''\ x'\ (u : e')\ c \ \} \\
& \mathit{exec}\ (\mathit{FUN}\ x'\ e'\ c)\ u \\
= & \{ \text{ induction hypothesis for } y \ \} \\
& \mathit{eval}''\ y\ e\ (\mathit{FUN}\ x'\ e'\ c)
\end{array}
$$

Finally, to apply the induction hypothesis for the expression x, i.e. $\mathit{eval}''\ x\ e\ c' = \mathit{exec}\ c'\ (\mathit{Clo}\ x'\ e')$, we need to solve the equation

$$\mathit{exec}\ c'\ (\mathit{Clo}\ x'\ e') = \mathit{eval}''\ y\ e\ (\mathit{FUN}\ x'\ e'\ c)$$

for which purposes we add another new constructor to the CONT type that takes the free variables y, e and c in this equation as arguments,

$$\mathit{ARG} :: \mathit{Expr} \to \mathit{Env} \to \mathit{CONT} \to \mathit{CONT}$$

add a new equation for exec

$$\mathit{exec}\ (\mathit{ARG}\ y\ e\ c)\ (\mathit{Clo}\ x'\ e') = \mathit{eval}''\ y\ e\ (\mathit{FUN}\ x'\ e'\ c)$$

and then conclude as follows:

$$
\begin{array}{ll}
& \mathit{eval}''\ y\ e\ (\mathit{FUN}\ x'\ e'\ c) \\
= & \{ \text{ define: } \mathit{exec}\ (\mathit{ARG}\ y\ e\ c)\ (\mathit{Clo}\ x'\ e') = \mathit{eval}''\ y\ e\ (\mathit{FUN}\ x'\ e'\ c) \ \} \\
& \mathit{exec}\ (\mathit{ARG}\ y\ e\ c)\ (\mathit{Clo}\ x'\ e') \\
= & \{ \text{ induction hypothesis for } x \ \} \\
& \mathit{eval}''\ x\ e\ (\mathit{ARG}\ y\ e\ c)
\end{array}
$$

In a similar manner to the first example, we can then define the top-level evaluation function simply by applying $eval''$ to a nullary constructor $HALT$.

In summary, we have calculated the following definitions, which together form an abstract machine for evaluating lambda expressions:

$$
\begin{array}{l}
\textbf{data}\ CONT\ \textbf{where} \\
\quad HALT :: CONT \\
\quad ARG\ \ :: Expr \to Env \to CONT \to CONT \\
\quad FUN\ \ :: Expr \to Env \to CONT \to CONT \\
\\
eval'' \qquad\qquad\qquad\qquad :: Expr \to Env \to CONT \to Value \\
eval''\ (Var\ i)\ e\ c \qquad\quad = exec\ c\ (e\ !!\ i) \\
eval''\ (Abs\ x)\ e\ c \qquad\quad = exec\ c\ (Clo\ x\ e) \\
eval''\ (App\ x\ y)\ e\ c \qquad = eval''\ x\ e\ (ARG\ y\ e\ c) \\
\\
exec \qquad\qquad\qquad\qquad :: CONT \to Value \to Value \\
exec\ (ARG\ y\ e\ c)\ (Clo\ x'\ e') = eval''\ y\ e\ (FUN\ x'\ e'\ c) \\
exec\ (FUN\ x'\ e'\ c)\ u \qquad = eval''\ x'\ (u : e')\ c \\
\\
eval \qquad\qquad\qquad\qquad :: Expr \to Env \to Value \\
eval\ x\ e \qquad\qquad\qquad\quad = eval''\ x\ e\ HALT
\end{array}
$$

The names of the control stack commands are based on the intuition that ARG evaluates the argument of a function application, whereas FUN evaluates the function body. The resulting abstract machine coincides with the CEK machine [6]. We have also calculated abstract machines for lambda calculi with call-by-name and call-by-need semantics, which correspond to the Krivine machine [9] and a lazy variant of the Krivine machine derived by Ager et. al [3], respectively. The calculations can be found in the associated Coq proofs [7].

5 Summary and Conclusion

We presented the simple idea of applying fusion on the level of program calculations. Instead of first producing continuations via CPS transformation and then immediately turning them into data via defunctionalisation, we transform the original program in one go into the target program. Despite its conceptual simplicity, the benefits of this idea are considerable: the input program and the output program are linked *directly* to the desired output program via the specification that drives the calculation. This directness simplifies the calculation process and allows us to guide the process towards the desired output more easily. Moreover, by cutting out the continuations as the middleman, we avoid some of its complexities. For example, we have found that using CPS transformation for deriving compilers often yields non-strictly-positive datatypes, which makes the transformation unsuitable for formalisation in proof assistants. However, the non-strictly-positive datatypes disappear after defunctionalisation, and by fusing the two transformations we avoid the issue entirely.

To demonstrate the generality of the idea presented in this paper, we applied it to a variety of examples. The associated Coq proofs [7] contain formal calculations of abstract machines for different varieties of lambda calculi as well as

languages with exception handling and state. The systematic nature of the calculations indicates the potential for automation of the entire derivation process with little or no interaction from the user apart from the specification.

Acknowledgements

We thank the anonymous reviewers for their comments and suggestions.

References

1. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A Functional Correspondence Between Evaluators and Abstract Machines. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming (2003)
2. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: From Interpreter to Compiler and Virtual Machine: A Functional Derivation. Technical Report RS-03-14, BRICS, Department of Computer Science, University of Aarhus (2003)
3. Ager, M.S., Danvy, O., Midtgaard, J.: A Functional Correspondence Between Call-by-need Evaluators and Lazy Abstract Machines. Information Processing Letters 90(5), 223 – 232 (2004)
4. Backhouse, R.: Program Construction: Calculating Implementations from Specifications. John Wiley and Sons, Inc. (2003)
5. Bahr, P., Hutton, G.: Calculating Correct Compilers. Journal of Functional Programming 25 (2015)
6. Felleisen, M., Friedman, D.P.: Control Operators, the SECD Machine, and the λ-calculus. In: Wirsing, M. (ed.) Formal Description of Programming Concepts III, pp. 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam (1986)
7. Hutton, G., Bahr, P.: Associated Coq proofs. `http://github.com/pa-ba/cps-defun`
8. Hutton, G., Wright, J.: Calculating an Exceptional Machine. In: Loidl, H.W. (ed.) Trends in Functional Programming. Intellect (Feb 2006)
9. Krivine, J.L.: Un Interpréteur du Lambda-calcul (1985), unpublished manuscript
10. Landin, P.J.: The Mechanical Evaluation of Expressions. Computer Journal 6(4), 308–320 (1964)
11. Meijer, E.: Calculating Compilers. Ph.D. thesis, Katholieke Universiteit Nijmegen (1992)
12. Reynolds, J.C.: Definitional Interpreters for Higher-Order Programming Languages. In: Proceedings of the ACM Annual Conference. pp. 717–740 (1972)
13. Wand, M.: Deriving Target Code as a Representation of Continuation Semantics. ACM Transactions on Programming Languages and Systems 4(3), 496–517 (Jul 1982)
14. Winskel, G.: The Formal Semantics of Programming Languages – An Introduction. Foundation of Computing Series, MIT Press (1993)