Accurate Step Counting

Catherine Hope and Graham Hutton

School of Computer Science and IT
University of Nottingham, UK
{cvh,gmh}@cs.nott.ac.uk

Abstract Starting with an evaluator for a language, an abstract machine for the same language can be mechanically derived using successive program transformations. This has relevance to studying both the time and space properties of programs because these can be estimated by counting transitions of the abstract machine and measuring the size of the additional data structures needed, such as environments and stacks. In this paper we will use this process to derive a function that accurately counts the number of steps required to evaluate expressions in a simple language, and illustrate this function with a range of examples.

1 Introduction

The problem of reasoning about intensional properties of functional programs, such as the time requirements, is a long-running one. It is complicated by different evaluation strategies and sharing of expressions, meaning that some parts of a program may not be run or only partially so. One of the issues involved in reasoning about the amount of time a program will take to complete is what to count as an atomic unit in evaluation, or an evaluation step.

An evaluator is usually an implementation of the denotational semantics [13] of a language — it evaluates an expression based on the meaning of its sub-expressions. This level of understanding helps us reason about extensional properties of the language, but it doesn't say anything about the underlying way that the evaluation is taking place. By contrast, an operational semantics [10] shows us the method that is being used to evaluate an expression, and the conventional approach is to use this to measure the number of steps that are required. This, however, may not be very accurate because what is usually being measured is β-reductions, each of which may take arbitrarily long. An example of this approach is using the tick monad [15], which counts β-reductions.

It is proposed that a more realistic measure would be to count transitions in an actual machine. The idea is to get more detailed information than by just counting β-reductions, but still have a principled way of obtaining this information. This would not be as accurate as actual time measurements, but would provide a useful half-way house between these two approaches. What is needed is a justification that the machine correctly implements the evaluation semantics and a way to reflect the number of steps required by the machine back to the semantic level.

Recently Danvy *et al* have explored the basis of abstract machines, and the process of deriving them from evaluators [1, 2, 3, 5]. This process uses, in particular, two program transformation techniques: transformation to continuation passing style, to make order of evaluation explicit; and defunctionalization, to eliminate the consequent use of higher-order functions.

In this paper, we will apply this process to the problem of accurately counting evaluation steps. In brief, this consists of introducing a simple language in which to write expressions, an evaluator for this language, and then deriving a corresponding abstract machine using successive program transformations. Simple step counting is then added to the machine, by threading and incrementing a counter that measures the number of transitions.

The next stage is to apply the same process but in the reverse order, resulting in an evaluator that additionally counts the number of steps, directly corresponding to the number of transitions of the underlying abstract machine. Finally, a direct step counting function will be calculated from this evaluator and will be used to reason about evaluation of some example computations expressed in the language.

A particular aspect of our derivation process is that the program at each stage is *calculated* directly from a specification of its behaviour [8]. All the programs are written in Haskell [9], but no specific features of this language are used, so they may be easily adapted to other functional languages.

2 A Simple Language

To start with we will consider a simple language with expressions consisting of integers and addition, and with integers as values:

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

Although this language is not powerful enough to be used to analyse the time requirements of any interesting computations, it will be sufficient to show the derivation process without over-complication. An extended language will be presented later to look at some example functions.

2.1 Evaluator

The initial evaluator takes an expression and evaluates it to a value:

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

That is, evaluating an integer value returns that integer, and evaluating an addition evaluates both sides of the addition to an integer and then adds them together. The order of evaluation is not specified at this level but will be determined by the semantics of the underlying language; in particular, when the expression is an addition, $\mathit{Add}\ x\ y$, the evaluation of the x or y may occur first.

2.2 Tail-recursive Evaluator

Our aim is to turn the evaluator into an *abstract machine*, a term-rewriting system that makes explicit the step-by-step process by which evaluation can be performed. More precisely, we seek to construct an abstract machine implemented in Haskell as a first-order, tail-recursive function.

The evaluator is already first order, but it is not tail-recursive. It can be made so by transforming it to continuation passing style (CPS) [11]. A continuation is a function that represents the rest of a computation; this makes the evaluation order of the arguments explicit, so intermediate results need to be ordered using the continuation. A program can be transformed in to CPS by redefining it to take an extra argument, a function which is applied to the result of the original one. In our case, the continuation function will take an argument of type *Value* and its result is a *Value*:

$$\textbf{type}\ Con = Value \to Value$$

The new tail-recursive evaluator can be calculated from the old one by using the specification:

$$\begin{aligned} &evalTail \quad :: Expr \to Con \to Value \\ &evalTail\ e\ c = c\ (eval\ e) \end{aligned}$$

That is, the new evaluator has the same behaviour as simply applying the continuation to the result of the original evaluator. The definition of this function can be calculated by performing induction on the structure of the expression, e.

Case : $e = Val\ v$

$$\begin{aligned} & evalTail\ (Val\ v)\ c \\ = & \qquad \{\ \text{specification}\ \} \\ & c\ (eval\ v) \\ = & \qquad \{\ \text{definition of } eval\ \} \\ & c\ v \end{aligned}$$

Case : $e = Add\ x\ y$

$$\begin{aligned} & evalTail\ (Add\ x\ y)\ c \\ = & \qquad \{\ \text{specification}\ \} \\ & c\ (eval\ (Add\ x\ y)) \\ = & \qquad \{\ \text{definition of } eval\ \} \\ & c\ (eval\ x + eval\ y) \\ = & \qquad \{\ \text{reverse } \beta\text{-reduction, abstract over } eval\ x\ \} \\ & (\lambda m \to c\ (m + eval\ y))\ (eval\ x) \\ = & \qquad \{\ \text{inductive assumption for } x\ \} \\ & evalTail\ x\ (\lambda m \to c\ (m + eval\ y)) \\ = & \qquad \{\ \text{reverse } \beta\text{-reduction, abstract over } eval\ y\ \} \\ & evalTail\ x\ (\lambda m \to (\lambda n \to c\ (m + n))\ (eval\ y)) \\ = & \qquad \{\ \text{inductive assumption for } y\ \} \\ & evalTail\ x\ (\lambda m \to evalTail\ y\ (\lambda n \to c\ (m + n))) \end{aligned}$$

In conclusion, we have calculated the following recursive definition:

$$
\begin{array}{lll}
\mathit{evalTail} & :: \mathit{Expr} \to \mathit{Con} \to \mathit{Value} \\
\mathit{evalTail}\ (\mathit{Val}\ v) & c = c\ v \\
\mathit{evalTail}\ (\mathit{Add}\ x\ y) & c = \mathit{evalTail}\ x\ (\lambda m \to \\
 & \qquad \mathit{evalTail}\ y\ (\lambda n \to c\ (m + n)))
\end{array}
$$

In the case when the expression is an integer the continuation is simply applied to the integer value. In the addition case, the first argument to the addition is evaluated first, with the result being passed in to a continuation. The second expression argument is then evaluated inside the continuation, with its result being passed in to an inner continuation. Both integer results are added together in the body of this function and the original continuation is applied to the result.

The evaluator is now tail recursive, in that the right hand side is a direct recursive call and there is nothing to be done after the call returns. In making the evaluator tail-recursive we have introduced an explicit evaluation order: the evaluation of the addition now has to occur in left-to-right order.

The semantics of the original evaluation function can be recovered by substituting in the identity function for the continuation:

$$\mathit{eval}\ e = \mathit{evalTail}\ e\ (\lambda v \to v)$$

2.3 Abstract Machine

The next step is to make the evaluator first order. This is done by defunctionalizing the continuations [11]. At the moment, the continuations are functions of the type $\mathit{Value} \to \mathit{Value}$, but the whole function space is not required: the continuation functions are only created in three different ways. Defunctionalization is performed by looking at all places where functions are made and replacing them with a new data structure that takes as arguments any free variables required.

The data structure required is as follows,

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Cont} = \mathit{Top} & & \text{for the initial continuation } (\lambda v \to v) \\
\quad \mid \mathit{AddL}\ \mathit{Cont}\ \mathit{Expr} & & \text{for } (\lambda m \to \mathit{evalTail}\ y\ (...)) \\
\quad \mid \mathit{AddR}\ \mathit{Value}\ \mathit{Cont} & & \text{for } (\lambda n \to c\ (m + n))
\end{array}
$$

The reason for the constructor names is that the data structure is the structure of evaluation contexts for the language [6]. It could alternatively be viewed as a stack, pushing expressions still to be evaluated and values to be saved.

To recover the functionality of the continuation we define an *apply* function which has the same semantics for each instance of the continuation function:

$$
\begin{array}{ll}
\mathit{apply} & :: \mathit{Cont} \to \mathit{Con} \\
\mathit{apply}\ \mathit{Top} & = \lambda v \to v \\
\mathit{apply}\ (\mathit{AddR}\ m\ c) & = \lambda n \to \mathit{apply}\ c\ (m + n) \\
\mathit{apply}\ (\mathit{AddL}\ c\ y) & = \lambda m \to \mathit{evalTail}\ y\ (\mathit{apply}\ (\mathit{AddL}\ c\ m))
\end{array}
$$

We now seek to construct a new evaluator, *evalMach*, that behaves in the same way as *evalTail*, except that it uses representations of continuations, rather than

real continuations; that is, we require $evalMach\ e\ c = evalTail\ e\ (apply\ c)$. From this specification, the definition of *evalMach* can be calculated by induction on e:

Case : $e = Val\ v$

$$
\begin{array}{ll}
 & evalMach\ (Val\ v)\ c \\
= & \quad \{\ \text{specification}\ \} \\
 & evalTail\ (Val\ v)\ (apply\ c) \\
= & \quad \{\ \text{definition of } evalTail\ \} \\
 & apply\ c\ v
\end{array}
$$

Case : $e = Add\ x\ y$

$$
\begin{array}{ll}
 & evalMach\ (Add\ x\ y)\ c \\
= & \quad \{\ \text{specification}\ \} \\
 & evalTail\ (Add\ x\ y)\ (apply\ c) \\
= & \quad \{\ \text{definition of } evalTail\ \} \\
 & evalTail\ x\ (\lambda m \rightarrow evalTail\ y\ (\lambda n \rightarrow apply\ c\ (m + n))) \\
= & \quad \{\ \text{definition of } apply\ \} \\
 & evalTail\ x\ (\lambda m \rightarrow evalTail\ y\ (apply\ (AddR\ m\ c))) \\
= & \quad \{\ \text{definition of } apply\ \} \\
 & evalTail\ x\ (apply\ (AddL\ c\ y)) \\
= & \quad \{\ \text{inductive assumption, for } x\ \} \\
 & evalMach\ x\ (AddL\ c\ y)
\end{array}
$$

We have now calculated the following recursive function:

$$
\begin{array}{lll}
evalMach & & :: Expr \rightarrow Cont \rightarrow Value \\
evalMach\ (Val\ v) & c & = apply\ c\ v \\
evalMach\ (Add\ x\ y) & c & = evalMach\ x\ (AddL\ c\ y)
\end{array}
$$

Evaluating an integer calls the *apply* function with the current context and the integer value. Evaluating an addition evaluates the first argument and stores the second with the current context using the *AddL* constructor.

Moving the λ-abstracted terms to the left and applying the specification in the *AddL* case, gives the following revised definition for *apply*:

$$
\begin{array}{lll}
apply & & :: Cont \rightarrow Value \rightarrow Value \\
apply\ Top & v & = v \\
apply\ (AddR\ m\ c) & n & = apply\ c\ (m + n) \\
apply\ (AddL\ c\ y) & m & = evalMach\ y\ (AddR\ m\ c)
\end{array}
$$

The *apply* function takes a context and a value and returns the value if the context is *Top*. When the context is *AddR* this represents the case when both sides of the addition have been evaluated, so the results are added together and the current context is applied to the result. The *AddL* context represents evaluating the second argument to the addition, so the *evalMach* function is called and the result from the first argument and the current context saved using the *AddR* context.

The original semantics can be recovered by passing in the equivalent of the initial continuation, the *Top* constructor:

$$eval\ e = evalMach\ e\ Top$$

2.4 Step Counting Machine

The number of time steps required to evaluate an expression is to be measured by counting the number of transitions of the abstract machine. The abstract machine derived can be simply modified by adding a step count that is incremented each time a transition, a function call to *evalMach* or *apply*, is made. The step count is added as an accumulator, rather than just incrementing the count that the recursive call returns, so that it is still an abstract machine.

It is relevant here to note that this is just one possible derivation to produce an abstract machine. Different abstract machines may be generated by applying different program transformations, as demonstrated in [5].

$$\begin{array}{lll}
\textbf{type}\ Step & & = Int \\
stepMach & & :: (Expr, Step) \to Cont \to (Value, Step) \\
stepMach\ (Val\ v, s) & c & = apply'\ c\ (v, s+1) \\
stepMach\ (Add\ x\ y, s) & c & = stepMach\ (x, s+1)\ (AddL\ c\ y) \\
apply' & & :: Cont \to (Value, Step) \to (Value, Step) \\
apply'\ Top & (v, s) & = (v, s+1) \\
apply'\ (AddL\ c\ y) & (m, s) & = stepMach\ (y, s+1)\ (AddR\ m\ c) \\
apply'\ (AddR\ m\ c) & (n, s) & = apply'\ c\ (m+n, s+1)
\end{array}$$

In this case we are only counting the machine transitions and the actual addition of the integers is defined to happen instantly, though this could be extended by introducing an additional factor that represents the number of steps to perform an addition.

The evaluation function now returns a pair, where the first part is the evaluated value, and the second is the number of steps taken, which is initialised to zero. Therefore, the semantics of the original evaluator can be recovered by taking the first part of the pair returned:

$$eval\ e = fst\ (stepMach\ e\ 0\ Top)$$

2.5 Step Counting Tail-recursive Evaluator

The aim now is to derive a function that counts the number of steps required to evaluate an expression. The specification for this is the second part of the pair returned by the abstract machine:

$$steps\ e = snd\ (stepMach\ (e, 0)\ Top)$$

The first stage in the reverse process is to refunctionalize the representation of the continuation. The original continuation was a function of type $Value \to Value$, so the type of the new one will be $(Value, Step) \to (Value, Step)$.

$$\textbf{type}\ \mathit{Con}' = (\mathit{Value}, \mathit{Step}) \to (\mathit{Value}, \mathit{Step})$$

Again, this can be calculated by induction on e, from the following specification:

$$\mathit{stepTail}\ (e, s)\ (\mathit{apply}'\ c') = \mathit{evalMach}\ (e, s)\ c'$$

The refunctionalized version is:

$$\begin{aligned}
&\mathit{stepTail} && :: (\mathit{Expr}, \mathit{Step}) \to \mathit{Con}' \to (\mathit{Value}, \mathit{Step})\\
&\mathit{stepTail}\ (\mathit{Val}\ v)\ \ s\ c &&= c\ (v, s+1)\\
&\mathit{stepTail}\ (\mathit{Add}\ x\ y)\ s\ c &&= \mathit{stepTail}\ x\ (s+1)\ (\lambda(m, s') \to\\
&&&\qquad \mathit{stepTail}\ y\ (s'+1)\ (\lambda(n, s'') \to c\ (m+n, s''+1)))
\end{aligned}$$

The step counting semantics can be redefined as:

$$\mathit{steps}\ e = \mathit{snd}\ (\mathit{stepTail}\ (e, 0)\ ((v, s) \to (v, s+1)))$$

Now, the same program transformations are performed in the reverse order to derive an evaluator that counts steps at the evaluator level, corresponding to the number of transitions of the abstract machine.

2.6 Step Counting Evaluator with Accumulator

The step counting evaluator can be transformed from CPS back to direct style by calculation, using the following specification to remove the continuation:

$$c\ (\mathit{stepAcc}\ (e, s)) = \mathit{stepTail}\ (e, s)\ c$$

The resulting evaluator is:

$$\begin{aligned}
&\mathit{stepAcc} && :: (\mathit{Expr}, \mathit{Step}) \to (\mathit{Value}, \mathit{Step})\\
&\mathit{stepAcc}\ (\mathit{Val}\ v)\ \ s &&= (v, s+1)\\
&\mathit{stepAcc}\ (\mathit{Add}\ x\ y)\ s &&= \textbf{let}\ (m, s') = \mathit{stepAcc}\ (x, s+1)\\
&&&\qquad\quad (n, s'') = \mathit{stepAcc}\ (y, s'+1)\\
&&&\quad \textbf{in}\ (m+n, s''+1)
\end{aligned}$$

The new step counting function becomes:

$$\mathit{steps}\ e = \mathit{snd}\ (\mathit{stepAcc}\ (e, 0)) + 1$$

2.7 Step Counting Evaluator

At the moment the step counting evaluator treats the step count as an accumulator. This can be removed, by calculating a new function without one, using the specification:

$$
\begin{aligned}
&\mathit{stepEval}\ e = \mathbf{let}\ (v, s') = \mathit{stepAcc}\ (e, s) \\
&\qquad\qquad\ \mathbf{in}\ (v, s' - s)
\end{aligned}
$$

Again, this can be calculated by induction over the structure of the expression, to give the new step counting evaluator:

$$
\begin{aligned}
&\mathit{stepEval} && :: \mathit{Expr} \to (\mathit{Value}, \mathit{Step}) \\
&\mathit{stepEval}\ (\mathit{Val}\ v) && = (v, 1) \\
&\mathit{stepEval}\ (\mathit{Add}\ x\ y) && = \mathbf{let}\ (m, s) = \mathit{stepEval}\ x \\
&&&\qquad\quad (n, s') = \mathit{stepEval}\ y \\
&&&\quad\ \mathbf{in}\ (m + n, s + s' + 3)
\end{aligned}
$$

The semantics of the *steps* function can be expressed as:

$$\mathit{steps}\ e = \mathit{snd}\ (\mathit{stepEval}\ e) + 1$$

2.8 Step Counting Function

The final stage is to calculate a standalone steps function. This will take an expression and return the number of steps to evaluate the expression, calling the original evaluator when the result of evaluation is required. The resulting function can be produced by routine calculation:

$$
\begin{aligned}
&\mathit{steps}\ e && = \mathit{steps}'\ e + 1 \\
&\mathit{steps}'\ (\mathit{Val}\ v) && = 1 \\
&\mathit{steps}'\ (\mathit{Add}\ x\ y) && = \mathit{steps}'\ x + \mathit{steps}'\ y + 3
\end{aligned}
$$

The derived *steps* function shows that that evaluation of an expression is an auxiliary function steps' plus a constant one. This increment operation at the end comes from the transition in the abstract machine that evaluates the initial (Top) context. In steps', the number of steps to evaluate an integer is a constant one, and the number of steps to evaluate an addition is the number of steps to evaluate each argument plus a constant three. This is more accurate because we now see the overhead of each addition: if we were counting β-reductions then this would only have been a single step of evaluation.

3 Extending the Language

We've now shown the derivation process for a small test language, but this language is not powerful enough to express computations that have interesting time behaviour. We now extend it with the untyped λ-calculus (variables, abstraction and application), lists and recursion over lists in the form of fold-right.

These could have been expressed directly in λ-calculus, for example by using Church encoding instead of integers, but this would introduce an unrealistic overhead in evaluation. Moreover, a more general recursion operator could have been introduced instead of fold-right, but for simplicity one tailored to our data structure, lists, is sufficient. Using fold-right will also simplify the process of reasoning about time properties, just as it has proved useful for reasoning about extensional ones [7].

The language is implemented as the following Haskell data type:

$$\begin{aligned}&\textbf{data } \mathit{Expr} = \mathit{Var}\ \mathit{String} \mid \mathit{Abs}\ \mathit{String}\ \mathit{Expr} \mid \mathit{App}\ \mathit{Expr}\ \mathit{Expr}\\ &\qquad\qquad \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Val}\ \mathit{Value}\\ &\qquad\qquad \mid \mathit{Cons}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Foldr}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr}\\ &\textbf{data } \mathit{Value} = \mathit{Const}\ \mathit{Int} \mid \mathit{ConsV}\ \mathit{Value}\ \mathit{Value} \mid \mathit{Nil} \mid \mathit{Clo}\ \mathit{String}\ \mathit{Expr}\ \mathit{Env}\end{aligned}$$

That is, an expression is either a variable, abstraction, application of two expressions, addition of two expressions, fold-right over an expression (where the function and empty list-case arguments are both expressions), a list containing further expressions or a value. In turn, a value is either an integer, list containing further values or a closure (an abstraction paired with an environment containing bindings for all free variables in the abstraction).

The data type differentiates between expressions and values, in particular lists that contain unevaluated and evaluated expressions, so that there is no need to iterate repeatedly over the list to check if each element is fully evaluated, which would introduce an artificial evaluation overhead.

The primitive functions, such as *Add* and *Foldr*, are implemented as fully saturated, in that they take their arguments all at once. The main reason for this is to make it easier to define what a value is: if they were introduced as constants then, for example, $\mathit{App}\ \mathit{Add}\ 1$ would be a value, since it cannot be further evaluated. This doesn't affect what can be expressed in the language; partial application can be expressed by using abstractions, so an equivalent expression would be Abs `"x"` $(\mathit{Add}\ 1\ (\mathit{Var}$ `"x"`$))$, which would be evaluated to a closure.

Note that, for simplicity, the language has not been provided with a type system. Rather, in this article we assume that only well-formed expressions are considered.

4 Evaluator

For simplicity, we will consider evaluation using the call-by-value strategy, where arguments to functions are evaluated before the function is applied. Evaluation is performed using an environment, that is used to look up what a variable is bound to. This avoids having to substitute in expressions for variables, which is complicated by the need to deal with avoiding name-capture. Under call-by-value evaluation arguments are evaluated before function application, so variables will be bound to a value. The environment is represented as a list of pairs:

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

The initial evaluator is given below:

$$
\begin{array}{lll}
\mathit{eval} & & :: \mathit{Expr} \to \mathit{Env} \to \mathit{Value} \\
\mathit{eval}\ (\mathit{Val}\ v) & \mathit{env} = v \\
\mathit{eval}\ (\mathit{Var}\ x) & \mathit{env} = \mathit{fromJust}\ (\mathit{lookup}\ x\ \mathit{env}) \\
\mathit{eval}\ (\mathit{Abs}\ x\ e) & \mathit{env} = \mathit{Clo}\ x\ e\ \mathit{env} \\
\mathit{eval}\ (\mathit{App}\ f\ e) & \mathit{env} = \mathbf{let}\ \mathit{Clo}\ x\ e'\ \mathit{env}' = \mathit{eval}\ f\ \mathit{env} \\
 & \qquad\qquad\quad v = \mathit{eval}\ e\ \mathit{env} \\
 & \qquad\quad \mathbf{in}\ \mathit{eval}\ e'\ ((x, v) : \mathit{env}') \\
\mathit{eval}\ (\mathit{Add}\ x\ y) & \mathit{env} = \mathbf{let}\ \mathit{Const}\ m = \mathit{eval}\ x\ \mathit{env} \\
 & \qquad\qquad\quad \mathit{Const}\ n = \mathit{eval}\ y\ \mathit{env} \\
 & \qquad\quad \mathbf{in}\ \mathit{Const}\ (m + n) \\
\mathit{eval}\ (\mathit{Cons}\ x\ \mathit{xs}) & \mathit{env} = \mathit{ConsV}\ (\mathit{eval}\ x\ \mathit{env})\ (\mathit{eval}\ \mathit{xs}\ \mathit{env}) \\
\mathit{eval}\ (\mathit{Foldr}\ f\ v\ \mathit{xs}) & \mathit{env} = \mathbf{case}\ \mathit{eval}\ \mathit{xs}\ \mathit{env}\ \mathbf{of} \\
\quad \mathit{Nil} \to \mathit{eval}\ v\ \mathit{env} & \\
\quad \mathit{ConsV}\ z\ \mathit{zs} \to \mathbf{let}\ f' = \mathit{eval}\ f\ \mathit{env} & \\
 & \qquad\qquad\quad x = \mathit{eval}\ (\mathit{Foldr}\ (\mathit{Val}\ f')\ v\ (\mathit{Val}\ \mathit{zs}))\ \mathit{env} \\
 & \qquad\quad \mathbf{in}\ \mathit{eval}\ (\mathit{App}\ (\mathit{App}\ (\mathit{Val}\ f')\ (\mathit{Val}\ z))\ (\mathit{Val}\ x))\ [\,]
\end{array}
$$

That is, values are already evaluated, so they are simply returned. Variables are evaluated by returning the value the variable is bound to in the environment. Under the call-by-value strategy, evaluation is not performed under λ-terms, so abstractions are turned in to values by pairing them with the current environment to make a closure. An application, $\mathit{App}\ f\ e$, is evaluated by first evaluating f to an abstraction, then evaluating the body of the abstraction, with the environment extended with the variable bound to the value that e evaluates to.

Addition is performed by first evaluating both sides to an integer and then adding them together. This will give another integer result and so does not require further evaluation. Evaluating a *Cons* consists of evaluating the first and second arguments (the head and the tail of the list) and then re-assembling them using the ConsV constructor to make an evaluated list.

Evaluation of the *Foldr* case proceeds by first evaluating the list argument and doing case analysis. If the list is *Nil* then the result is the evaluation of the second argument, v. In the non-empty case, first the function argument is evaluated, then the fold-right applied to the tail of the list, and finally the function is applied to the head of the list and the result of folding the tail of the list. This evaluation is performed with an empty environment, since all the expressions are values at that point.

The evaluation of the fold-right could have been specified in different ways. The completely call-by-value way would be to evaluate the arguments in left to right order, so that the first two arguments are evaluated before the list argument. However, for the *Nil* list argument case, the function argument to *Foldr* is evaluated even though it is not required. The approach in the evaluator is to evaluate the list argument first to allow pattern matching and then evaluate the other arguments depending on what the list evaluated to. So when the list evaluates to *Nil* only the second argument to *Foldr* is evaluated. The justification not to use the purely call-by-value way is that it would introduce

some artificial behaviour of the *Foldr* function. When the λ-calculus is extended with a conditional function, for example, it is not implemented to expand both branches under call-by-value evaluation, but more efficiently by evaluating the condition first and then one branch depending on the value of the condition. In practice, this has little effect because the function supplied to the fold is often an abstraction and therefore is already evaluated.

5 Complete Function

Performing the derivation process for this extended language (which, as previously, proceeds by calculation) yields the following *steps* function:

$$
\begin{array}{lll}
steps\ e & & = steps'\ e\ [\,] + 1 \\
steps'\ (Val\ v) & env & = 1 \\
steps'\ (Var\ x) & env & = 1 \\
steps'\ (Abs\ x\ e) & env & = 1 \\
steps'\ (App\ f\ e) & env & = \mathbf{let}\ (Clo\ x\ e'\ env') = eval\ f\ env \\
 & & \qquad\quad v = eval\ e\ env \\
 & & \quad \mathbf{in}\ steps'\ f\ env + steps'\ e\ env + \\
 & & \qquad steps'\ e'\ ((x, v) : env') + 3 \\
steps'\ (Add\ x\ y) & env & = steps'\ x\ env + steps'\ y\ env + 3 \\
steps'\ (Cons\ x\ xs) & env & = steps'\ x\ env + steps'\ xs\ env + 3 \\
steps'\ (Foldr\ f\ v\ xs) & env & = steps'\ xs\ env + \mathbf{case}\ eval\ xs\ env\ \mathbf{of} \\
\quad Nil \rightarrow steps'\ v\ env + 2 & & \\
\quad ConsV\ y\ ys \rightarrow \mathbf{let}\ f' = Val\ (eval\ f\ env) & & \\
\qquad\qquad\qquad\qquad x = Val\ (eval\ (Foldr\ (Val\ f')\ v\ (Val\ ys))\ env) & & \\
\qquad\qquad\qquad\quad \mathbf{in}\ steps'\ f\ env + steps'\ (Foldr\ (Val\ f')\ v\ (Val\ ys))\ env + & & \\
\qquad\qquad\qquad\qquad\qquad steps'\ (App\ (App\ f'\ (Val\ y))\ x)\ [\,] + 4 & &
\end{array}
$$

As mentioned earlier, the derived function has calls to the original evaluator, where the result of evaluation is required. For example, in the *Foldr* case, a case analysis has to be performed on the evaluated third argument, to know if it is empty or not, and so whether to supply the v argument, or to keep folding.

We want to be able to reason about how the time requirements of some example functions depends on the size of the arguments to the function. In the case of functions defined using fold-right, it would be easier to reason about the time usage if it was expressed as a function over the size of the list, rather than as a recursive function — the *steps* function above the *Foldr* case makes a recursive call to fold the tail of the list. This can naturally be expressed as a fold-right over the value list data structure, defined as:

$$
\begin{array}{ll}
foldrVal & :: (Value \rightarrow b \rightarrow b) \rightarrow b \rightarrow Value \rightarrow b \\
foldrVal\ f\ v\ Nil & = v \\
foldrVal\ f\ v\ (ConsV\ x\ xs) & = f\ x\ (foldrVal\ f\ v\ xs)
\end{array}
$$

Also, if the number of steps to apply the function f to two arguments does not depend on the value of the arguments, such as adding two expressions, then a useful further simplification is to express this as a function over the length of the list argument supplied, defined as:

$$lengthVal = foldrVal\ (\lambda\_\ n \rightarrow n + 1)\ 0$$

6 Example Functions

We can now use the derived *steps* function to look at some examples. Each step counting function produced was simplified by hand and then QuickCheck [4] was used to verify that the result produced was equal to the original function, to check that no errors had been introduced during simplification.

6.1 Summing a List

Summing a list of integers can be expressed using fold-right:

$$\begin{array}{l} sum\ [\,] \quad\ \ = 0 \\ sum\ (x : xs) = x + sum\ xs \end{array} \quad \Leftrightarrow \quad sum\ xs = foldr\ (+)\ 0\ xs$$

The fold-right definition replaces each list constructor (:) with +, and the [] at the end of the list with 0, the unit of addition. First we translate the definition of *sum* into the language syntax and then call the *steps* function on the application of *sum* to an argument. The number of steps required to evaluate applying the *sum* function to a list of integers, *xs*, is given below:

$$steps\ (App\ sum\ (Val\ xs)) = 21 * (lengthVal\ xs) + 10$$

The function can be expressed as a length because the steps required to evaluate the addition of two values is a constant. The step count is a constant multiplied by the length of the list argument plus a constant amount; it is directly proportional to the length of the list argument. Though, of more interest is that we can see the constant factors involved in the evaluation.

6.2 Sum with an Accumulator

An alternative definition of *sum* is to use an accumulator; the fold-right is used to generate a function which is applied to the identity function in the empty list case, and in the non-empty case adds the head of the list to the accumulator.

$$\begin{array}{l} sumAcc\ [\,]\ a \quad\ \ = a \\ sumAcc\ (x : xs)\ a = sumAcc\ xs\ (a + x) \end{array} \quad \Leftrightarrow \quad \begin{array}{l} sumAcc\ xs = foldr\ f\ id\ xs\ 0 \\ \quad \textbf{where}\ f\ x\ g\ a = g\ (a + x) \end{array}$$

Using an accumulator could potentially save on space, because additions could be performed without having to expand the whole list first. It would be useful to know what effect an accumulator has on the number of steps taken.

Translating the accumulator version and applying *steps* gives a result of the same form, linear on the length of the list, but the constant values are larger, because there is an additional overhead in evaluating the extra abstractions:

$$steps\ (App\ sumAcc\ (Val\ xs)) = 26 * (lengthVal\ xs) + 15$$

6.3 Concatenation

Concatenating a list of lists can be defined by folding the *append* function over the list, and *append* can also be expressed as a fold-right:

$$\begin{aligned} concat\ xs &= foldr\ append\ [\,]\ xs \\ append\ xs\ ys &= foldr\ (:)\ ys\ xs \end{aligned}$$

First we need to analyse the *append* function. The number of steps to evaluate *append* applied to two list arguments is as follows:

$$steps\ (App\ (App\ append\ (Val\ xs))\ (Val\ ys)) = 21 * (lengthVal\ xs) + 15$$

So the number of steps to evaluate an *append* is proportional to the length of the first list argument. The step count of the *concat* function can now be calculated using this function. With the step count from the *append* function inlined, the resulting *steps* function is:

$$\begin{aligned} steps\ (App\ concat\ (Val\ xss)) &= foldrVal\ f\ 10\ xss \\ &\quad \textbf{where}\ f\ ys\ s = 21 * (lengthVal\ ys) + 20 + s \end{aligned}$$

The number of steps required in evaluation is the sum of the steps taken to apply the *append* function to each element in the list. If the argument to *concat* is a list where all the list elements are of the same length (so the number of steps taken in applying the *append* function will always be constant), then this can be simplified to:

$$\begin{aligned} steps\ (App\ concat\ (Val\ xss)) = 20 + &\textbf{case}\ xss\ \textbf{of} \\ &Nil \rightarrow 0 \\ &ConsV\ ys\ yss \rightarrow (lengthVal\ xss) * (21 * (lengthVal\ ys)) \end{aligned}$$

The number of steps is now proportional to the length of the input list multiplied by the number of steps to evaluate appending an element of the list, which is proportional to the length of that element.

6.4 Reversing a List

Reversing a list can be expressed directly as a fold-right by appending the reversed tail of the list to the head element made in to a singleton list:

$$\begin{array}{l} \mathit{reverse}\ [\,] \quad\ = [\,] \\ \mathit{reverse}\ (x : \mathit{xs}) = \mathit{reverse}\ \mathit{xs} +\!\!+ [x] \end{array} \quad \Leftrightarrow \quad \begin{array}{l} \mathit{reverse}\ \mathit{xs} = \mathit{foldr}\ f\ [\,]\ \mathit{xs} \\ \quad \textbf{where}\ f\ x\ \mathit{xs} = \mathit{xs} +\!\!+ [x] \end{array}$$

Translating the definition of *reverse* in to the language syntax and applying the *steps* function gives the step count function:

$$\begin{array}{l} \mathit{steps}\ (\mathit{App}\ \mathit{reverse}\ (\mathit{Val}\ \mathit{xs})) = \mathit{fst}\ (\mathit{foldrVal}\ g\ (10, \mathit{Nil})\ \mathit{xs}) \\ \quad \textbf{where}\ g\ z\ (s, \mathit{zs}) = (s + 21 * (\mathit{lengthVal}\ \mathit{zs}) + 34, \\ \qquad\qquad\qquad\qquad \mathit{eval}\ (\mathit{App}\ (\mathit{App}\ \mathit{append}\ (\mathit{Val}\ \mathit{zs}))\ (\mathit{Val}\ (\mathit{ConsV}\ z\ \mathit{Nil})))\ [\,]) \end{array}$$

The steps function for *reverse* is dependent on the steps required to perform the *append* function for each element, which is proportional to length of the first argument to *append*. The size of this argument is increased by one each time, so the function is a sum up to the length of the list:

$$10 + \sum_{x=0}^{\mathit{length}\ \mathit{xs}-1} 21x + 34$$

Expanding this sum gives the following expression:

$$10 + 34 * (\mathit{length}\ \mathit{xs}) + \frac{21 * (\mathit{length}\ \mathit{xs} - 1) * (\mathit{length}\ \mathit{xs})}{2} \leqslant c\ (\mathit{length}\ \mathit{xs})^2$$

This is less than a constant multiplied by the square of the length of the list, for example when $c = 11$ for all lists of length greater than 47. This means that the time requirements are quadratic on the length of the list [14].

6.5 Fast Reverse

The reverse function can also be expressed using an accumulator:

$$\begin{array}{l} \mathit{fastrev}\ [\,]\ a \quad\ = a \\ \mathit{fastrev}\ (x : \mathit{xs})\ a = \mathit{fastrev}\ \mathit{xs}\ (x : a) \end{array} \quad \Leftrightarrow \quad \begin{array}{l} \mathit{fastrev}\ \mathit{xs} = \mathit{foldr}\ f\ \mathit{id}\ \mathit{xs}\ 0 \\ \quad \textbf{where}\ f\ x\ g\ a = g\ (x : a) \end{array}$$

This definition should have better time properties because, as shown above, the steps required in evaluating the *append* function is proportional to the length of the first argument, so appending the tail of the list would be inefficient. The *steps* function produced for the accumulator version is directly proportional to the length of the list:

$$\mathit{steps}\ (\mathit{App}\ \mathit{fastrev}\ (\mathit{Val}\ \mathit{xs})) = 26 * (\mathit{lengthVal}\ \mathit{xs}) + 15$$

7 Conclusion and Further Work

We have presented a process that takes an evaluator for a language, and derives a function that gives an accurate count of the number of steps required to evaluate expressions using an abstract machine for the language. Moreover, all the steps in the derivation process are purely calculational, in that the required function at each stage is calculated directly from a specification of its desired behaviour.

Using an extended λ-calculus under call-by-value evaluation, the examples in the previous section give the expected complexity results, but also show the constants involved. This is useful to know because of the additional overheads that functions of the same complexity may have, for example in summing a list with and without an accumulator. They also show the boundaries at which one function with a lower growth rate but larger constants becomes quicker than another, for example in the two different definitions of the reverse function.

Ultimately, the most accurate information is to do real timing of programs. The approach taken in the profiler [12] that is distributed with the Glasgow Haskell Compiler, is similar to that here, in that abstract machines are used to bridge the gap between a big-step cost semantics and a small-step implementation. A regular clock interrupt is used to collect time information and assign to the costs of functions. However, this is a profiler, and we want to be able to express costs as a function over the arguments supplied, and not just in terms of hard time measurements. The thesis of this paper is that we can obtain this useful information relatively easily.

This work could be extended by considering more complicated evaluation strategies, such as call-by-name or lazy evaluation. It would also be interesting to apply the same technique to look at the space requirements for functions, by measuring the size of the additional data structures produced at the abstract machine level. Finally, a useful addition to this work would be to develop a calculus to automate deriving the step functions.

Acknowledgement

Finally, the authors would like to thank the anonymous referees for their comments and suggestions, which have considerably improved the paper.

References

[1] Mads Sig Ager. From natural semantics to abstract machines. In *Logic Based Program Synthesis and Transformation, 14th International Symposium (LOPSTR 2004), Revised Selected Papers.*

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. Technical Report RS-03-13, March 2003. 28 pp. Appears in , pages 8–19.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the technical report BRICS-RS-04-3.

[4] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[5] Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report RS-03-33, October 2003.

[6] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. Number CSR-04-1, pages 13–23, Birmingham B15 2TT, United Kingdom, 2004. Invited talk.

[7] Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.

[8] Graham Hutton and Joel Wright. Calculating an Exceptional Machine. To appear in the Proceedings of the Fifth Symposium on Trends in Functional Programming, 2005.

[9] S. Peyton Jones. Haskell 98 language and libraries: The revised report. Technical report.

[10] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[11] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher Order Symbol. Comput.*, 11(4):363–397, 1998.

[12] Patrick M. Sansom and Simon L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Trans. Program. Lang. Syst.*, 19(2):334–385, 1997.

[13] David A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[14] Clifford A. Shaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[15] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.