Accurate Step Counting

Catherine Hope and Graham Hutton

University of Nottingham

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 space and time 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.

1 Introduction

The problem of reasoning about intensional properties of functional programs, such as the running time, is itself 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 of the language — it evaluates the expression based on the meaning of its subexpressions. This level of understanding helps us reason about extensional properties of the language and makes the evaluation strategy explicit, but it doesn't say anything about the underlying way that the evaluation is taking place. By contrast, an operational semantics 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 is required. This, however, may not be very accurate because what is usually being measured is beta-reductions, each of which may take arbitrarily long.

It is proposed that a more realistic measure would be to count transitions in an actual machine. What is needed is some way to relate these two concepts together to produce a machine implementation of a language directly from the evaluator for it, so that we can be sure that they are both using the same evaluation method.

In the last couple of years Olivier Danvy *et al* have produced several papers exploring the basis of abstract machines and outlining the process of deriving them from evaluators [1, 2, 3]. Abstract machines can be viewed as an implementation of the operational semantics at a low level [4]. This process uses, in particular, two program transformation techniques: transformation in to continuation-passing style and defunctionalization. These will be explained in detail when they are required later.

In this paper, we will apply this process to that of accurately counting evaluation steps. In brief this consists of introducing a simple language in which to write expressions, together with an evaluator and 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 do the same process but in reverse order to get out an evaluator that additionally counts the number of steps, directly corresponding to the number of transitions of the underlying abstract machine. A step counting function will be calculated from this evaluator and this will be used to reason about evaluation of some example functions expressed in the language.

All the programs are written in Haskell [5], but no specific features of this language are used, so they may be easily adapted to other functional programming languages.

2 Language

The test language needs to be small, so that it's easy to reason about, but powerful enough to express functions that have interesting time behaviour. The language chosen is the untyped lambda calculus (variables, abstraction and application) extended with integers, addition, lists and recursion over lists in the form of fold-right. These could have been expressed directly in lambda calculus, for example by using peano numbers instead of introducing integers, but this would introduce an unrealistic overhead in evaluation. 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. It is also hoped that using fold-right will simplify the process of reasoning about time properties in the same way as it has proved useful for reasoning about extensional ones [6]. The syntax of the language chosen is expressed below using BNF.

$$E ::= x \mid \lambda x.e \mid E\ E \mid \mathbb{Z} \mid E + E \mid [\,] \mid E : E \mid \mathit{foldr}\ E\ E\ E$$

Using this language some common functions over lists can be expressed, such as summing a list, both with and without an accumulator, appending two lists, concatening a list of lists and reversing a list, again using an accumulator or not.

The primitive functions (*add*, *foldr* and :) are implemented as fully saturated, in that they take their arguments all at once. The main reason for doing this is to make it easier to define what a value is. If they were introduced as constants then, for example, *add* 1 would be a value. 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 $\lambda x.\mathit{add}\ 1\ x$.

3 Derivation

For simplicity, we will consider evaluation using the call-by-value strategy, where arguments to functions are evaluated before the function is performed.

The language is implemented as the Haskell data type shown below,

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

The data type diffentiates 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.

An expression is either a variable, abstraction, application of two expressions, addition of two expressions, fold-right over an expression (where the function and nil-case are both expressions), a list containing further expressions or a value. 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).

Evaluation is performed using an environment that is passed in to the evaluator function and 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 intermediate programs are written very concretely - the environment and step counter are threaded through the program - instead of introducing more structure, such as using monads. This makes it more obvious what is going on at a low-level, in that there are no hidden steps happening, but it can be verbose, so the derivations are shown for just a subset of the language: integers and addition.

3.1 Evaluator

An evaluator is a representation of the denotational semantics of the language. It is compositional, in that an expression is described as the meaning of its parts, and evaluation order isn't specified, it will depend on the implementing language. By contrast an abstract machine is a term rewriting system that represents a one-step transition function. However it differs from the reflexive transitive closure of a single step transition semantics (usually written as $\xrightarrow{*}$) because the next transition of the machine is entirely determined by the value currently being examined and the current state [4]. The implementation of this is a first-order tail-recursive program.

The starting evaluator is given below,

```
eval                       :: Expr → Env → Value
eval (Val v) env           =  v
eval (Var x) env           =  fromJust (lookup x env)
eval (Abs x e) env         =  Clo x e env
eval (App f e) env         =  let Clo x e′ env′ = eval f env
                                  v = eval e env
                              in eval e′ ((x, v) : env′)
eval (Add x y) env         =  let Const m = eval x env
                                  Const n = eval y env
                              in Const (m + n)
eval (Cons x xs) env       =  ConsV (eval x env) (eval xs env)
eval (Foldr f v xs) env    =  case eval xs env of
        Nil → eval v env
        ConsV z zs → let f′ = eval f env
                             x = eval (Foldr (Val f′) v (Val zs)) env
                         in eval (App (App (Val f′) (Val z)) (Val x)) []
```

Values are already evaluated, so they are just returned. Variables are evaluated by returning the value the variable is bound to in the environment. Under call-by-value evaluation, the arguments to functions are evaluated before the function is performed, so no further evaluation is required. An application *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 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, so does not need to be further evaluated. Evaluating a *cons* consists of evaluating the first element and the tail and then re-assembling them to make an evaluated list.

Evaluation of the *Foldr* case could be 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 justication not to use the purely call-by-value way is that it would introduce some artificial behaviour of the *Foldr* function. When the lambda-calculus is extended with a conditional function, for example, it is not implemented to expand both branches under call-by-value evaluation, but in the way that it behaves in practise: evaluating the condition and then one branch depending on the value of the condition.

Evaluating an expression is performed by passing the expression to the *eval* function along with an empty environment, *eval e* [ ].

4 Derivation

The derivation of the step counting function will be shown in detail for a subset of the language, for conciseness. We will consider expressions consisting of integers and addition, with values being integers,

$$\begin{array}{lcl} \textbf{data}\ \mathit{Expr} & = & \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Val}\ \mathit{Value} \\ \textbf{type}\ \mathit{Value} & = & \mathit{Int} \end{array}$$

An environment is not required for this subset of the language, so it will be left out. The starting evaluator is then simply,

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

4.1 Tail-recursive Evaluator

Our aim is to turn the evaluator in to an abstract machine; a first order, tail-recursive program. The evaluator is already first order, but it is not tail-recursive. It can be made so by transforming it in to continuation passing style (CPS). 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. The continuation function will take an argument of type *Value*, and its result is a *Value*,

$$\textbf{type}\ \mathit{Con} = \mathit{Value} \rightarrow \mathit{Value}$$

The new evaluator can be calculated from the old one by using the specification,

$$\mathit{evalTail}\ e\ c = c\ (\mathit{eval}\ e)$$

By induction on e,

Case : $e = \mathit{Val}\ v$

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

Case : $e = \mathit{Add}\ x\ y$

$$\begin{array}{ll} & \mathit{evalTail}\ (\mathit{Add}\ x\ y)\ c \\ = & \quad \{\ \text{specification}\ \} \\ & c\ (\mathit{eval}\ (\mathit{Add}\ x\ y)) \end{array}$$

$$
\begin{array}{ll}
= & \quad \{ \text{ definition of } \mathit{eval} \ \} \\
& c\ (\mathit{eval}\ x + \mathit{eval}\ y) \\
= & \quad \{ \text{ reverse beta-reduction, abstract over } \mathit{eval}\ x \ \} \\
& (\lambda m \rightarrow c\ (m + \mathit{eval}\ y))\ (\mathit{eval}\ x) \\
= & \quad \{ \text{ inductive assumption } \} \\
& \mathit{evalTail}\ x\ (\lambda m \rightarrow c\ (m + \mathit{eval}\ y)) \\
= & \quad \{ \text{ reverse beta-reduction, abstract over } \mathit{eval}\ y \ \} \\
& \mathit{evalTail}\ x\ (\lambda m \rightarrow (\lambda n \rightarrow c\ (m + n))\ (\mathit{eval}\ y)) \\
= & \quad \{ \text{ inductive assumption } \} \\
& \mathit{evalTail}\ x\ (\lambda m \rightarrow \mathit{evalTail}\ y\ (\lambda n \rightarrow c\ (m + n)))
\end{array}
$$

In conclusion, we have calculated the following recursive definition,

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

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

$$\mathit{eval}\ e = \mathit{evalTail}\ e\ (\lambda x \rightarrow x)$$

4.2 Abstract Machine

The next step is to make the evaluator first order. This is done by defunctionalising the continuation. At the moment the continuation is a function *Value* $\rightarrow$ *Value* but the whole function space is not required; the continuation function is only created in three different ways. Defunctionalisation 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, and has an *apply* function which does the same thing as the original function.

The data structure required is shown below. The names of the constructors will become clear later.

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

By definition the *apply* function needs to do the same for each instance of the continuation function,

$$
\begin{array}{lll}
\mathit{apply}\ \mathit{Top} & = & \lambda v \rightarrow v \\
\mathit{apply}\ (\mathit{AddR}\ m\ c) & = & \lambda n \rightarrow \mathit{apply}\ c\ (m + n) \\
\mathit{apply}\ (\mathit{AddL}\ c\ y) & = & \lambda m \rightarrow \mathit{evalTail}\ y\ (\mathit{apply}\ (\mathit{AddL}\ c\ m))
\end{array}
$$

Using the specification, $\mathit{evalMachine}\ e\ c = \mathit{evalTail}\ e\ (\mathit{apply}\ c)$, the function $\mathit{evalMachine}$ can be calculated by induction on e,

Case : $e = \mathit{Val}\ v$

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

Case : $e = \mathit{Add}\ x\ y$

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

We have now calculated the following recursive function,

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

Moving the lambda abstracted terms to the left and applying the specification in the AddL case, gives the following function,

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

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

$$\mathit{eval}\ e = \mathit{evalMachine}\ e\ \mathit{Top}$$

as shown below,

$$
\begin{array}{ll}
 & \mathit{eval}\ e \\
= & \quad \{\ \text{definition of } \mathit{eval}\ \} \\
 & \mathit{evalTail}\ e\ (\lambda v \rightarrow v) \\
= & \quad \{\ \text{definition of } \mathit{apply}\ \}
\end{array}
$$

$$\begin{aligned} & \mathit{evalTail}\ e\ (\mathit{apply}\ \mathit{Top}) \\ = & \quad \{\ \text{specification}\ \} \\ & \mathit{evalMachine}\ e\ \mathit{Top} \end{aligned}$$

The evaluator is now an abstract machine that has two states, for *eval* and *apply*. The reason for the constructor names can be revealed; the data structure is the structure of evaluation contexts for the language [7]. It can alternatively be viewed as a stack, pushing expressions still to be evaluated and values to be saved.

4.3 Step Counting Abstract Machine

The number of time steps required to evaluate an expression is going 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 *evalMachine* 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. Then 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.

```
type Step                          =  Int

stepMachine                        ::  (Expr, Step) → Cont → (Value, Step)
stepMachine (Val v, s) c           =  apply′ c (v, s + 1)
stepMachine (Add x y, s) c         =  stepMachine (x, s + 1) (AddL c y)

apply′                             ::  Cont → (Value, Step) → (Value, Step)
apply′ Top (v, s)                  =  (v, s + 1)
apply′ (AddL c y) (m, s)           =  stepMachine (y, s + 1) (AddR m c)
apply′ (AddR m c) (n, s)           =  apply′ c (m + n, s + 1)
```

The number of steps is initialised to zero and incremented every time the machine makes a transition. The evaluation function now returns a pair, where the first part is the evaluated value, and the second is the number of steps taken. The actual addition of the integers is a primitive operation and so happens in one step no matter what the size of the arguments. The semantics of the original evaluator can be recovered as,

```
eval e = fst (stepMachine e 0 Top) |
```

The aim now is to derive a function that counts the number of steps required to evaluate an expression; from the specification,

```
steps e = snd (stepMachine (e, 0) Top)
```

The first step in the reverse process is to refunctionalise the representation of the continuation. The original continuation was a function from $\mathit{Value} \rightarrow \mathit{Value}$, the new one will go from $(\mathit{Value}, \mathit{Step}) \rightarrow (\mathit{Value}, \mathit{Step})$.

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

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

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

The refunctionlised version is,

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

The new step counting function becomes,

$$\begin{array}{ll}
 & \mathit{steps}\ e \\
= & \quad \{\ \text{definition of } \mathit{steps}\ \} \\
 & \mathit{snd}\ (\mathit{stepMachine}\ (e, 0)\ \mathit{Top}) \\
= & \quad \{\ \text{specification}\ \} \\
 & \mathit{snd}\ (\mathit{stepTail}\ (e, 0)\ (\lambda(v, s) \rightarrow \mathit{apply}'\ \mathit{Top}\ v\ s)) \\
= & \quad \{\ \text{definition of } \mathit{apply}'\ \} \\
 & \mathit{snd}\ (\mathit{stepTail}\ (e, 0)\ ((v, s) \rightarrow (v, s+1)))
\end{array}$$

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

4.4 Evaluator with Accumulator

The step counting evaluator can be transformed from CPS back to direct style, by removing the continuation, to give a function with the type,

$$\mathit{stepAcc} :: (\mathit{Expr}, \mathit{Step}) \rightarrow (\mathit{Value}, \mathit{Step})$$

This is performed by induction on e, using the specification,

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

The resulting evaluator is,

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

The new step counting function becomes,

$$\begin{aligned}
& \mathit{steps}\ e \\
= & \quad \{\ \text{specification}\ \} \\
& \mathit{snd}\ (\mathit{stepTail}\ (e,0)\ ((v,s) \to (v,s+1))) \\
= & \quad \{\ \text{definition of } \mathit{stepTail}\ \} \\
& \mathit{snd}\ ((\lambda(v,s) \to (v,s+1))\ (\mathit{stepAcc}\ (e,0))) \\
= & \quad \{\ \text{lemma, } \mathit{snd}\ \} \\
& \mathit{snd}\ (\mathit{stepAcc}\ (e,0)) + 1
\end{aligned}$$

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

4.5 Step Counting Evaluator

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

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

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

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

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

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

4.6 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 where the result of evaluation is required.

This can be calculated by pushing through the definition of the *snd* function, to give,

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

The *steps* function is then simply,

$$\mathit{steps}\ e = \mathit{steps}'\ e + 1$$

5 Complete Function

Adding the rest of the language and the environment, the complete *steps′* function now looks like,

```
steps′ (Val v) env          = 1
steps′ (Var x) env          = 1
steps′ (Abs x e) env        = 1
steps′ (App f e) env        =
        steps′ f env + steps′ e env + steps′ e′ ((x, v) : env′) + 3
                where (Clo x e′ env′) = eval f env
                        v = eval e env
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 + case eval xs env of
        Nil → steps′ v env + 2
        ConsV y ys → steps′ f env + steps′ (Foldr (Val f′) v (Val ys)) env+
                steps′ (App (App f′ (Val y)) x) [] + 4
                  where f′ = Val (eval f env)
                        x = Val (eval (Foldr (Val f′) v (Val ys)) env)
```

The derived function shows the constant overheads involved in evaluation — for example, the number of steps needed to evaluate an addition is the sum of the number of steps to evaluate each side plus a constant three.

We want to be able to reason about how the time usage of some example functions depends on the size of the arguments to the function. In the case of fold-right functions, 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 a recursive function. In 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,

```
foldrVal                    :: (Value → b → b) → b → Value → b
foldrVal f v Nil            = v
foldrVal f v (ConsV x xs)   = f x (foldrVal f v xs)
```

Also, if the number of steps to apply the binary function f does not depend on the value of the arguments passed, such as adding two expressions, then this can be expressed as a function over the length of the list argument supplied,

```
lengthVal                   = foldrVal (λ_ n → n + 1) 0
```

6 Example Functions

The derived *steps* function can now be used on the examples to calculate the number of steps required in evaluating the function. QuickCheck [8] was used to test each function produced.

6.1 Summing a List

Below are two ways to sum a list of integers expressed using the fold-right operator,

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

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

The first replaces each : in the list with + and the unit of addition, 0, for the empty list case. The second has an accumulator; the *fold* is used to generate a function which is applied to the identity function in the *nil* case and in the *cons* case adds the current value to the accumulator.

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.

Sum

$$\begin{array}{l} \mathit{sum} = \mathit{Abs}\ \texttt{xs}\ (\mathit{Foldr}\ \mathit{add}\ (\mathit{Val}\ (\mathit{Const}\ 0))\ (\mathit{Var}\ \texttt{xs})) \\ \mathit{add} = \mathit{Abs}\ \texttt{x}\ (\mathit{Abs}\ \texttt{y}\ (\mathit{Add}\ (\mathit{Var}\ \texttt{x})\ (\mathit{Var}\ \texttt{y}))) \end{array}$$

The number of steps required to evaluate applying the *sum* function to a list *xs* is given below,

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

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.

Sum with an Accumulator

$$\begin{array}{l} \mathit{sumAcc} = \mathit{Abs}\ \texttt{xs}\ (\mathit{App}\ (\mathit{Foldr}\ f\ \mathit{idExpr}\ (\mathit{Var}\ \texttt{xs}))\ (\mathit{Val}\ (\mathit{Const}\ 0))) \\ \qquad\qquad \textbf{where}\ f = \mathit{Abs}\ \texttt{x}\ (\mathit{Abs}\ \texttt{g}\ (\mathit{Abs}\ \texttt{a}\ (\mathit{App}\ (\mathit{Var}\ \texttt{g})\ (\mathit{Add}\ (\mathit{Var}\ \texttt{a})\ (\mathit{Var}\ \texttt{x}))))) \\ \mathit{idExpr} = \mathit{Abs}\ \texttt{x}\ (\mathit{Var}\ \texttt{x}) \end{array}$$

The step count of the sum with an accumulator is 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 extra abstractions.

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

6.2 Concatenation

Concatenating a list of lists can be defined by folding the append function over the list,

$$concat\ xs = foldr\ append\ [\,]\ xs$$

where *append* is defined as,

$$append\ xs\ ys = foldr\ (\lambda z\ zs \to z : zs)\ ys\ xs$$

Append First we need to analyse the *append* function.

$$\begin{aligned}&append = Abs\ \texttt{xs}\ (Abs\ \texttt{ys}\ (Foldr\ f\ (Var\ \texttt{ys})\ (Var\ \texttt{xs})))\\&\qquad\textbf{where}\ f = Abs\ \texttt{z}\ (Abs\ \texttt{zs}\ (Cons\ (Var\ \texttt{z})\ (Var\ \texttt{zs})))\end{aligned}$$

The number of steps to evaluate the *append* function applied to two list arguments is given below.

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

The function is proportional to the length of the list that the first argument evaluates to.

Concat The step count of the *concat* function can now be calculated using this function,

$$concat = Abs\ \texttt{xss}\ (Foldr\ append\ (Val\ Nil)\ (Var\ \texttt{xss}))$$

With the step count from the *append* function inlines, the function is,

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

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* evaluates to 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,

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

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.

6.3 Reversing a List

Reversing a list can be expressed directly as a *fold* by appending the reversed tail of the list to the head element made in to a singleton list, as shown below,

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

The function can also be expressed using an accumulator,

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

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.

Reverse

$$\begin{array}{l} \textit{reverse} = \textit{Abs}\ \texttt{xs}\ (\textit{Foldr}\ f\ (\textit{Val Nil})\ (\textit{Var}\ \texttt{xs})) \\ \qquad \textbf{where}\ f = \textit{Abs}\ \texttt{x}\ (\textit{Abs}\ \texttt{xs}\ (\textit{App}\ (\textit{App append}\ (\textit{Var}\ \texttt{xs})) \\ \qquad\qquad\qquad (\textit{Cons}\ (\textit{Var}\ \texttt{x})\ (\textit{Val Nil})))) \end{array}$$

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

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 summation up to the length of the list,

$$8 + \sum_{x=0}^{\textit{length}\ xs - 1} 21x + 34 = 8 + \frac{(\textit{length}\ xs)(47 + 21(\textit{length}\ xs))}{2} \leqslant c\ (\textit{length}\ xs)^2$$

The summation is equivalent to the fraction in the middle (from Gauss' sum of a finite series) which 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 function's time requirements are quadratic on the length of the list [9].

Reverse with an Accumulator The version of reverse with an accumulator is expressed in the language as,

$$\begin{array}{l} \textit{reverseAcc} = \textit{Abs}\ \texttt{xs}\ (\textit{App}\ (\textit{Foldr}\ f\ \textit{idExpr}\ (\textit{Var}\ \texttt{xs}))\ (\textit{Val}\ (\textit{Const}\ 0))) \\ \qquad \textbf{where}\ f = \textit{Abs}\ \texttt{x}\ (\textit{Abs}\ \texttt{g}\ (\textit{Abs}\ \texttt{a}\ (\textit{App}\ (\textit{Var}\ \texttt{g})\ (\textit{Cons}\ (\textit{Var}\ \texttt{x})\ (\textit{Var}\ \texttt{a}))))) \end{array}$$

The *steps* function is proportional to the length of the list.

$$\textit{stpes}\ (\textit{App reverseAcc}\ (\textit{Val}\ xs)) = 26 * (\textit{lengthVal}\ xs) + 15$$

7 Conclusion and Further Work

We have outlined a process that takes an evaluator for a language, with a given evaluation strategy, and derives a function that will give an accurate count of how many steps are required to evaluate expressions in the language. The derivation itself is language and evaluation strategy non-specific. Using an extended lambda-calculus under call-by-value evaluation, the examples in the previous section give the appropriate complexity results that would be expected, 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 reverse function with or without an accumulator.

This work could be taken in a few directions. Looking at what happens with more complicated evaluation strategies, such as call-by-need/lazy evaluation, would be one next stage. It would also be interesting to apply the same technique to look at the space requirements for functions, as mentioned briefly earlier. A useful addition to this work would be to develop a calculus to automate deriving the step functions.

References

[1] 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.

[2] Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report RS-03-33, October 2003. 32 pp. This report supersedes the earlier BRICS report RS-02-53.

[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] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[5] Technical report.

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

[7] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. pages 13–23, 2004. Invited talk.

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

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