Worker/Wrapper/Makes it/Faster

Jennifer Hackett Graham Hutton
School of Computer Science, University of Nottingham
{jph,gmh}@cs.nott.ac.uk

Abstract

Much research in program optimization has focused on formal approaches to correctness: proving that the meaning of programs is preserved by the optimisation. Paradoxically, there has been comparatively little work on formal approaches to efficiency: proving that the performance of optimized programs is actually improved. This paper addresses this problem for a general-purpose optimization technique, the worker/wrapper transformation. In particular, we use the call-by-need variant of improvement theory to establish conditions under which the worker/wrapper transformation is formally guaranteed to preserve or improve the time performance of programs in lazy languages such as Haskell.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

Keywords general recursion; improvement

1. Introduction

To misquote Oscar Wilde [31], "functional programmers know the value of everything and the cost of nothing"[1]. More precisely, the functional approach to programming emphasises what programs *mean* in a denotational sense, rather than what programs *do* in terms of their operational behaviour. For many programming tasks this emphasis is entirely appropriate, allowing the programmer to focus on the high-level description of what is being computed rather than the low-level details of how this is realised. However, in the context of program optimisation both aspects play a central role, as the aim of optimisation is to improve the operational performance of programs while maintaining their denotational correctness.

A research paper on program optimisation therefore should justify both the correctness and performance aspects of the optimisation described. There is a whole spectrum of possible approaches to this, ranging from informal tests and benchmarks [19], to tool-based methods such as property-based testing [3] and space/time profiling [24], all the way up to formal mathematical proofs [17]. For correctness, it is now becoming standard to formally prove that an optimisation preserves the meaning of programs. For performance, however, the standard approach is to provide some form of empirical evidence that an optimisation improves the efficiency of programs, and there is little published work on formal proofs of improvement.

In this paper, we aim to go some way toward redressing this imbalance in the context of the *worker/wrapper transformation* [7], putting the denotational and operational aspects on an equally formal footing. The worker/wrapper transformation is a general purpose optimisation technique that has already been formally proved correct, as well as being realised in practice as an extension to the Glasgow Haskell Compiler [26]. In this paper we formally prove that this transformation is guaranteed to preserve or improve time performance with respect to an established operational theory. In other words, we show that the worker/wrapper transformation never makes programs slower. Specifically, the paper makes the following contributions:

- We show how Moran and Sands' work on *call-by-need improvement theory* [15] can be applied to formally justify that the worker/wrapper transformation for least fixed points preserves or improves time performance;
- We present preconditions that ensure the transformation improves performance in this manner, which come naturally from the preconditions that ensure correctness;
- We demonstrate the utility of the new theory by verifying that examples from previous worker/wrapper papers indeed exhibit a time improvement.

The use of *call-by-need* improvement theory means that our work applies to lazy functional languages such as Haskell. Traditionally, the operational beheaviour of lazy evaluation has been seen as difficult to reason about, but we show that with the right tools this need not be the case. To the best of our knowledge, this paper is the first time that a general purpose optimisation method for lazy languages has been formally proved to improve time performance.

Improvement theory does not seem to have attracted much attention in recent years, but we hope that this paper can help to generate more interest in this and other techniques for reasoning about lazy evaluation. Whereas in many papers calculations and proofs are often omitted or compressed for reasons of brevity, in this paper they are the central focus, so are presented in detail.

[1] The general form of this misquote is due to Alan Perlis, who originally said it of Lisp programmers.

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-2873-9 /14/09…$15.00.
http://dx.doi.org/10.1145/2628136.2628142

2. Example: Fast Reverse

We shall begin with an example that motivates the rest of the paper: transforming the naïve list reverse function into the so-called "fast reverse" function. This transformation is an instance of the worker/wrapper transformation, and there is an intuitive, informal justification of why this is an optimisation. Here we give this non-rigorous explanation; the remainder of this paper will focus on building the tools to strengthen this to a rigorous argument.

We start with a naïve definition of the reverse function, which takes quadratic time to run as each append $+\!\!+$ takes time linear in the length of its left argument:

$$\begin{aligned} &\mathit{reverse} && :: [a] \to [a] \\ &\mathit{reverse}\,[\,] && = [\,] \\ &\mathit{reverse}\,(x : \mathit{xs}) && = \mathit{reverse}\,\mathit{xs} +\!\!+ [x] \end{aligned}$$

We can write a more efficient version by using a *worker* function *revcat* with a *wrapper* around it that simply applies the worker function with $[\,]$ as the second argument:

$$\begin{aligned} &\mathit{reverse}' && :: [a] \to [a] \\ &\mathit{reverse}'\ \mathit{xs} && = \mathit{revcat}\,\mathit{xs}\,[\,] \end{aligned}$$

The specification for the worker *revcat* is as follows:

$$\begin{aligned} &\mathit{revcat} && :: [a] \to [a] \to [a] \\ &\mathit{revcat}\ \mathit{xs}\ \mathit{ys} && = \mathit{reverse}\,\mathit{xs} +\!\!+ \mathit{ys} \end{aligned}$$

From this specification we can calculate a new definition that does not depend on *reverse*. Because *reverse* is defined by cases, we will have one calculation for each case.

Case for $[\,]$:

$$\begin{aligned} & \mathit{revcat}\,[\,]\ \mathit{ys} \\ = \quad & \{ \text{ specification of } \mathit{revcat} \ \} \\ & \mathit{reverse}\,[\,] +\!\!+ \mathit{ys} \\ = \quad & \{ \text{ definition of } \mathit{reverse} \ \} \\ & [\,] +\!\!+ \mathit{ys} \\ = \quad & \{ \text{ definition of } +\!\!+ \ \} \\ & \mathit{ys} \end{aligned}$$

Case for $(x : \mathit{xs})$:

$$\begin{aligned} & \mathit{revcat}\,(x : \mathit{xs})\ \mathit{ys} \\ = \quad & \{ \text{ specification of } \mathit{revcat} \ \} \\ & \mathit{reverse}\,(x : \mathit{xs}) +\!\!+ \mathit{ys} \\ = \quad & \{ \text{ definition of } \mathit{reverse} \ \} \\ & (\mathit{reverse}\,\mathit{xs} +\!\!+ [x]) +\!\!+ \mathit{ys} \\ = \quad & \{ \text{ associativity of } +\!\!+ \ \} \\ & \mathit{reverse}\,\mathit{xs} +\!\!+ ([x] +\!\!+ \mathit{ys}) \\ = \quad & \{ \text{ definition of } +\!\!+ \ \} \\ & \mathit{reverse}\,\mathit{xs} +\!\!+ (x : \mathit{ys}) \\ = \quad & \{ \text{ specification of } \mathit{revcat} \ \} \\ & \mathit{revcat}\,\mathit{xs}\,(x : \mathit{ys}) \end{aligned}$$

Note the use of associativity of $+\!\!+$ in the third step, which is the only step not simply by definition or specification. Left-associated appends such as $(\mathit{xs} +\!\!+ \mathit{ys}) +\!\!+ \mathit{zs}$ are less time-efficient than the equivalent right-associated appends $\mathit{xs} +\!\!+ (\mathit{ys} +\!\!+ \mathit{zs})$, as the former traverses *xs* twice. The intuition here is that the efficiency gain from this step in the proof carries over in some way to the rest of the proof, so that overall our calculated definition of *revcat* is more efficient than its original specification. The calculation gives us the following definition, which runs in linear time:

$$\begin{aligned} &\mathit{reverse}\ \mathit{xs} && = \mathit{revcat}\,\mathit{xs}\,[\,] \\ &\mathit{revcat}\,[\,]\ \mathit{ys} && = \mathit{ys} \\ &\mathit{revcat}\,(x : \mathit{xs})\ \mathit{ys} && = \mathit{revcat}\,\mathit{xs}\,(x : \mathit{ys}) \end{aligned}$$

Unfortunately, there are a number of problems with this approach. Firstly, we calculated *revcat* using the *fold-unfold* style of program calculation [2]. This is an informal calculation, which fails to guarantee total correctness. Thus the resulting *reverse* function may fail in some cases where the original succeeded. Secondly, while we are applying the common pattern of factorising a program into a worker and a wrapper, the reasoning we use is ad-hoc and does not take advantage of this. We would like to abstract out this pattern to make future applications of this technique more straightforward. Finally, while intuitively we can see an efficiency gain from the use of associativity of $+\!\!+$, this is not a rigorous argument. Put simply, we need rigorous proofs of both *correctness* and *improvement* for our transformation.

3. Worker/Wrapper Transformation

The worker/wrapper transformation, as originally formulated by Gill and Hutton [7], allowed a function written using general recursion to be split into a recursive *worker* function and a *wrapper* function that allows the new definition to be used in the same contexts as the original. The usual application of this technique would be to write the worker to use a different type than the original program that supports more efficient operations, thus hopefully resulting in a more efficient program overall. Gill and Hutton gave conditions for the correctness of the transformation; here we present the more general theory and correctnesss conditions recently developed by Sculthorpe and Hutton [25].

3.1 The Fix Theory

The idea of the worker/wrapper transformation for fixed-points is as follows. Given a recursive program *prog* of some type A, we can write *prog* as some function f of itself:

$$\begin{aligned} &\mathit{prog} :: A \\ &\mathit{prog} = f\ \mathit{prog} \end{aligned}$$

We can rewrite this definition so that it is explicitly written using the well-known fixpoint operator fix:

$$\begin{aligned} &\mathrm{fix} && :: (a \to a) \to a \\ &\mathrm{fix}\ f && = f\,(\mathrm{fix}\ f) \end{aligned}$$

resulting in the following definition:

$$\mathit{prog} = \mathrm{fix}\ f$$

Next, we write functions $\mathit{abs} :: B \to A$ and $\mathit{rep} :: A \to B$ that allow us to convert from the original type A to some other type B that supports more efficient operations. We finish by constructing a new function $g : B \to B$ that allows us to rewrite our original definition of *prog* as follows:

$$\mathit{prog} = \mathit{abs}\ (\mathrm{fix}\ g)$$

Here *abs* is the *wrapper* function, while fix g is the *worker*. The pattern of the worker/wrapper transformation can be captured by a theorem that expresses necessary and sufficient conditions for its correctness [25]. This theorem has assumptions that express the required relationship between the functions *abs* and *rep*, and conditions that provide a specification for the function g in terms of *abs*, *rep* and f:

Theorem 1 (Worker/Wrapper Factorisation)**.**

Given

$$abs : B \to A \quad f : A \to A$$
$$rep : A \to B \quad g : B \to B$$

satisfying one of the assumptions

$$(A)\ abs \circ rep = id_A$$
$$(B)\ abs \circ rep \circ f = f$$
$$(C)\ \text{fix}\ (abs \circ rep \circ f) = \text{fix}\ f$$

and one of the conditions

$$(1)\ g = rep \circ f \circ abs \quad (1\beta)\ \text{fix}\ g = \text{fix}\ (rep \circ f \circ abs)$$
$$(2)\ g \circ rep = rep \circ f \quad (2\beta)\ \text{fix}\ g = rep\ (\text{fix}\ f)$$
$$(3)\ f \circ abs = abs \circ g$$

we have the factorisation

$$\text{fix}\ f = abs\ (\text{fix}\ g)$$

The different assumptions and conditions allow one to choose which will be easiest to verify.

3.2 Proving Fast Reverse Correct

Recall once again the naïve definition of *reverse*:

$$\begin{aligned} &reverse && :: [a] \to [a] \\ &reverse\ [\,] && = [\,] \\ &reverse\ (x : xs) && = reverse\ xs +\!\!+ [x] \end{aligned}$$

As we mentioned before, this naïve implementation is inefficient due to the use of the append operation $+\!\!+$. We would like to use worker/wrapper factorisation to improve it. The first step is to rewrite the function using fix:

$$\begin{aligned} &reverse && = \text{fix}\ rev \\ &rev && :: ([a] \to [a]) \to ([a] \to [a]) \\ &rev\ r\ [\,] && = [\,] \\ &rev\ r\ (x : xs) && = r\ xs +\!\!+ [x] \end{aligned}$$

The next step in applying worker/wrapper is to select a new type to replace the original type $[a] \to [a]$, and to write *abs* and *rep* functions to perform the conversions. We can represent a list *xs* by its *difference list* $\lambda ys \to xs +\!\!+ ys$, as first demonstrated by Hughes [12]. Difference lists have the advantage that the usually costly operation of $+\!\!+$ can be implemented with function composition, typically leading to an increase of efficiency. We write the following functions to convert between the two representations:

$$\begin{aligned} &\textbf{type}\ DiffList\ a = [a] \to [a] \\ &toDiff && :: [a] \to DiffList\ a \\ &toDiff\ xs && = \lambda ys \to xs +\!\!+ ys \\ &fromDiff && :: DiffList\ a \to [a] \\ &fromDiff\ h && = h\ [\,] \end{aligned}$$

We have $fromDiff \circ toDiff = id$:

$$\begin{aligned} & fromDiff\ (toDiff\ xs) \\ =\ & \{\ \text{definition of } toDiff\ \} \\ & fromDiff\ (\lambda ys \to xs +\!\!+ ys) \\ =\ & \{\ \text{definition of } fromDiff\ \} \\ & (\lambda ys \to xs +\!\!+ ys)\ [\,] \\ =\ & \{\ \beta\text{-reduction}\ \} \\ & xs +\!\!+ [\,] \\ =\ & \{\ [\,] \text{ is identity of } +\!\!+\ \} \\ & xs \end{aligned}$$

From these functions it is straightforward to create the actual *abs* and *rep* functions. These convert between the original function type $[a] \to [a]$ and a new function type $[a] \to DiffList\ a$ where the returned value is represented as a difference list, rather than a regular list:

$$\begin{aligned} &rep && :: ([a] \to [a]) \to ([a] \to DiffList\ a) \\ &rep\ h && = toDiff \circ h \end{aligned}$$

$$\begin{aligned} &abs && :: ([a] \to DiffList\ a) \to ([a] \to [a]) \\ &abs\ h && = fromDiff \circ h \end{aligned}$$

Assumption (A) holds trivially:

$$\begin{aligned} & abs\ (rep\ h) \\ =\ & \{\ \text{definitions of } abs \text{ and } rep\ \} \\ & fromDiff \circ toDiff \circ h \\ =\ & \{\ fromDiff \circ toDiff = id\ \} \\ & h \end{aligned}$$

Now we must verify that the definition of *revcat* that we calculated in the previous section

$$\begin{aligned} &revcat\ [\,]\ ys && = ys \\ &revcat\ (x : xs)\ ys && = revcat\ xs\ (x : ys) \end{aligned}$$

satisfies one of the worker/wrapper conditions. We first rewrite *revcat* as an explicit fixed point.

$$\begin{aligned} &revcat && = \text{fix}\ rev' \\ &rev'\ h\ [\,]\ ys && = ys \\ &rev'\ h\ (x : xs)\ ys && = h\ xs\ (x : ys) \end{aligned}$$

We now verify condition (2), $rev' \circ rep = rep \circ rev$, which expands to $rev'\ (rep\ r)\ xs = rep\ (rev\ r)\ xs$. We calculate from the right-hand side, performing case analysis on *xs*. Firstly, we calculate for the case when *xs* is empty:

$$\begin{aligned} & rep\ (rev\ r)\ [\,] \\ =\ & \{\ \text{definition of } rep\ \} \\ & toDiff\ (rev\ r\ [\,]) \\ =\ & \{\ \text{definition of } rev\ \} \\ & toDiff\ [\,] \\ =\ & \{\ \text{definition of } toDiff\ \} \\ & \lambda ys \to [\,] +\!\!+ ys \\ =\ & \{\ [\,] \text{ is identity of } +\!\!+\ \} \\ & \lambda ys \to ys \\ =\ & \{\ \text{definiton of } rev'\ \} \\ & rev'\ (rep\ r)\ [\,] \end{aligned}$$

and then for the case where *xs* is non-empty:

$$\begin{aligned} & rep\ (rev\ r)\ (x : xs) \\ =\ & \{\ \text{definition of } rep\ \} \\ & toDiff\ (rev\ r\ (x : xs)) \\ =\ & \{\ \text{definition of } rev\ \} \\ & toDiff\ (r\ xs +\!\!+ [x]) \\ =\ & \{\ \text{definition of } toDiff\ \} \\ & \lambda ys \to (r\ xs +\!\!+ [x]) +\!\!+ ys \\ =\ & \{\ \text{associativity and definition of } +\!\!+\ \} \\ & \lambda ys \to r\ xs +\!\!+ (x : ys) \\ =\ & \{\ \text{definition of } toDiff\ \} \\ & \lambda ys \to toDiff\ (r\ xs)\ (x : ys) \\ =\ & \{\ \text{definition of } rep\ \} \\ & \lambda ys \to rep\ r\ xs\ (x : ys) \\ =\ & \{\ \text{definition of } rev'\ \} \\ & rev'\ (rep\ r)\ (x : xs) \end{aligned}$$

For total correctness on infinite lists we must also verify the condition holds for the undefined value $\bot$:

$$
\begin{array}{ll}
& \mathit{rep}\ (\mathit{rev}\ r)\ \bot \\
= & \quad \{\ \text{definition of } \mathit{rep}\ \} \\
& \mathit{toDiff}\ (\mathit{rev}\ r\ \bot) \\
= & \quad \{\ \mathit{rev} \text{ pattern matches on second argument}\ \} \\
& \mathit{toDiff}\ \bot \\
= & \quad \{\ \text{definition of } \mathit{toDiff}\ \} \\
& \lambda ys \to \bot +\!\!+ ys \\
= & \quad \{\ +\!\!+ \text{ strict in first argument}\ \} \\
& \lambda ys \to \bot \\
= & \quad \{\ \mathit{rev}' \text{ pattern matches on second argument}\ \} \\
& \mathit{rev}'\ (\mathit{rep}\ r)\ \bot
\end{array}
$$

Now that we know our rev' satisfies condition (2), we have a new definition of *reverse*

$$\mathit{reverse} = \mathit{abs}\ \mathit{revcat} = \mathit{fromDiff} \circ \mathit{revcat}$$

which eta-expands as follows:

$$
\begin{array}{ll}
\mathit{reverse}\ xs & = \mathit{revcat}\ xs\ [\,] \\
\mathit{revcat}\ [\,]\ ys & = ys \\
\mathit{revcat}\ (x : xs)\ ys & = \mathit{revcat}\ xs\ (x : ys)
\end{array}
$$

The end result is the same improved definition of *reverse* we had before. Thus the worker/wrapper theory has allowed us to formally verify the correctness of our earlier transformation. Furthermore, the use of a general theory has allowed us to avoid the need for induction which would usually be needed to reason about recursive definitions.

4. Improvement Theory

Thus far we have only reasoned about correctness. In order to develop a worker/wrapper theory that can prove *efficiency* properties, we need an operational theory of *program improvement*. More than just expressing extensional information, this should be based on intensional properties of resources that a program requires. For the purpose of this paper, the resource we shall consider is execution time.

We have two main design goals for our operational theory. Firstly, it ought to be based on the operational semantics of a realistic programming language, so that conclusions we draw from it are as applicable as possible. Secondly, it should be amenable to techniques such as (in)equational reasoning, as these are the techniques we used to apply the worker/wrapper correctness theory.

For the first goal, we use a language with similar syntax and semantics to GHC Core, except that arguments to functions are required to be atomic, as was the case in earlier versions of the language [20]. (Normalisation of the current version of GHC Core into this form is straightforward.) The language is call-by-need, reflecting the use of lazy evaluation in Haskell. The efficiency behaviour of call-by-need programs is notoriously counterintuitive. Our hope is that providing formal techniques for reasoning about call-by-need efficiency we will go some way toward easing this problem.

For the second goal, our theory must be based around relation R that is a preorder, as transitivity and reflexivity are necessary for inequational reasoning to be valid. Furthermore, to support reasoning in a compositional manner, it is essential to allow substitution. That is, given terms M and N, if $M\ R\ N$ then $\mathbb{C}[M]\ R\ \mathbb{C}[N]$ should also hold for any context $\mathbb{C}$. A relation R that satisfies both of these properties is called a *precongruence*.

A naïve approach to measuring execution time would be to simply count the number of steps taken to evaluate a term to some normal form, and consider that a term M is more efficient than a term N if its evaluation finishes in fewer steps. The resulting relation is clearly a preorder; however it is not a precongruence in a call-by-need setting, because meaningful computations can be done with terms that are not fully normalised. For example, just because M normalises and N does not, it does not follow that M is necessarily more efficient in *all* contexts.

The approach we use is due to Moran and Sands [15]. Rather than counting the steps taken to normalise a term, we compare the steps taken in *all* contexts, and only say that M is improved by N if for any context $\mathbb{C}$, the term $\mathbb{C}[M]$ requires no more evaluation steps than the term $\mathbb{C}[N]$. The result is a relation that is trivially a precongruence: it inherits transitivity and reflexivity from the numerical ordering $\leqslant$, and is substitutive by definition.

Improvement theory [23] was originally developed for call-by-name languages by Sands [21]. The remainder of this section presents the call-by-need time improvement theory due to Moran and Sands [15], which will provide the setting for our operational worker/wrapper theory. The essential difference between call-by-name and call-by-need is that the latter implements a *sharing* strategy, avoiding the repeated evaluation of terms that are used more than once.

4.1 Operational Semantics of the Core Language

We shall begin by presenting the operational model that forms the basis of this improvement theory. The semantics presented here are originally due to Sestoft [27].

We start from a set of variables Var and a set of constructors Con. We assume all constructors have a fixed arity. The grammar of terms is as follows:

$$
\begin{array}{lcl}
x, y, z & \in & Var \\
c & \in & Con \\
M, N & ::= & x \\
& \mid & \lambda x \to M \\
& \mid & M\ x \\
& \mid & \mathbf{let}\ \{\vec{x} = \vec{M}\}\ \mathbf{in}\ N \\
& \mid & c\ \vec{x} \\
& \mid & \mathbf{case}\ M\ \mathbf{of}\ \{c_i\ \vec{x_i} \to N_i\}
\end{array}
$$

We use $\vec{x} = \vec{M}$ as a shorthand for a list of bindings of the form $x = M$. Similarly, we use $c_i\ \vec{x_i} \to N_i$ as a shorthand for a list of cases of the form $c\ \vec{x} \to N$. All constructors are assumed to be saturated, that is, we assume that any $\vec{x}$ that is the operand of a constructor c has length equal to the arity of c. Literals are represented by constructors of arity 0. We treat α-equivalent terms as identical.

A term is a *value* if it is of the form $c\ \vec{x}$ or $\lambda x \to M$. In Haskell this is referred to as a *weak head normal form*. We shall use letters such as V, W to denote value terms.

Term contexts take the following form, with substitution defined in the obvious way.

$$
\begin{array}{lcl}
\mathbb{C}, \mathbb{D} & ::= & [-] \\
& \mid & x \\
& \mid & \lambda x \to \mathbb{C} \\
& \mid & \mathbb{C}\ x \\
& \mid & \mathbf{let}\ \{\vec{x} = \vec{\mathbb{C}}\}\ \mathbf{in}\ \mathbb{D} \\
& \mid & c\ \vec{x} \\
& \mid & \mathbf{case}\ \mathbb{C}\ \mathbf{of}\ \{c_i\ \vec{x_i} \to \mathbb{D}_i\}
\end{array}
$$

A value context is a context that is either a lambda abstraction or a constructor applied to variables.

The restriction that the arguments of functions and constructors always be variables has the effect that all bindings

$$
\begin{array}{lll}
\langle \Gamma\ \{x = M\}, x, S\rangle & \to \langle \Gamma, M, \#x : S\rangle & \{\ \textsc{Lookup}\ \} \\
\langle \Gamma, V, \#x : S\rangle & \to \langle \Gamma\ \{x = V\}, V, S\rangle & \{\ \textsc{Update}\ \} \\
\langle \Gamma, M\ x, S\rangle & \to \langle \Gamma, M, x : S\rangle & \{\ \textsc{Unwind}\ \} \\
\langle \Gamma, \lambda x \to M, y : S\rangle & \to \langle \Gamma, M\ [y\ /\ x], S\rangle & \{\ \textsc{Subst}\ \} \\
\langle \Gamma, \mathbf{case}\ M\ \mathbf{of}\ alts, S\rangle & \to \langle \Gamma, M, alts : S\rangle & \{\ \textsc{Case}\ \} \\
\langle \Gamma, c_j\ \vec{y}, \{c_i\ \vec{x_i} \to N_i\} : S\rangle & \to \langle \Gamma, N_j\ [\vec{y}\ /\ \vec{x_j}], S\rangle & \{\ \textsc{Branch}\ \} \\
\langle \Gamma, \mathbf{let}\ \{\vec{x} = \vec{M}\}\ \mathbf{in}\ N, S\rangle & \to \langle \Gamma\ \{\vec{x} = \vec{M}\}, N, S\rangle & \{\ \textsc{Letrec}\ \}
\end{array}
$$

Figure 1. The call-by-need abstract machine

made during evaluation must have been created by a **let**. Sometimes we will use $M\ N$ (where N is not a variable) as a shorthand for $\mathbf{let}\ \{x = N\}\ \mathbf{in}\ M\ x$, where x is fresh. We use this shorthand for both terms and contexts.

An abstract machine for executing terms in the language maintains a state $\langle \Gamma, M, S\rangle$ consisting of: a heap Γ, given by a set of bindings from variables to terms; the term M currently being evaluated; the evaluation stack S, given by a list of tokens used by the abstract machine. The machine works by evaluating the current term to a value, and then decides what to do with the value based on the top of the stack. Bindings generated by **let** constructs are put on the heap, and only taken off when performing a Lookup. A Lookup executes by putting a token on the stack representing where the term was looked up, and then evaluating that term to value form before replacing it on the heap. In this way, each binding is only ever evaluated at most once. The semantics of the machine is given in Figure 1. Note that the Letrec rule assumes that $\vec{x}$ is disjoint from the domain of Γ; if not, we need only α-rename so that this is the case.

4.2 The Cost Model and Improvement Relations

Now that we have a semantics for our model, we must devise a *cost model* for this semantics. The natural way to do this for an operational semantics is to count steps taken to evaluate a given term. We use the notation $M{\downarrow^n}$ to mean the abstract machine progresses from the initial state $\langle \emptyset, M, \epsilon\rangle$ to some final state $\langle \Gamma, V, \epsilon\rangle$ with n occurences of the Lookup step. It is sufficient to count Lookup steps because the total number of steps is bounded by a linear function of the number of Lookup steps [15]. Furthermore, we use the notation $M{\downarrow^{\leqslant n}}$ to mean that $M{\downarrow^m}$ for some $m \leqslant n$.

From this, we can define our improvement relation. We say that "M is *improved* by N", written $M \gtrsim N$, if the following statement holds for all contexts $\mathbb{C}$:

$$\mathbb{C}[M]{\downarrow^m} \Longrightarrow \mathbb{C}[N]{\downarrow^{\leqslant m}}$$

In other words, a term M is improved by a term N if N takes no more steps to evaluate than M in all contexts. That this relation is a congruence follows immediately from the definition, and that it is a preorder follows from the fact that $\leqslant$ is itself a preorder. We sometimes write $M \lesssim N$ for $N \gtrsim M$. If both $M \gtrsim N$ and $M \lesssim N$, we write $M \mathrel{\underset{\sim}{\lhd\rhd}} N$ and say that M and N are *cost-equivalent*.

For convenience, we define a "tick" operation on terms that adds exactly one unit of cost to a term:

$$\checkmark M \equiv \mathbf{let}\ \{x = M\}\ \mathbf{in}\ x \quad \{\ \text{where } x \text{ is free in } M\ \}$$

This definition for $\checkmark M$ takes exactly two steps to evaluate to M: one to add the binding to the heap, and the other to look it up. Only one of these steps is a Lookup step, so the result is that the cost of evaluating the term is increased by exactly one. Using ticks allows us to annotate terms with individual units of cost, allowing us to use rules to "push" cost around a term, making the calculations more convenient. We could also define the tick operation by adding it to the grammar of terms and modifying the abstract machine and cost model accordingly, but this definition is equivalent. We have the following law: $\checkmark M \gtrsim M$.

The improvement relation $\gtrsim$ covers when one term is at least as efficient as another in all contexts, but this is a very strong statement. We use the notion of "weak improvement" when one term is at least as efficient as another within a constant factor. Specifically, we say M is weakly improved by N, written $M \mathrel{\underset{\sim}{\gtrsim}} N$, if there exists a linear function $f(x) = kx + c$ (where $k, c \geqslant 0$) such that the following statement holds for all contexts $\mathbb{C}$:

$$\mathbb{C}[M]{\downarrow^m} \Longrightarrow \mathbb{C}[N]{\downarrow^{\leqslant f(m)}}$$

This can be read as "replacing M with N may make programs worse, but cannot make them *asymptotically* worse". We use symbols $\mathrel{\underset{\sim}{\lesssim}}$ and $\mathrel{\underset{\sim}{\lhd\rhd}}$ for inverse and equivalence analogously as for standard $\sim$improvement.

Because weak improvement ignores constant factors, we have the following *tick introduction/elimination* law:

$$M \mathrel{\underset{\sim}{\lhd\rhd}} \checkmark M$$

It follows from this that any improvement $M \gtrsim N$ can be *weakened* to a weak improvement $M' \mathrel{\underset{\sim}{\gtrsim}} N'$ where M' and N' denote the terms M and N with all the ticks removed.

The last notation we define is *entailment*, which is used when we have a chain of improvements that all apply with respect to a particular set of definitions. Specifically, where $\Gamma = \{\vec{x} = \vec{V}\}$ is a list of bindings, we write:

$$\Gamma \vdash M_1 \gtrsim M_2 \gtrsim \ldots \gtrsim M_n$$

to mean:

$$\mathbf{let}\ \Gamma\ \mathbf{in}\ M_1 \gtrsim \mathbf{let}\ \Gamma\ \mathbf{in}\ M_2 \gtrsim \ldots \gtrsim \mathbf{let}\ \Gamma\ \mathbf{in}\ M_n$$

4.3 Selected Laws

We finish this section with a selection of laws taken from [15]. The first two are β-reduction rules. The following cost equivalence holds for function application:

$$(\lambda x \to M)\ y \mathrel{\underset{\sim}{\lhd\rhd}} M\ [y\ /\ x]$$

This holds because the abstract machine evaluates the left-hand-side to the right-hand-side without performing any Lookups, resulting the same heap and stack as before. Note that the substitution is variable-for-variable, as the grammar for our language requires that the argument to function application always be a variable.

In general, where a term M can be evaluated to a term M', we have the following relationships:

$$M \mathrel{\underset{\sim}{\gtrsim}} M'$$
$$M' \mathrel{\underset{\sim}{\lhd\rhd}} M$$

The latter fact may be non-obvious, but it holds because evaluating a term will produce a constant number of ticks, and tick-elimination is a weak cost-equivalence. In this manner we can see that partial evaluation by itself will never save more than a constant-factor of time.

The following cost equivalence allows us to substitute a variable for its binding. However, note that this is only valid for *values*, as bindings to other terms will be modified in the course of execution. We thus call this rule *value-β*.

$$\begin{array}{l}\quad \textbf{let}\ \{x = V, \vec{y} = \vec{\mathbb{C}}[x]\}\ \textbf{in}\ \mathbb{D}[x] \\ \lessapprox \\ \quad \textbf{let}\ \{x = V, \vec{y} = \vec{\mathbb{C}}[\checkmark V]\}\ \textbf{in}\ \mathbb{D}[\checkmark V]\end{array}$$

The following law allows us to move let bindings in and out of a context when the binding is to a value. Note that we assume that x does not appear free in $\mathbb{C}$, which can be ensured by α-renaming, and that no free variables in V are captured in $\mathbb{C}$. We call this rule *value let-floating*.

$$\mathbb{C}[\textbf{let}\ \{x = V\}\ \textbf{in}\ M] \lessapprox \textbf{let}\ \{x = V\}\ \textbf{in}\ \mathbb{C}[M]$$

We also have a *garbage collection* law allowing us to remove unused bindings. Assuming that x is not free in $\vec{N}$ or L, we have the following cost equivalence:

$$\textbf{let}\ \{x = M; \vec{y} = \vec{N}\}\ \textbf{in}\ L \lessapprox \textbf{let}\ \{\vec{y} = \vec{N}\}\ \textbf{in}\ L$$

The final law we present here is the rule of *improvement induction*. The version that we present is stronger than the version in [15], but can be obtained by a simple modification of the proof given there. For any set of value bindings Γ and context $\mathbb{C}$, we have the following rule:

$$\frac{\Gamma \vdash M \gtrsim \checkmark\mathbb{C}[M] \qquad \Gamma \vdash \checkmark\mathbb{C}[N] \gtrsim N}{\Gamma \vdash M \gtrsim N}$$

This allows us to prove an $M \gtrsim N$ simply by finding a context $\mathbb{C}$ where we can "unfold" M to $\checkmark\mathbb{C}[M]$ and "fold" $\checkmark\mathbb{C}[N]$ to N. In other words, the following proof is valid:

$$\begin{array}{l}\Gamma \vdash M \\ \quad \gtrsim \\ \quad\quad \checkmark\mathbb{C}[M] \\ \quad \gtrsim \quad \{\ \text{hypothesis}\ \} \\ \quad\quad \checkmark\mathbb{C}[N] \\ \quad \gtrsim \\ \quad\quad N\end{array}$$

In this way the technique is similar to proof principles such as guarded coinduction [4, 28].

As a corollary to this law, we have the following law for *cost-equivalence* improvement induction. For any set of value bindings Γ and context $\mathbb{C}$, we have:

$$\frac{\Gamma \vdash M \lessapprox \checkmark\mathbb{C}[M] \qquad \Gamma \vdash \checkmark\mathbb{C}[N] \lessapprox N}{\Gamma \vdash M \lessapprox N}$$

The proof is simply to start from the assumptions and make two applications of improvement induction: first to prove $M \gtrsim N$, and second to prove $N \gtrsim M$.

5. Worker/Wrapper and Improvement

In this section, we prove a factorisation theorem for improvement theory analogous to the worker/wrapper factorisation theorem given in section 3.1. Before we do this, however, we must prove two preliminary results: a *rolling rule* and a *fusion rule*. Rolling and fusion are central to the worker/wrapper transformation [7, 13], so it is only natural that we would need versions of these to apply worker/wrapper transformation in this context.

5.1 Preliminary Results

The first rule we prove is the *rolling rule*, so named because of its similarity to the rolling rule for least-fixed points. In particular, for any pair of value contexts $\mathbb{F}$, $\mathbb{G}$, we have the following weak cost equivalence:

$$\textbf{let}\ \{x = \mathbb{F}[\mathbb{G}[x]]\}\ \textbf{in}\ \mathbb{G}[x] \lessapprox \textbf{let}\ \{x = \mathbb{G}[\mathbb{F}[x]]\}\ \textbf{in}\ x$$

The proof begins with an application of cost-equivalence improvement induction. We let $\Gamma = \{x = \mathbb{F}[\checkmark\mathbb{G}[x]], y = \mathbb{G}[\checkmark\mathbb{F}[y]]\}$, $M = \checkmark\mathbb{G}[x]$, $N = y$, $\mathbb{C} = \mathbb{G}[\checkmark\mathbb{F}[-]]$. The premises of induction are proved as follows:

$$\begin{array}{l}\Gamma \vdash M \\ \quad \equiv \quad \{\ \text{definitions}\ \} \\ \quad\quad \checkmark\mathbb{G}[x] \\ \quad \lessapprox \quad \{\ \text{value-}\beta\ \} \\ \quad\quad \checkmark\mathbb{G}[\checkmark\mathbb{F}[\checkmark\mathbb{G}[x]]] \\ \quad \equiv \quad \{\ \text{definitions}\ \} \\ \quad\quad \checkmark\mathbb{C}[M]\end{array}$$

and

$$\begin{array}{l}\Gamma \vdash \checkmark\mathbb{C}[N] \\ \quad \equiv \quad \{\ \text{definitions}\ \} \\ \quad\quad \checkmark\mathbb{G}[\checkmark\mathbb{F}[y]] \\ \quad \lessapprox \quad \{\ \text{value-}\beta\ \} \\ \quad\quad y \\ \quad \equiv \quad \{\ \text{definitions}\ \} \\ \quad\quad N\end{array}$$

Thus we can conclude $\Gamma \vdash M \lessapprox N$, or equivalently $\textbf{let}\ \Gamma\ \textbf{in}\ M \lessapprox \textbf{let}\ \Gamma\ \textbf{in}\ N$. We expand this out and apply garbage collection to remove the unused bindings:

$$\textbf{let}\ \{x = \mathbb{F}[\checkmark\mathbb{G}[x]]\}\ \textbf{in}\checkmark\mathbb{G}[x] \lessapprox \textbf{let}\ \{y = \mathbb{G}[\checkmark\mathbb{F}[y]]\}\ \textbf{in}\ y$$

By applying α-renaming and weakening we obtain the desired result. The second rule we prove is *letrec-fusion*, analogous to fixed-point fusion. For any value contexts $\mathbb{F}$, $\mathbb{G}$, we have the following implication:

$$\begin{array}{l}\quad \mathbb{H}[\checkmark\mathbb{F}[x]] \gtrsim \mathbb{G}[\checkmark\mathbb{H}[x]] \\ \Rightarrow \\ \quad \textbf{let}\ \{x = \mathbb{F}[x]\}\ \textbf{in}\ \mathbb{H}[x] \gtrsim \textbf{let}\ \{x = \mathbb{G}[x]\}\ \textbf{in}\ x\end{array}$$

For the proof, we assume the premise and proceed by improvement induction. Let $\Gamma = \{x = \mathbb{F}[x], y = \mathbb{G}[y]\}$, $M = \checkmark\mathbb{H}[x]$, $N = y$, $\mathbb{C} = \mathbb{G}$. The premises are proved by:

$$\begin{array}{l}\Gamma \vdash M \\ \quad \equiv \quad \{\ \text{by definitions}\ \} \\ \quad\quad \checkmark\mathbb{H}[x] \\ \quad \lessapprox \quad \{\ \text{value beta}\ \} \\ \quad\quad \checkmark\mathbb{H}[\checkmark\mathbb{F}[x]] \\ \quad \gtrsim \quad \{\ \text{by assumption}\ \} \\ \quad\quad \checkmark\mathbb{G}[\checkmark\mathbb{H}[x]] \\ \quad \equiv \quad \{\ \text{definition}\ \} \\ \quad\quad \checkmark\mathbb{C}[M]\end{array}$$

and

$$\begin{array}{l}\Gamma \vdash \checkmark\mathbb{C}[N] \\ \quad \equiv \quad \{\ \text{by definitions}\ \} \\ \quad\quad \checkmark\mathbb{G}[y] \\ \quad \lessapprox \quad \{\ \text{value beta}\ \} \\ \quad\quad y \\ \quad \equiv \quad \{\ \text{definition}\ \} \\ \quad\quad N\end{array}$$

Thus we conclude that $\Gamma \vdash M \gtrsim N$. Expanding and applying garbage collection, we obtain the following:

$$\textbf{let } \{x = \mathbb{F}[x]\} \textbf{ in} \checkmark\mathbb{H}[x] \gtrsim \textbf{let } y = \mathbb{G}[y] \textbf{ in } y$$

Again we obtain the desired result via weakening and α-renaming. As improvement induction is symmetrical, we can also prove the following dual fusion law, in which the improvement relations are reversed:

$$\begin{array}{l} \mathbb{H}[\checkmark\mathbb{F}[x]] \lesssim \mathbb{G}[\checkmark\mathbb{H}[x]] \\ \Rightarrow \\ \textbf{let } \{x = \mathbb{F}[x]\} \textbf{ in } \mathbb{H}[x] \lessapprox \textbf{let } \{x = \mathbb{G}[x]\} \textbf{ in } x \end{array}$$

For both the rolling and fusion rules, we first proved a version of the conclusion with normal improvement, and then weakened to weak improvement. We do this to avoid having to deal with ticks, and because the weaker version is strong enough for our purposes.

Moran and Sands also prove their own fusion law. This law requires that the context $\mathbb{H}$ satisfy a form of *strictness*. Specifically, For any value contexts $\mathbb{F}$, $\mathbb{G}$ and fresh variable x, we have the following implication:

$$\begin{array}{l} \mathbb{H}[\mathbb{F}[x]] \gtrsim \mathbb{G}[\mathbb{H}[x]] \wedge \mathit{strict}\,(\mathbb{H}) \\ \Rightarrow \\ \textbf{let } \{x = \mathbb{F}[x]\} \textbf{ in } \mathbb{C}[\mathbb{H}[x]] \gtrsim \textbf{let } \{x = \mathbb{G}[x]\} \textbf{ in } \mathbb{C}[x] \end{array}$$

This version of fusion has the advantage of having a stronger conclusion, but its strictness side-condition and lack of symmetry make it unsuitable for our purposes.

5.2 The Worker/Wrapper Improvement Theorem

Using the above set of rules, we can prove the following *worker/wrapper improvement* theorem, giving conditions under which a program factorisation is a time improvement:

Theorem 2 (Worker/Wrapper Improvement)**.**

Given value contexts Abs, Rep, $\mathbb{F}$, $\mathbb{G}$ *for which* x *is free satisfying one of the assumptions*

$$\begin{array}{ll} (A)\ \mathsf{Abs}[\mathsf{Rep}[x]] & \lessapprox\gtrsim x \\ (B)\ \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[x]]] & \lessapprox\gtrsim \mathbb{F}[x] \\ (C)\ \textbf{let } x = \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[x]]] \textbf{ in } x & \lessapprox\gtrsim \textbf{let } x = \mathbb{F}[x] \textbf{ in } x \end{array}$$

and one of the conditions

$$\begin{array}{ll} (1)\ \mathbb{G}[x] & \lessapprox \mathsf{Rep}[\mathbb{F}[\mathsf{Abs}[x]]] \\ (2)\ \mathbb{G}[\checkmark\mathsf{Rep}[x]] & \lesssim \mathsf{Rep}[\checkmark\mathbb{F}[x]] \\ (3)\ \mathsf{Abs}[\checkmark\mathbb{G}[x]] & \lesssim \mathbb{F}[\checkmark\mathsf{Abs}[x]] \end{array}$$

$$(1\beta)\ \textbf{let } x = \mathbb{G}[x] \textbf{ in } x \lessapprox \textbf{let } x = \mathsf{Rep}[\mathbb{F}[\mathsf{Abs}[x]]] \textbf{ in } x$$
$$(2\beta)\ \textbf{let } x = \mathbb{G}[x] \textbf{ in } x \lessapprox \textbf{let } x = \mathbb{F}[x] \textbf{ in } \mathsf{Rep}[x]$$

we have the improvement

$$\textbf{let } x = \mathbb{F}[x] \textbf{ in } x \gtrapprox \textbf{let } x = \mathbb{G}[x] \textbf{ in } \mathsf{Abs}[x]$$

Given a recursive program $\textbf{let } x = \mathbb{F}[x] \textbf{ in } x$ and *abstraction* and *representation* contexts Abs and Rep, this theorem gives us conditions we can use to derive a factorised program $\textbf{let } x = \mathbb{G}[x] \textbf{ in } \mathsf{Abs}[x]$. This factorised program will be at worst a constant factor slower than the original program, but can potentially be asymptotically faster. In other words, we have conditions that guarantee that such an optimisation is "safe" with respect to time performance.

The proof given in [25] for the original factorisation theorem centers on the use of the rolling and fusion rules. Because we have proven analogous rules in our setting, the proofs can be adapted fairly straightforwardly, simply by keeping the general form of the proofs and using the rules of improvement theory as structural rules that fit between the original steps. The details are as follows.

We begin by noting that $(A) \Rightarrow (B) \Rightarrow (C)$, as in the original case. The first implication $(A) \Rightarrow (B)$ no longer follows immediately, but the proof is simple. Leting y be a fresh variable, we reason as follows:

$$\begin{array}{ll} & \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[y]]] \\ \lessapprox\gtrsim & \{\ \text{garbage collection, value-}\beta\ \} \\ & \textbf{let } x = \mathbb{F}[y] \textbf{ in } \mathsf{Abs}[\mathsf{Rep}[x]] \\ \lessapprox\gtrsim & \{\ (A)\ \} \\ & \textbf{let } x = \mathbb{F}[y] \textbf{ in } x \\ \lessapprox\gtrsim & \{\ \text{value-}\beta\text{, garbage collection}\ \} \\ & \mathbb{F}[y] \end{array}$$

The final step is to observe that as both x and y are fresh, we can substitute one for the other and the relationship between the terms will remain the same. Hence, we can conclude (B).

As in the original theorem, we have that (1) implies (1β) by simple application of substitution, (2) implies (2β) by fusion and (3) implies the conclusion also by fusion. Under assumption (C), we have that (1β) and (2β) are equivalent. We show this by proving their right hand sides cost-equivalent, after which we can simply apply transitivity.

$$\begin{array}{ll} & \textbf{let } x = \mathbb{F}[x] \textbf{ in } \mathsf{Rep}[x] \\ \lessapprox\gtrsim & \{\ \text{value-}\beta\ \} \\ & \textbf{let } x = \mathbb{F}[x] \textbf{ in } \mathsf{Rep}[\mathbb{F}[x]] \\ \lessapprox\gtrsim & \{\ \text{value let-floating}\ \} \\ & \mathsf{Rep}[\mathbb{F}[\textbf{let } x = \mathbb{F}[x] \textbf{ in } x]] \\ \lessapprox\gtrsim & \{\ (C)\ \} \\ & \mathsf{Rep}[\mathbb{F}[\textbf{let } x = \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[x]]] \textbf{ in } x]] \\ \lessapprox\gtrsim & \{\ \text{value let-floating}\ \} \\ & \textbf{let } x = \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[x]]] \textbf{ in } \mathsf{Rep}[\mathbb{F}[x]] \\ \lessapprox\gtrsim & \{\ \text{rolling}\ \} \\ & \textbf{let } x = \mathsf{Rep}[\mathbb{F}[\mathsf{Abs}[x]]] \textbf{ in } x \end{array}$$

Finally, we must show that condition (1β) and assumption (C) together imply the conclusion. This follows exactly the same pattern of reasoning as the original proof, with the addition of two applications of value-let floating:

$$\begin{array}{ll} & \textbf{let } x = \mathbb{F}[x] \textbf{ in } x \\ \lessapprox\gtrsim & \{\ (C)\ \} \\ & \textbf{let } x = \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[x]]] \textbf{ in } x \\ \lessapprox\gtrsim & \{\ \text{rolling}\ \} \\ & \textbf{let } x = \mathsf{Rep}[\mathbb{F}[\mathsf{Abs}[x]]] \textbf{ in } \mathsf{Abs}[x] \\ \lessapprox\gtrsim & \{\ \text{value let-floating}\ \} \\ & \mathsf{Abs}[\textbf{let } x = \mathsf{Rep}[\mathbb{F}[\mathsf{Abs}[x]]] \textbf{ in } x] \\ \gtrapprox & \{\ (1\beta)\ \} \\ & \mathsf{Abs}[\textbf{let } x = \mathbb{G}[x] \textbf{ in } x] \\ \lessapprox\gtrsim & \{\ \text{value let-floating}\ \} \\ & \textbf{let } x = \mathbb{G}[x] \textbf{ in } \mathsf{Abs}[x] \end{array}$$

We conclude this section by discussing a few important points about the worker/wrapper improvement theorem and its applications. Firstly, we note that the condition (A) will never actually hold. To see this, we let Ω be a divergent term; that is, one that the abstract machine will never finish evaluating. By substituting into the context $\textbf{let } x = \Omega \textbf{ in } [-]$, we obtain the following cost-equivalence:

$$\textbf{let } x = \Omega \textbf{ in } \mathsf{Abs}[\mathsf{Rep}[x]] \lessapprox\gtrsim \textbf{let } x = \Omega \textbf{ in } x$$

This is clearly false, as the left-hand side will terminate almost immediately (as Abs is a value context), while the right-hand side will diverge. Thus we see that assumption (A) is impossible to satisfy. We leave it in the theorem

for completeness of the analogy with the earlier theorem from section 3.1. In situations where (A) would have been used with the earlier theory, the weaker assumption (B) can always be used instead. As we will see later with the examples, frequently only very few properties of the context $\mathbb{F}$ will be used in the proof of (B). A *typed* improvement theory might allow these properties to be assumed of x instead, thus making (A) useful again.

Secondly, we note the restriction to value contexts. This is not actually a particularly severe restriction: for the common application of recursively-defined functions, it is fairly straightforward to ensure that all contexts be of the form $\lambda x \to \mathbb{C}$. For other applications it may be more difficult to find Abs and Rep contexts with the required relationship.

Finally, we note that only conditions (2) and (3) use normal improvement, with all other assumptions and conditions using the weaker version. This is because weak improvement is not strong enough to permit the use of fusion, which these conditions rely on. This makes these conditions harder to prove. However, when these conditions are used, their strength allows us to narrow down the source of any constant-factor slowdown that may take place.

6. Examples

6.1 Reversing a List

In this section we shall demonstrate the utility of our theory with two practical examples. We begin by revisiting the earlier example of reversing a list. In order to apply our theory, we must first write *reverse* as a recursive let:

$$\begin{aligned}
&\mathit{reverse} &&= \mathbf{let}\ \{f = \mathsf{Revbody}\ [f]\}\ \mathbf{in}\ f\\
&\mathsf{Revbody}[M] &&= \lambda xs \to \mathbf{case}\ xs\ \mathbf{of}\\
&&&\qquad [\,] \to [\,]\\
&&&\qquad (y : ys) \to M\ ys +\!\!+ [y]
\end{aligned}$$

The *abs* and *rep* functions from before give rise to to the following contexts:

$$\begin{aligned}
&\mathsf{Abs}[M] = \lambda xs \to M\ xs\ [\,]\\
&\mathsf{Rep}[M] = \lambda xs \to \lambda ys \to M\ xs +\!\!+ ys
\end{aligned}$$

We also require some extra theoretical machinery that we have yet to introduce. To start with, we must assume some rules about the append operation $+\!\!+$. The following associativity rules were proved by Moran and Sands [15].

$$\begin{aligned}
&(xs +\!\!+ ys) +\!\!+ zs \gtrsim xs +\!\!+ (ys +\!\!+ zs)\\
&xs +\!\!+ (ys +\!\!+ zs) \lesssim (xs +\!\!+ ys) +\!\!+ zs
\end{aligned}$$

We assume the following identity improvement as well, which follows from theorems also proved in [15]:

$$[\,] +\!\!+ xs \rhd xs$$

We also require the notion of an *evaluation context*. An evaluation context is a context where evaluation is impossible unless the hole is filled, and have the following form:

$$\begin{aligned}
&\mathbb{E} ::= \mathbb{A}\\
&\quad \mid \mathbf{let}\ \{\vec{x} = \vec{M}\}\ \mathbf{in}\ \mathbb{A}\\
&\quad \mid \mathbf{let}\ \{\vec{y} = \vec{M};\\
&\qquad\quad x_0 = \mathbb{A}_0[x_1];\\
&\qquad\quad x_1 = \mathbb{A}_1[x_2];\\
&\qquad\quad \ldots\\
&\qquad\quad x_n = \mathbb{A}_n\}\\
&\qquad \mathbf{in}\ \mathbb{A}[x_0]
\end{aligned}$$

$$\begin{aligned}
&\mathbb{A} ::= [-]\\
&\quad \mid \mathbb{A}\ x\\
&\quad \mid \mathbf{case}\ \mathbb{A}\ \mathbf{of}\ \{c_i\ \vec{x_i} \to M_i\}
\end{aligned}$$

Note that a context of this form must have exactly one hole. The usefulness of evaluation contexts is that they satisfy some special laws. We use the following in this example:

$$\begin{aligned}
&\mathbb{E}[\checkmark M]\\
&\lhd\!\!\rhd \quad \{\text{ tick floating }\}\\
&\checkmark\mathbb{E}[M]
\end{aligned}$$

$$\begin{aligned}
&\mathbb{E}[\mathbf{case}\ M\ \mathbf{of}\ \{c_i\ \vec{x_i} \to N_i\}]\\
&\lhd\!\!\rhd \quad \{\text{ case floating }\}\\
&\mathbf{case}\ M\ \mathbf{of}\ \{c_i\ \vec{x_i} \to \mathbb{E}[N_i]\}
\end{aligned}$$

$$\begin{aligned}
&\mathbb{E}[\mathbf{let}\ \{\vec{x} = \vec{M}\}\ \mathbf{in}\ N]\\
&\lhd\!\!\rhd \quad \{\text{ let floating }\}\\
&\mathbf{let}\ \{\vec{x} = \vec{M}\}\ \mathbf{in}\ \mathbb{E}[N]
\end{aligned}$$

We conclude by noting that while the context $[-] +\!\!+ ys$ is not strictly speaking an evaluation context (as the hole is in the wrong place), it is cost-equivalent to an evaluation context and so also satisfies these laws. The proof is as follows:

$$\begin{aligned}
&[-] +\!\!+ ys\\
&\equiv \quad \{\text{ desugaring }\}\\
&(\mathbf{let}\ \{xs = [-]\}\ \mathbf{in}\ (+\!\!+)\ xs)\ ys\\
&\lhd\!\!\rhd \quad \{\text{ let floating } [-]\ ys\ \}\\
&\mathbf{let}\ \{xs = [-]\}\ \mathbf{in}\ (+\!\!+)\ xs\ ys\\
&\lhd\!\!\rhd \quad \{\text{ unfolding } +\!\!+\ \}\\
&\mathbf{let}\ \{xs = [-]\}\ \mathbf{in}\\
&\quad \checkmark\mathbf{case}\ xs\ \mathbf{of}\\
&\qquad\quad [\,] \to ys\\
&\qquad\quad (z : zs) \to \mathbf{let}\ \{rs = (+\!\!+)\ zs\ ys\}\ \mathbf{in}\ z : rs\\
&\lhd\!\!\rhd \quad \{\text{ desugaring tick and collecting lets }\}\\
&\mathbf{let}\ \{xs = [-];\\
&\quad r = \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad\quad [\,] \to ys\\
&\qquad\quad (z : zs) \to \mathbf{let}\ \{rs = (+\!\!+)\ zs\ ys\}\ \mathbf{in}\ z : rs\\
&\quad \}\ \mathbf{in}\ r
\end{aligned}$$

Now we can begin the example proper. We start by verifying that Abs and Rep satisfy one of the worker/wrapper assumptions. While earlier we used (A) for this example, the corresponding assumption for worker/wrapper improvement is unsatisfiable. Thus we instead verify assumption (B). The proof is fairly straightforward:

$$\begin{aligned}
&\mathsf{Abs}[\mathsf{Rep}[\mathsf{Revbody}[f]]]\\
&\equiv \quad \{\text{ definitions }\}\\
&\lambda xs \to (\lambda xs \to \lambda ys \to \mathsf{Revbody}[f]\ xs +\!\!+ ys)\ xs\ [\,]\\
&\lhd\!\!\rhd \quad \{\ \beta\text{-reduction }\}\\
&\lambda xs \to \mathsf{Revbody}[f]\ xs +\!\!+ [\,]\\
&\equiv \quad \{\text{ definition of } \mathsf{Revbody}\ \}\\
&\lambda xs \to (\lambda xs \to \mathbf{case}\ xs\ \mathbf{of}\\
&\quad [\,] \to [\,]\\
&\quad (y : ys) \to f\ ys +\!\!+ [y])\ xs +\!\!+ [\,]\\
&\lhd\!\!\rhd \quad \{\ \beta\text{-reduction }\}\\
&\lambda xs \to (\mathbf{case}\ xs\ \mathbf{of}\\
&\quad [\,] \to [\,]\\
&\quad (y : ys) \to f\ ys +\!\!+ [y]) +\!\!+ [\,]\\
&\lhd\!\!\rhd \quad \{\text{ case floating } [-] +\!\!+ [\,]\ \}\\
&\lambda xs \to \mathbf{case}\ xs\ \mathbf{of}\\
&\quad [\,] \to [\,] +\!\!+ [\,]\\
&\quad (y : ys) \to (f\ ys +\!\!+ [y]) +\!\!+ [\,]
\end{aligned}$$

$\lhd\!\rhd$ { associativity is weak cost equivalence }
$\lessapprox$ $\lambda xs \to$ **case** xs **of**
  $[\,] \to [\,] +\!\!+ [\,]$
  $(y : ys) \to f\, ys +\!\!+ ([y] +\!\!+ [\,])$
$\lhd\!\rhd$ { evaluating $[\,] +\!\!+ [\,]$, $[y] +\!\!+ [\,]$ }
$\lessapprox$ $\lambda xs \to$ **case** xs **of**
  $[\,] \to [\,]$
  $(y : ys) \to f\, ys +\!\!+ [y]$
$\equiv$ { definition of revbody }
  Revbody $[f]$

As before, we use condition (2) to derive our $\mathbb{G}$. The derivation is somewhat more involved than before, requiring some care with the manipulation of ticks.

  Rep[✓Revbody$[f]$]
$\equiv$ { definitions }
  $\lambda xs \to \lambda ys \to$
    $(\checkmark \lambda xs \to$ **case** xs **of**
      $[\,] \to [\,]$
      $(z : zs) \to f\, zs +\!\!+ [z])\ xs +\!\!+ ys$
$\lhd\!\rhd$ { float tick out of $[-]\ xs +\!\!+ ys$ }
  $\lambda xs \to \lambda ys \to$
    $\checkmark((\lambda xs \to$ **case** xs **of**
      $[\,] \to [\,]$
      $(z : zs) \to f\, zs +\!\!+ [z])\ xs +\!\!+ ys)$
$\lhd\!\rhd$ { β-reduction }
  $\lambda xs \to \lambda ys \to \checkmark($(**case** xs **of**
      $[\,] \to [\,]$
      $(z : zs) \to f\, zs +\!\!+ [z]) +\!\!+ ys)$
$\lhd\!\rhd$ { case floating $[-] +\!\!+ ys$ }
  $\lambda xs \to \lambda ys \to \checkmark($**case** xs **of**
      $[\,] \to [\,] +\!\!+ ys$
      $(z : zs) \to (f\, zs +\!\!+ [z]) +\!\!+ ys)$
$\gtrsim$ { associativity and identity of $+\!\!+$ }
  $\lambda xs \to \lambda ys \to \checkmark($**case** xs **of**
      $[\,] \to ys$
      $(z : zs) \to f\, zs +\!\!+ ([z] +\!\!+ ys))$
$\gtrsim$ { evaluating $[y] +\!\!+ ys$ }
  $\lambda xs \to \lambda ys \to \checkmark($**case** xs **of**
      $[\,] \to ys$
      $(z : zs) \to f\, zs +\!\!+ (z : ys))$
$\lhd\!\rhd$ { case floating tick ($\star$) }
  $\lambda xs \to \lambda ys \to$ **case** xs **of**
      $[\,] \to \checkmark\, ys$
      $(z : zs) \to \checkmark(f\, zs +\!\!+ (z : ys))$
$\gtrsim$ { removing a tick }
  $\lambda xs \to \lambda ys \to$ **case** xs **of**
      $[\,] \to ys$
      $(z : zs) \to \checkmark(f\, zs +\!\!+ (z : ys))$
$\lhd\!\rhd$ { desugaring }
  $\lambda xs \to \lambda ys \to$ **case** xs **of**
      $[\,] \to ys$
      $(z : zs) \to$
        $\checkmark($**let** $ws = (z : ys)$ **in**
          $f\, zs +\!\!+ ws)$
$\lhd\!\rhd$ { β-expansion }
  $\lambda xs \to \lambda ys \to$ **case** xs **of**
      $[\,] \to ys$
      $(z : zs) \to$
        $\checkmark$**let** $ws = (z : ys)$ **in**
          $(\lambda as \to \lambda bs \to f\, as +\!\!+ bs)\ zs\ ws$
$\lhd\!\rhd$ { tick floating $[-]\ zs\ ws$ }
  $\lambda xs \to \lambda ys \to$ **case** xs **of**
      $[\,] \to ys$
      $(z : zs) \to$
        let $ws = (z : ys)$ **in**
          $(\checkmark \lambda as \to \lambda bs \to f\, as +\!\!+ bs)\ zs\ ws$
$\equiv$ { definition of Rep }
  $\lambda xs \to \lambda ys \to$ **case** xs **of**
      $[\,] \to ys$
      $(z : zs) \to$
        let $ws = (z : ys)$ **in**
          $(\checkmark$Rep$[f])\ zs\ ws$
$\equiv$ { taking this as our definition of $\mathbb{G}$ }
  $\mathbb{G}[\checkmark$Rep$[f]]$

The step marked $\star$ is valid because $\checkmark[-]$ is itself an evaluation context, being syntactic sugar for **let** $x = [-]$ **in** x. Thus we have derived a definition of $\mathbb{G}$, from which we create the following factorised program:

$$reverse = \textbf{let}\ \{rec = \mathbb{G}[rec]\}\ \textbf{in}\ \mathsf{Abs}[rec]$$
$$\begin{aligned}\mathbb{G}[rec]\ = {} & \lambda xs \to \lambda ys \to \textbf{case}\ xs\ \textbf{of} \\ & \quad [\,] \to ys \\ & \quad (z : zs) \to \textbf{let}\ ws = (z : ys)\ \textbf{in} \\ & \qquad rec\ zs\ ws\end{aligned}$$

Expanding this out, we obtain:

$$\begin{aligned}reverse = {} & \textbf{let}\ \{rec = \\ & \quad \lambda xs \to \lambda ys \to \textbf{case}\ xs\ \textbf{of} \\ & \qquad [\,] \to ys \\ & \qquad (z : zs) \to \textbf{let}\ ws = (z : ys)\ \textbf{in} \\ & \qquad\quad rec\ zs\ ws\} \\ & \quad \textbf{in} \quad \lambda xs \to rec\ xs\ [\,]\end{aligned}$$

The result is an implementation of fast reverse as a recursive let. The calculations here have essentially the same structure as the correctness proofs, with the addition of some administrative steps to do with the manipulation of ticks.

To illustrate the performance gain, we have graphed the performance of the original *reverse* function against the optimised version in Figure 2. We used the Criterion benchmarking library [18] with a range of list lengths to compare the performance of the two functions The resulting graph shows a clear improvement from quadratic time to linear. We chose to use relatively small list lengths for our graphs, but the trend continues for larger values.

6.2 Tabulating a Function

Our second example is that of tabulating a function by producing a stream (infinite list) of results. Given a function f that takes a natural number as its argument, the *tabulate* function should produce the following result:

$$[f\,0, f\,1, f\,2, f\,3, \ldots$$

This function can be implemented in Haskell as follows:

$$tabulate\ f = f\,0\ :\ tabulate\ (f \circ (+1))$$

This definition is inefficient, as it requires that the argument to f be recalculated for each element of the result stream. Essentially, this definition corresponds to the following calculation, involving a significant amount of repeated work:

$$[f\,0, f\,(0+1), f\,((0+1)+1), f\,(((0+1)+1)+1), \ldots$$

We wish to apply the worker/wrapper technique to improve the time performance of this program. The first step is to write it as a recursive let in our language:

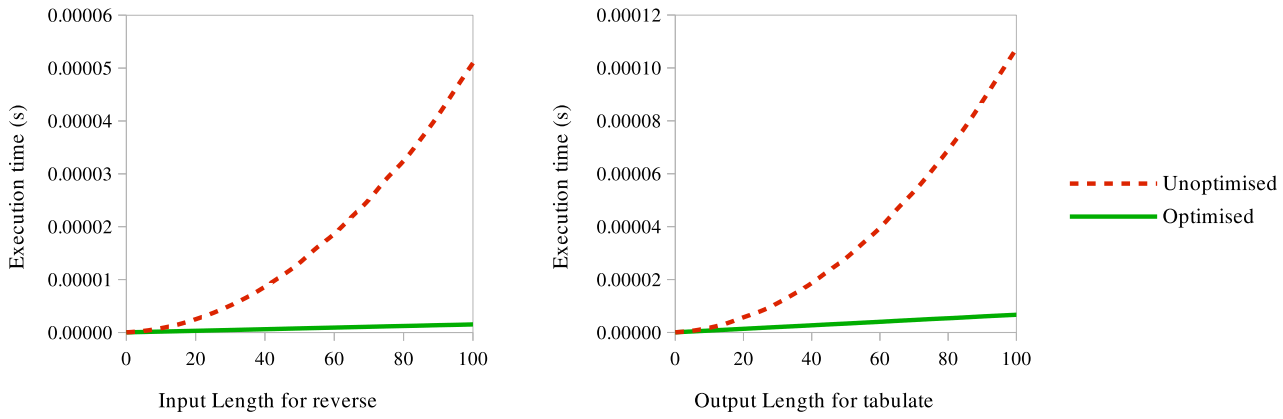

Figure 2. Performance comparisons of *reverse* and *tabulate*

$$
\begin{aligned}
\mathit{tabulate} &= \mathbf{let}\ \{h = \mathbb{F}[h]\}\ \mathbf{in}\ h \\
\mathbb{F}[M] \quad &= \lambda f \to \mathbf{let}\ \{f' = \lambda x \to \\
&\qquad \mathbf{let}\ \{x' = x + 1\}\ \mathbf{in}\ f\,x'\} \\
&\qquad \mathbf{in}\ f\,0 : M\,f'
\end{aligned}
$$

Next, we must devise Abs and Rep contexts. In order to avoid the repeated work, we hope to derive a version of the *tabulate* function that takes an additional number argument telling it where to "start" from. The following Abs and Rep contexts convert between these two versions:

$$
\begin{aligned}
&\mathsf{Abs}[M] = \lambda f \to M\ 0\ f \\
&\mathsf{Rep}[M] = \lambda n \to \lambda f \to \mathbf{let}\ \{f' = \lambda x \to \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ \{x' = x + n\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ f\,x'\} \\
&\qquad\qquad \mathbf{in}\ M\ f'
\end{aligned}
$$

Once again, we must introduce some new rules before we can derive the factorised program. Firstly, we require the following two *variable substitution* rules from [15]:

$$
\begin{aligned}
&\mathbf{let}\ \{x = y\}\ \mathbf{in}\ \mathbb{C}\,[x] \gtrsim \mathbf{let}\ \{x = y\}\ \mathbf{in}\ \mathbb{C}\,[y] \\
&\mathbf{let}\ \{x = y\}\ \mathbf{in}\ \mathbb{C}\,[y] \mathrel{\underset{\approx}{\lhd\rhd}} \mathbf{let}\ \{x = y\}\ \mathbf{in}\ \mathbb{C}\,[x]
\end{aligned}
$$

Next, we must use some properties of addition. Firstly, we have the following *identity* properties:

$$
\begin{aligned}
&x + 0 \mathrel{\underset{\sim}{\lhd\rhd}} x \\
&0 + x \mathrel{\underset{\sim}{\lhd\rhd}} x
\end{aligned}
$$

We also use the following property, combining associativity and commutativity. We shall refer to this as *associativity of* $+$. Where t is not free in $\mathbb{C}$, we have:

$$
\begin{aligned}
&\mathbf{let}\ \{t = x + y\}\ \mathbf{in} \\
&\quad \mathbf{let}\ \{r = t + z\}\ \mathbf{in}\ \mathbb{C}\,[r] \\
&\mathrel{\underset{\sim}{\lhd\rhd}} \\
&\mathbf{let}\ \{t = z + y\}\ \mathbf{in} \\
&\quad \mathbf{let}\ \{r = x + t\}\ \mathbf{in}\ \mathbb{C}\,[r]
\end{aligned}
$$

Finally, we use the fact that sums may be floated out of arbitrary contexts. Where z does not occur in $\mathbb{C}$, we have:

$$
\mathbb{C}\,[\mathbf{let}\ \{z = y + x\}\ \mathbf{in}\ M] \mathrel{\underset{\sim}{\lhd\rhd}} \mathbf{let}\ \{z = y + x\}\ \mathbf{in}\ \mathbb{C}\,[M]
$$

Now we can begin to apply worker/wrapper. Firstly, we verify that Abs and Rep satisfy assumption (B). Again, this is relatively straightforward:

$$
\begin{aligned}
&\mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[h]]] \\
&\equiv \quad \{\ \text{definitions}\ \} \\
&\lambda f \to (\lambda n \to \lambda f \to \mathbf{let}\ \{f' = \lambda x \to \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ \{x' = x + n\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ f\,x'\} \\
&\qquad\qquad\qquad\qquad \mathbf{in}\ \mathbb{F}[h]\ f)\ 0\ f' \\
&\mathrel{\underset{\sim}{\lhd\rhd}} \quad \{\ \beta\text{-reduction}\ \} \\
&\lambda f \to \mathbf{let}\ \{f' = \lambda x \to \\
&\qquad\qquad\qquad\qquad \mathbf{let}\ \{x' = x + 0\} \\
&\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ f\,x'\} \\
&\qquad \mathbf{in}\ \mathbb{F}[h]\ f' \\
&\mathrel{\underset{\approx}{\lhd\rhd}} \quad \{\ x + 0 \mathrel{\underset{\sim}{\lhd\rhd}} x\ \} \\
&\lambda f \to \mathbf{let}\ \{f' = \lambda x \to \\
&\qquad\qquad\qquad\qquad \mathbf{let}\ \{x' = x\} \\
&\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ f\,x'\} \\
&\qquad \mathbf{in}\ \mathbb{F}[h]\ f' \\
&\mathrel{\underset{\approx}{\lhd\rhd}} \quad \{\ \text{variable substitution, garbage collection}\ \} \\
&\lambda f \to \mathbf{let}\ \{f' = \lambda x \to f\,x\} \\
&\qquad \mathbf{in}\ \mathbb{F}[h]\ f' \\
&\equiv \quad \{\ \text{defintion of}\ \mathbb{F}\ \} \\
&\lambda f \to \mathbf{let}\ \{f' = \lambda x \to f\,x\} \\
&\qquad \mathbf{in}\ (\lambda f \to \mathbf{let}\ \{f'' = \lambda x \to \\
&\qquad\qquad\qquad\qquad \mathbf{let}\ \{x' = x + 1\}\ \mathbf{in}\ f\,x\} \\
&\qquad\quad \mathbf{in}\ f\,0 : h\,f'')\ f' \\
&\mathrel{\underset{\sim}{\lhd\rhd}} \quad \{\ \beta\text{-reduction}\ \} \\
&\lambda f \to \mathbf{let}\ \{f' = \lambda x \to f\,x\} \\
&\qquad \mathbf{in}\ \mathbf{let}\ \{f'' = \lambda x \to \\
&\qquad\qquad \mathbf{let}\ \{x' = x + 1\}\ \mathbf{in}\ f'\ x'\} \\
&\qquad\quad \mathbf{in}\ f'\ 0 : h\,f'' \\
&\mathrel{\underset{\approx}{\lhd\rhd}} \quad \{\ \text{value-}\beta\ \text{on}\ f'\ \} \\
&\lambda f \to \mathbf{let}\ \{f'' = \lambda x \to \\
&\qquad \mathbf{let}\ \{x' = x + 1\}\ \mathbf{in}\ (\lambda x \to f\,x)\ x'\} \\
&\qquad \mathbf{in}\ (\lambda x \to f\,x)\ 0 : h\,f'' \\
&\mathrel{\underset{\sim}{\lhd\rhd}} \quad \{\ \beta\text{-reduction}\ \} \\
&\lambda f \to \mathbf{let}\ \{f'' = \lambda x \to \\
&\qquad \mathbf{let}\ \{x' = x + 1\}\ \mathbf{in}\ f\,x'\} \\
&\qquad \mathbf{in}\ f\,0 : h\,f''
\end{aligned}
$$

$\equiv$ { definition of $\mathbb{F}$ }
$\mathbb{F}[h]$

Now we use condition (2) to derive the new definition of *tabulate*. This requires the use of a number of the properties that we presented earlier:

$\mathsf{Rep}[\checkmark\mathbb{F}[h]]$
$\equiv$ { definitions }
$\lambda n \to \lambda f \to \mathbf{let}\ \{f' = \lambda x \to$
$\quad \mathbf{let}\ \{x' = x + n\}$
$\quad\quad \mathbf{in}\ f\,x'\}$
$\quad \mathbf{in}\ (\frown\backslash f \to \mathbf{let}\ \{f'' = \lambda x \to$
$\quad\quad \mathbf{let}\ \{x'' = x + 1\}\ \mathbf{in}\ f\,x''\}$
$\quad\quad \mathbf{in}\ f\,0 : h\,f')\ f$
$\lhd\!\rhd$ { tick floating $[-]\,f'$ }
$\lambda n \to \lambda f \to \mathbf{let}\ \{f' = \lambda x \to$
$\quad \mathbf{let}\ \{x' = x + n\}$
$\quad\quad \mathbf{in}\ f\,x'\}$
$\quad \mathbf{in}\checkmark(\lambda f \to \mathbf{let}\ \{f'' = \lambda x \to$
$\quad\quad \mathbf{let}\ \{x'' = x + 1\}\ \mathbf{in}\ f\,x''\}$
$\quad\quad \mathbf{in}\ f\,0 : h\,f')\ f$
$\lhd\!\rhd$ { β-reduction }
$\lambda n \to \lambda f \to \mathbf{let}\ \{f' = \lambda x \to$
$\quad \mathbf{let}\ \{x' = x + n\}$
$\quad\quad \mathbf{in}\ f\,x'\}$
$\quad \mathbf{in}\checkmark\mathbf{let}\ \{f'' = \lambda x \to$
$\quad\quad \mathbf{let}\ \{x'' = x + 1\}\ \mathbf{in}\ f'\ x''\}$
$\quad\quad \mathbf{in}\ f'\ 0 : h\,f''$
$\lhd\!\rhd$ { value-β on f', garbage collection }
$\lambda n \to \lambda f \to \checkmark\mathbf{let}\ \{f'' = \lambda x \to$
$\quad \mathbf{let}\ \{x' = x + 1\}\ \mathbf{in}$
$\quad\quad (\checkmark\lambda x \to$
$\quad\quad\quad \mathbf{let}\ \{x'' = x + n\}$
$\quad\quad\quad\quad \mathbf{in}\ f\,x'')\ x'\}$
$\quad \mathbf{in}\ (\checkmark\lambda x \to \mathbf{let}\ \{x'' = x + n\}$
$\quad\quad \mathbf{in}\ f\,x'')\ 0 : h\,f''$
$\trianglerighteq$ { removing ticks, β-reduction }
$\lambda n \to \lambda f \to \checkmark\mathbf{let}\ \{f'' = \lambda x \to$
$\quad \mathbf{let}\ \{x' = x + 1\}\ \mathbf{in}$
$\quad\quad \mathbf{let}\ \{x'' = x' + n\}$
$\quad\quad\quad \mathbf{in}\ f\,x''\}$
$\quad \mathbf{in}\ (\mathbf{let}\ \{x'' = 0 + n\}$
$\quad\quad \mathbf{in}\ f\,x'')\ :\ h\,f''$
$\lhd\!\rhd$ { associativity and identity of $+$ }
$\lambda n \to \lambda f \to \checkmark\mathbf{let}\ \{f'' = \lambda x \to$
$\quad \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in}$
$\quad\quad \mathbf{let}\ \{x'' = x + n'\}$
$\quad\quad\quad \mathbf{in}\ f\,x''\}$
$\quad \mathbf{in}\ (\mathbf{let}\ \{x'' = n\}$
$\quad\quad \mathbf{in}\ f\,x'')\ :\ h\,f''$
$\trianglerighteq$ { variable substitution, garbage collection }
$\lambda n \to \lambda f \to \checkmark\mathbf{let}\ \{f'' = \lambda x \to$
$\quad \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in}$
$\quad\quad \mathbf{let}\ \{x'' = x + n'\}$
$\quad\quad\quad \mathbf{in}\ f\,x''\}$
$\quad \mathbf{in}\ f\,n : h\,f''$
$\lhd\!\rhd$ { value let-floating }
$\lambda n \to \lambda f \to f\,n :$
$\quad \checkmark\mathbf{let}\ \{f'' = \lambda x \to$
$\quad\quad \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in}$
$\quad\quad\quad \mathbf{let}\ \{x'' = x + n'\}$
$\quad\quad\quad\quad \mathbf{in}\ f\,x''\}$
$\quad\quad\quad \mathbf{in}\ h\,f''$
$\lhd\!\rhd$ { sums float }
$\lambda n \to \lambda f \to f\,n :$
$\quad \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in}$
$\quad\quad \checkmark\mathbf{let}\ \{f'' = \lambda x \to$
$\quad\quad\quad \mathbf{let}\ \{x'' = x + n'\}$
$\quad\quad \mathbf{in}\ f\,x''\}$
$\quad\quad\quad \mathbf{in}\ h\,f''$
$\lhd\!\rhd$ { β-expansion, tick floating }
$\lambda n \to \lambda f \to f\,n :$
$\quad \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in}$
$\quad\quad (\checkmark\lambda n \to \lambda f \to \mathbf{let}\ \{f'' = \lambda x \to$
$\quad\quad\quad\quad \mathbf{let}\ \{x' = x + n\}$
$\quad\quad\quad\quad\quad \mathbf{in}\ f\,x'\}$
$\quad\quad\quad \mathbf{in}\ h\,f'')\ n'\ f$
$\equiv$ { definition of Rep }
$\lambda n \to \lambda f \to f\,n :$
$\quad \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in}$
$\quad (\checkmark\mathsf{Rep}[h])\ n'\ f$
$\equiv$ { taking this as our definition of $\mathbb{G}$ }
$\mathbb{G}[\checkmark\mathsf{Rep}[h]]$

Thus we have derived a definition of $\mathbb{G}$, from which we create the following factorised program:

$$\mathit{tabulate} = \mathbf{let}\ \{h = \mathbb{G}[h]\}\ \mathbf{in}\ \mathsf{Abs}[h]$$
$$\mathbb{G}[M] \quad = \lambda n \to \lambda f \to f\,n : \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in}\ M\ n'\ f$$

This is the same optimised *tabulate* function that was proved correct in [10], and the proofs here have a similar structure to the correctness proofs from that paper, except that we have now formalised that the new version of the *tabulate* function is indeed a time improvement of the original version. We note that the proof of (B) is complicated by the fact that η-reduction is not valid in this setting. In fact, if we assumed η-reduction then our proof of (B) here could be adapted into a proof of (A).

We demonstrate the performance gain in Figure 2, again based on Criterion benchmarks. This time, we keep the same input (in this case the function $\lambda n \to n * n$), but vary how many elements of the result stream we evaluate. Once again, we have an improvement from quadratic to linear performance, and the trend continues for larger values.

7. Related Work

We divide the related work into three sections. Firstly, we discuss various approaches to the operational semantics of lazy languages. Secondly, we discuss the history of improvement theory. Finally, we discuss other approaches that have been used to formally reason about efficiency.

7.1 Lazy Operational Semantics

The notion of call-by-need evaluation was first introduced in 1971 by Wadsworth [30]. However, the semantics most widely regarded as the definition of call-by-need is the natural semantics due to Launchbury [14], which was later used by Sestoft to derive the virtual machine semantics we use in this paper [27]. Ariola, Felleisen, Maraist, Odersky and Wadler presented a *call-by-need lambda calculus* [1], with operational semantics based on reductions between terms in the source language. This calculus supports an equational theory. However, Moran and Sands showed that this equational theory is subsumed by weak cost-equivalence [15].

7.2 Improvement Theory

Improvement theory was originally developed in 1991 by Sands [21], and applied in a call-by-name setting. In 1997 this was generalised to a wide class of call-by-name and call-by-value languages, also by Sands [22]. This theory was also applicable to a general class of *resources*, rather than just space and time. The theory for lazy languages was developed by Moran and Sands for time efficiency [15] and Gustavsson and Sands for space efficiency [8, 9]. Since the last of these papers was published in 2001, there does not seem to have been much work on improvement theory. We hope that this paper can help to regenerate interest in this topic.

7.3 Formal Reasoning About Efficiency

Okasaki [17] uses techniques of amortised cost analysis to reason about the asymptotic time complexity of lazy functional data structures. This is achieved by modifying analysis techniques such as the Banker's Method, where the notion of *credit* is used to spread out the notional cost of an expensive but infrequent operations over more frequent and cheaper operations. The key idea in Okasaki's work is to invert such techniques to use the notion of *debt*. This allows the analyses to deal with the persistence of data structures, where the same structure may exist in multiple versions at once. While credit may only be spent once, a single debt may be paid off multiple times (in different versions of the same structure) without risking bankruptcy. These techniques have been used to analyse the asymptotic performance of a number of functional data structures.

Sansom and Peyton Jones [24] give a presentation of the GHC profiler, which can be used to measure time as well as space usage of Haskell programs. In doing so, they give a formal cost semantics for GHC Core programs based around the notion of *cost centres*. Cost centres are a way of annotating expressions, so that the profiler can indicate which parts of the source program cost the most to execute. The cost semantics is used as a specification to develop a precise profiling framework, as well as to prove various properties about cost attribution and verify that certain program transformations do not affect the attribution of costs, though they may of course reduce cost overall. Cost centres are now widely-used in profiling Haskell programs.

Hope [11] applies a technique based on *instrumenting* an abstract machine with cost information to derive a cost semantics for call-by-value functional programs. More specifically, starting from a denotational semantics for the source language, one derives an abstract machine for this language using standard program transformation techniques, instruments this machine with cost information, and then reverses the derivation to arrive at an instrumented denotational semantics. This semantics can then be used to reason about the cost of programs in the high-level source language without reference to the details of the abstract machine. This approach was used to calculate the space and time cost of a range of programming examples, as well as to derive a new deforestation theorem for hylomorphisms.

8. Conclusion

In this paper, we have shown how improvement theory can be used to justify the worker/wrapper transformation as a program optimisation, by formally proving that, under certain natural conditions, the transformation is guaranteed to preserve or improve time performance. This guarantee is with respect to an established operational semantics for call-by-need evaluation. We then verified that two examples from previous worker/wrapper papers met the preconditions for this performance guarantee, demonstrating the use of our theory while also verifying the validity of the examples. This work appears to be the first time that rigorous performance guarantees have been given for a general purpose optimisation technique in a call-by-need setting.

8.1 Further Work

As well as for fixed points, worker/wrapper theories also exist for more structured recursion operators such as folds [13] and unfolds [10]. Though the theory we present here can be specialised to such operators, it may be beneficial to investigate this more closely, as doing so may reveal more interesting and subtle details yet to be uncovered.

As we mentioned earlier in this paper, a typed theory would be more useful, allowing more power when reasoning about programs. This would also match more closely with the original worker/wrapper theories, which were typed. The key barrier to this is that there is currently no typed improvement theory, so such a theory would have to be developed before the theory here could be made typed.

The theory we present here only applies to time efficiency. Gustavsson and Sands have developed an improvement theory for space [8, 9], so this would be an obvious next step for developing our theory. More generally, we could apply a technique such as that used by Sands [22] to develop a theory that applies to a large class of resources, and examine which assumptions must be made about the resources we consider for our theory to apply.

Assumptions (A), (B) and (C) are written as weak cost-equivalences, which limits the scope of our theory to cases where Abs and Rep are fairly simple. We would like to also be able to cover cases where the Abs and Rep contexts correspond to expensive operations, but the extra cost is made up for by the overall efficiency gain of the transformation. To cover such cases, we would require a richer version of improvement theory that is able to quantify *how much* better one program is than another.

As our examples show, the calculations required to derive an improved program can often be quite involved. The HERMIT system, devised by a team at the University of Kansas [6, 26], facilitates program transformations by providing an interactive interface for program transformation that verifies correctness. If improvement theory could be integrated into such a system, it would be significantly easier to apply our worker/wrapper improvement theory.

Finally, we are working on a general worker/wrapper theory that will apply to any operator with the property of *dinaturality* [5]. It is also interesting to consider whether such a general categorical approach can be applied to an operational theory. If this is the case, dinaturality may also provide the necessary machinery to *unify* the denotational (correctness) and operational (efficiency) theories, which as we have already observed in this paper are very similar in terms of their formulations and proofs. Voigtländer and Johann used parametricity to justify program transformations from a perspective of *observational approximation* [29]. It may be productive to investigate whether their techniques can be applied to a notion of improvement.

Acknowledgments

The authors would like to thank the reviewers as well as Neil Sculthorpe, for their helpful comments on this paper.

References

[1] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The Call-by-Need Lambda Calculus. In *POPL '95*, pages 233–246. ACM, 1995.

[2] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24 (1):44–67, 1977.

[3] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*. ACM, 2000.

[4] T. Coquand. Infinite Objects in Type Theory. In *TYPES '93*, volume 806 of *Lecture Notes in Computer Science*. Springer, 1993.

[5] E. Dubuc and R. Street. Dinatural Transformations. In *Lecture Notes in Mathematics*, volume 137 of *Lecture Notes in Mathematics*, pages 126–137. Springer Berlin Heidelberg, 1970.

[6] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Haskell Symposium (Haskell '12)*, pages 1–12. ACM, 2012.

[7] A. Gill and G. Hutton. The Worker/Wrapper Transformation. *Journal of Functional Programming*, 19(2), 2009.

[8] J. Gustavsson and D. Sands. A Foundation for Space-Safe Transformations of Call-by-Need Programs. *Electronic Notes on Theoretical Computer Science*, 26:69–86, 1999.

[9] J. Gustavsson and D. Sands. Possibilities and Limitations of Call-by-Need Space Improvement. In *ICFP*, pages 265–276. ACM, 2001.

[10] J. Hackett, G. Hutton, and M. Jaskelioff. The Under Performing Unfold: A New Approach to Optimising Corecursive Programs. 2013. To appear in the volume of selected papers from the 25th International Symposium on Implementation and Application of Functional Languages, Nijmegen, The Netherlands, August 2013.

[11] C. Hope. *A Functional Semantics for Space and Time*. PhD thesis, School of Computer Science, University of Nottingham, 2008.

[12] J. Hughes. A Novel Representation of Lists and its Application to the Function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[13] G. Hutton, M. Jaskelioff, and A. Gill. Factorising Folds for Faster Functions. *Journal of Functional Programming Special Issue on Generic Programming*, 20(3&4), June 2010.

[14] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *POPL '93*, pages 144–154. ACM, 1993.

[15] A. Moran and D. Sands. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. Extended version of [16], available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.144&rep=rep1&type=pdf`.

[16] A. Moran and D. Sands. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *POPL '99*, pages 43–56. ACM, 1999.

[17] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[18] B. O'Sullivan. Criterion, A New Benchmarking Library for Haskell, 2009. URL `http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/`.

[19] W. Partain. The nofib Benchmark Suite of Haskell Programs. In *Glasgow Workshop on Functional Programming*, pages 195–202. Springer, 1992.

[20] S. L. Peyton Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *ESOP*, pages 18–44. Springer, 1996.

[21] D. Sands. Operational Theories of Improvement in Functional Languages (Extended Abstract). In *Glasgow Workshop on Functional Programming*, 1991.

[22] D. Sands. From SOS Rules to Proof Principles: An Operational Metatheory for Functional Languages. In *POPL '97*, pages 428–441. ACM Press, 1997.

[23] D. Sands. Improvement Theory and its Applications. In *Higher Order Operational Techniques in Semantics, Publications of the Newton Institute*. Cambridge University Press, 1997.

[24] P. M. Sansom and S. L. Peyton Jones. Formally Based Profiling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems*, 19(2), 1997.

[25] N. Sculthorpe and G. Hutton. Work It, Wrap It, Fix It, Fold It. *Journal of Functional Programming*, 2014.

[26] N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language. In *Proceedings of Implementation and Application of Functional Languages (IFL '12)*, volume 8241 of *Lecture Notes in Computer Science*, pages 86–103, 2013.

[27] P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[28] D. A. Turner. Elementary Strong Functional Programming. In *FPLE '95*, volume 1022 of *Lecture Notes in Computer Science*. Springer, 1995.

[29] J. Voigtländer and P. Johann. Selective Strictness and Parametricity in Structural Operational Semantics, Inequationally. *Theor. Comput. Sci.*, 388(1-3):290–318, 2007.

[30] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Computing Laboratory, University of Oxford, 1971.

[31] O. Wilde. Lady Windermere's Fan, A Play About a Good Woman. First performed in 1892.