Improving Haskell

Martin A.T. Handley and Graham Hutton

School of Computer Science, University of Nottingham, UK

Abstract. Lazy evaluation is a key feature of Haskell, but can make it difficult to reason about the efficiency of programs. Improvement theory addresses this problem by providing a foundation for proofs of program improvement in a call-by-need setting, and has recently been the subject of renewed interest. However, proofs of improvement are intricate and require an inequational style of reasoning that is unfamiliar to most Haskell programmers. In this article, we present the design and implementation of an inequational reasoning assistant that provides mechanical support for improvement proofs, and demonstrate its utility by verifying a range of improvement results from the literature.

1 Introduction

Reasoning about the efficiency of Haskell programs is notoriously difficult and counterintuitive. The source of the problem is Haskell's use of lazy evaluation, or more precisely, call-by-need semantics, which allows computations to be performed with terms that are not fully normalised. In practice, this means that the operational efficiency of a term does not necessarily follow from the number of steps it takes to evaluate to normal form, in contrast to a call-by-value setting where reasoning about efficiency is much simpler.

Moran and Sands' *improvement theory* [1] offers the following solution to this problem: rather than counting the steps required to normalise a term in isolation, we compare the number of steps required in all program contexts. This idea gives rise to a compositional approach to reasoning about efficiency in call-by-need languages that can be used to verify improvement results.

Improvement theory was originally developed in the 1990s, but has recently been the subject of renewed interest, with a number of general-purpose program optimisations being formally shown to be improvements [2,3,4]. In an effort to bridge the so-called correctness/efficiency 'reasoning gap' [5], these articles show that it is indeed possible to formally reason about the performance aspects of optimisation techniques in a call-by-need setting.

While improvement theory provides a suitable basis for reasoning about efficiency in Haskell, the resulting proofs are often rather intricate [2], and constructing them by hand is challenging. In particular, comparing the cost of evaluating terms in all program contexts requires a somewhat elaborate reasoning process, and the resulting *inequational* style of reasoning is inherently more demanding than the equational style that is familiar to most Haskell programmers.

To support interactive *equational* reasoning about Haskell programs, the Hermit toolkit [6] was recently developed, and its utility has been demonstrated in a series of case studies [7,8,9,10,11,12]. Although inequational reasoning is more involved than its equational counterpart, both approaches share the same calculational style. In addition, the Hermit system and improvement theory are both

```
≡ λxs.λys.✓(case xs of
  []     → ys
  (z:zs) → ((f zs) ++ [z]) ++ ys)
[18]> append-assoc-lr-i
⊵ λxs.λys.✓(case xs of
  []     → ys
  (z:zs) → (f zs) ++ ([z] ++ ys))
[19]> right
≡ λxs.λys.✓(case xs of
  []     → ys
  (z:zs) → (f zs) ++ ([z] ++ ys))
[20]> eval-i
⊵ λxs.λys.✓(case xs of
  []     → ys
  (z:zs) → (f zs) ++ (z:ys))
```

Fig. 1. An extract from a proof in our system

based on the same underlying setting: the core language of the Glasgow Haskell Compiler. As such, a system developed in a similar manner to Hermit could prove to be effective in supporting inequational reasoning for proofs of program improvement, just as Hermit has proven to be effective in supporting equational reasoning for proofs of program correctness.

To the best of our knowledge, no such inequational reasoning system exists. To fill this gap, we developed the University of Nottingham Improvement Engine (Unie): an interactive, mechanised assistant for call-by-need improvement theory. More specifically, the article makes the following contributions:

- We show how the Kansas University Rewrite Engine (Kure), which forms the basis for equational reasoning in the Hermit system, can also form the basis for inequational reasoning in our system (section 4);
- We implement Moran and Sands' tick algebra in our system, which is an inequational theory that allows evaluation costs to be moved around in terms, and verify a range of basic tick algebra results (sections 3.5 and 4);
- We explain how program contexts, a central aspect of improvement theory, are automatically managed by our system, and show how this simplifies reasoning steps in mechanised improvement proofs (section 4.5);
- We demonstrate the practicality of our system by mechanically verifying all the improvement results in the article that renewed interest in improvement theory [2], and a number from the original article [1] (section 6).

By way of example, an extract from an improvement proof in our system — concerning the *reverse* function on lists — is given in Fig. 1. In each step, the term highlighted in orange is being transformed. In the first step, the append operator $+\!\!+$ is reassociated to the right, which is an improvement, denoted using the $\trianglerighteq$ symbol. We then move to the right in the term, and evaluate the $+\!\!+$, which is also an improvement. The tick symbol ✓ in the proof represents a unit time cost. We will revisit this example in more detail throughout the article.

Our improvement assistant comprises approximately 13,000 lines of Haskell on top of the Kure framework, and is freely available online [13].

2 Example

To provide some intuition for improvement theory, and demonstrate how its technicalities can benefit from mechanical support, we begin with an example that

underpins the proof in Fig. 1. Consider the following property, which formalises that Haskell's list append operator $+\!\!+$ is associative (for finite lists):

$$(xs +\!\!+ ys) +\!\!+ zs \;\; = \;\; xs +\!\!+ (ys +\!\!+ zs) \tag{1}$$

A common, informal argument about the above equation is that the left-hand side is less time efficient than the right-hand side, because the former traverses the list xs twice. This insight is often exploited when optimising functions defined in terms of append [14]. Optimisations of this kind typically demonstrate the correctness of (1), which can be verified by a simple inductive proof, but fail to make precise any efficiency claims about the equation. Can we do better?

Using improvement theory, we can formally verify which side of the equation is more efficient by comparing the evaluation costs of each term in all program contexts. That is, we can show that one side is 'improved by' the other, written:

$$(xs +\!\!+ ys) +\!\!+ zs \;\; \trianglerighteq \;\; xs +\!\!+ (ys +\!\!+ zs) \tag{2}$$

Before sketching how the above inequation can be proven, we introduce some necessary background material. As the focus here is on illustrating the basic ideas by means of an example, we simplify the theory where possible and will return to the precise details in the next section.

Contexts and improvement. In the usual manner, contexts are 'terms with holes', denoted $[-]$, which can be substituted with other terms. Informally, a term M is *improved by* a term N, written $M \trianglerighteq N$, if in all contexts the evaluation of N requires no more function calls than the evaluation of M. If the evaluations of the terms require the same number of function calls in all contexts, then the terms are said to be *cost-equivalent*, written $M \lhd\!\rhd N$.

Ticks. While reasoning about improvement, it is necessary to keep track of evaluation cost explicitly within the syntax of the source language (see [1] for a detailed explanation). This is achieved by means of a tick annotation $\checkmark$ that represents a unit time cost, i.e. one function call. Denotationally, ticks have no effect on terms. Operationally, however, a tick represents a function call. Hence, a term M evaluates with n function calls iff $\checkmark M$ evaluates with $n+1$ function calls. Moreover, for any function definition $f\ \boldsymbol{x} = M$, we have the cost-equivalence

$$f\ \boldsymbol{x} \;\; \lhd\!\rhd \;\; \checkmark M \tag{3}$$

because unfolding the definition eliminates the function call. Removing a tick improves a term, $\checkmark M \trianglerighteq M$, but the converse $M \trianglerighteq \checkmark M$ is not valid.

Improvement induction. A difficulty with the definition of improvement is that it quantifies over all contexts, and hence proving (2) notionally requires considering all possible contexts. However, this is a standard issue with contextual definitions, and there are a number of methods for constructing proofs in a more tractable manner. We use improvement induction [1] for this purpose, presented here in a simplified form. For any context $\mathbb{C}$, we have:

$$\frac{M \;\trianglerighteq\; \checkmark\mathbb{C}[M] \qquad \checkmark\mathbb{C}[N] \;\lhd\!\rhd\; N}{M \;\trianglerighteq\; N}$$

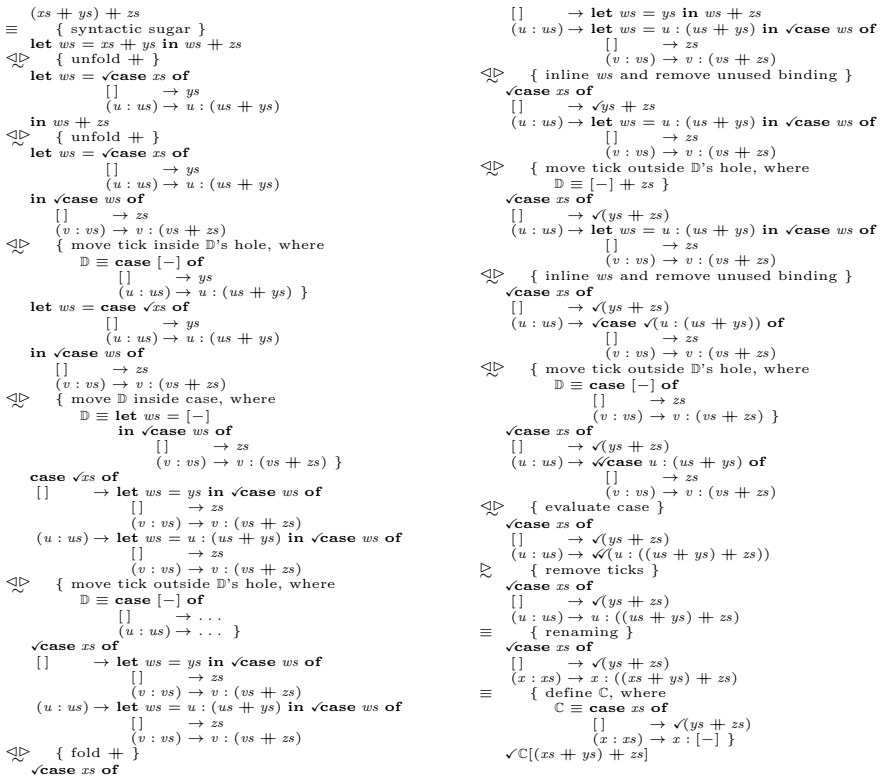


Fig. 2. Proof of property (4)

Intuitively, this rule allows us to prove $M \trianglerighteq N$ by finding a single context $\mathbb{C}$ for which we can 'unfold' M to $\checkmark\mathbb{C}[M]$ and 'fold' $\checkmark\mathbb{C}[N]$ to N. For example, applying improvement induction to inequation (2) reduces the problem to finding a context $\mathbb{C}$ that satisfies the following two properties:

$$(xs +\!\!+ ys) +\!\!+ zs \quad \trianglerighteq \quad \checkmark\mathbb{C}[(xs +\!\!+ ys) +\!\!+ zs] \tag{4}$$

$$\checkmark\mathbb{C}[xs +\!\!+ (ys +\!\!+ zs)] \quad \lhd\!\rhd \quad xs +\!\!+ (ys +\!\!+ zs) \tag{5}$$

Proof of property (2)**.** For the purposes of this example, we can assume that the source language is Haskell, with one small caveat: arguments to functions must be variables. Improvement theory requires this assumption and it is easy to achieve by introducing **let** bindings. For example, the term $(xs +\!\!+ ys) +\!\!+ zs$ can be viewed as syntactic sugar for **let** $ws = xs +\!\!+ ys$ **in** $ws +\!\!+ zs$.

Using improvement induction, we can prove (2) by finding a context $\mathbb{C}$ for which properties (4) and (5) hold. We prove the first of these properties in Fig. 2; the second proceeds similarly. As we have not yet presented the rules of the tick algebra (section 3.5), the reader is encouraged to focus on the overall structure of the reasoning in Fig. 2, rather than the technicalities of each step. Our system is specifically designed to support and streamline such reasoning.

3 Improvement Theory

In this section, we return to the formalities of Moran and Sands' call-by-need improvement theory. While explaining the theory, we describe how our system supports, and in many cases simplifies, its resulting technicalities.

3.1 Syntax and Semantics

The operational model that forms the basis of call-by-need improvement theory is an untyped, higher-order language with mutually recursive let bindings. The call-by-need semantics is originally due to Sestoft [15] and reflects Haskell's use of lazy evaluation. Furthermore, the language is comparable to (a normalised version of) the core language of the Glasgow Haskell Compiler. We use these similarities to apply results from this theory directly to Haskell.

Terms of the language are defined by the following grammar, which also comprises the abstract syntax manipulated by our system:

$$M, N ::= x \mid \lambda x.M \mid M\ x \mid \mathbf{let}\ \{\boldsymbol{x} = \boldsymbol{M}\}\ \mathbf{in}\ N \mid c\ \boldsymbol{x} \mid \mathbf{case}\ M\ \mathbf{of}\ \{c_i\ \boldsymbol{x_i} \to N_i\}$$

We use the symbols x, y, z for variables, c for constructors, and write $\boldsymbol{x} = \boldsymbol{M}$ for a sequence of bindings of the form $x = M$. Similarly, we write $c_i\ \boldsymbol{x_i} \to N_i$ (or sometimes *alts*) for a sequence of **case** alternatives of the form $c\ \boldsymbol{x} \to N$. Literals are represented by constructors of arity 0, and all constructors are assumed to be fully applied. A term is a *value*, denoted V, if it is of the form $\lambda x.M$ or $c\ \boldsymbol{x}$, which corresponds to the usual notion of weak head normal form.

The abstract machine for evaluating terms 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, and the evaluation stack S given by a list of tokens used by the abstract machine. The machine operates by evaluating the current term to a value, and then decides how to continue based on the top token on the stack. Bindings generated by **let**s are added to the heap, and only taken off when performing a `Lookup` operation. A `Lookup` executes by putting a token on top of the stack representing where the term was looked up, and then evaluating that term to a value before replacing it on the heap. This ensures that each binding is evaluated at most once: a key aspect of call-by-need semantics. Restricting function arguments to be variables means that all non-atomic arguments must be introduced via **let** statements and thus can be evaluated at most once.

The transitions of the machine are given in Fig. 3. The `Letrec` transition assumes that $\boldsymbol{x}$ is disjoint from (written here as $\not\downarrow$) the domain of Γ and S, which can always be achieved by alpha-renaming.

$$
\begin{array}{lcll}
\langle \Gamma\{x = M\}, x, S\rangle & \longrightarrow & \langle \Gamma, M, \#x : S\rangle & \{\ \texttt{Lookup}\ \} \\
\langle \Gamma, V, \#x : S\rangle & \longrightarrow & \langle \Gamma\{x = V\}, V, S\rangle & \{\ \texttt{Update}\ \} \\
\langle \Gamma, M\ x, S\rangle & \longrightarrow & \langle \Gamma, M, x : S\rangle & \{\ \texttt{Unwind}\ \} \\
\langle \Gamma, \lambda x.M, y : S\rangle & \longrightarrow & \langle \Gamma, M[y/x], S\rangle & \{\ \texttt{Subst}\ \} \\
\langle \Gamma, \mathbf{case}\ M\ \mathbf{of}\ alts, S\rangle & \longrightarrow & \langle \Gamma, M, alts : S\rangle & \{\ \texttt{Case}\ \} \\
\langle \Gamma, c_j\ \boldsymbol{y}, \{c_i\ \boldsymbol{x}_i \longrightarrow N_i\} : S\rangle & \longrightarrow & \langle \Gamma, N_j[\boldsymbol{y}/\boldsymbol{x}_j], S\rangle & \{\ \texttt{Branch}\ \} \\
\langle \Gamma, \mathbf{let}\ \{\boldsymbol{x} = \boldsymbol{M}\}\ \mathbf{in}\ N, S\rangle & \longrightarrow & \langle \Gamma\{\boldsymbol{x} = \boldsymbol{M}\}, N, S\rangle \quad \boldsymbol{x} \not\downarrow \mathrm{dom}(\Gamma, S) & \{\ \texttt{Letrec}\ \}
\end{array}
$$

Fig. 3. Semantics of the call-by-need abstract machine

3.2 Contexts

Program contexts are defined by the following grammar:

$$\mathbb{C}, \mathbb{D} ::= [-] \mid x \mid \lambda x.\mathbb{C} \mid \mathbb{C}\ x \mid \mathbf{let}\ \{\boldsymbol{x} = \mathbb{C}\}\ \mathbf{in}\ \mathbb{D} \mid c\ \boldsymbol{x} \mid \mathbf{case}\ \mathbb{C}\ \mathbf{of}\ \{c_i\ \boldsymbol{x_i} \to \mathbb{D}_i\}$$

Note that **let** and **case** statements admit contexts with multiple holes.

A *value context*, denoted $\mathbb{V}$, is a context that is in weak head normal form. There are also two other kinds of contexts, which can contain at most one hole that must appear as the target of evaluation: meaning evaluation cannot proceed until the hole is substituted. These are known as *evaluation contexts* and *applicative contexts*, and are defined by the following two grammars, respectively:

$$\begin{array}{l} \mathbb{E} ::= \mathbb{A} \\ \quad \mid\ \mathbf{let}\ \{\boldsymbol{x} = \boldsymbol{M}\}\ \mathbf{in}\ \mathbb{A} \\ \quad \mid\ \mathbf{let}\ \{\ \boldsymbol{y}\ = \boldsymbol{M}; \\ \qquad\qquad x_0\ = \mathbb{A}_0[x_1]; \\ \qquad\qquad x_1\ = \mathbb{A}_1[x_2]; \\ \qquad\qquad \ldots \\ \qquad\qquad x_n\ = \mathbb{A}_n\}\ \mathbf{in}\ \mathbb{A}[x_0] \end{array} \qquad \begin{array}{l} \mathbb{A} ::= [-] \\ \quad \mid\ \mathbb{A}\ x \\ \quad \mid\ \mathbf{case}\ \mathbb{A}\ \mathbf{of}\ \{c_i\ \boldsymbol{x_i} \to \boldsymbol{M_i}\} \end{array}$$

As improvement is a contextual definition, intuitively, the transformation rules we apply when reasoning about improvement must also be defined contextually. In general, however, it is not the case that a given transformation rule is valid for *all* contexts. For example, a tick can be freely moved in and out of an evaluation context using the rule

$$\mathbb{E}[\checkmark M] \ \lhd\!\rhd\!\!\approx\ \checkmark\mathbb{E}[M] \qquad (\checkmark\text{-}\mathbb{E})$$

but this is not the case for other kinds of contexts. Similarly, under certain conditions regarding free (FV) and bound (BV) variables, an evaluation context can be moved in and out of a **case** statement:

$$\begin{array}{ll} & \mathbb{E}[\mathbf{case}\ M\ \mathbf{of}\ \{pat_i \to N_i\}] \\ \lhd\!\rhd & \quad FV(M) \not\downarrow BV(\mathbb{E}) \qquad FV(\mathbb{E}) \not\downarrow pat_i \\ & \mathbf{case}\ M\ \mathbf{of}\ \{pat_i \to \mathbb{E}[N_i]\} \end{array} \qquad (\text{case-}\mathbb{E})$$

Consequently, when applying a transformation rule to a term, we must ensure that its syntactic form is compatible with the chosen rule. When conducted manually, the process of deconstructing a term $M \equiv \mathbb{C}[S]$ into a context $\mathbb{C}$ and substitution S becomes tedious, time consuming, and prone to error.

To address this problem, our system handles all aspects of context manipulation automatically. Each time a rule is applied, the system analyses the syntactic form of the term and ensures it is compatible with the rule's specification. If this is not the case, the system prevents the rule from being applied and reports an error. The same is also true if a rule's side conditions are not satisfied, such as those regarding free and bound variables for case-$\mathbb{E}$. Thus, with regards to contexts, not only does the Unie system make a correct transformation much easier to apply, it makes an incorrect transformation impossible to apply.

3.3 Improvement

Moran and Sands [1] showed that the total number of steps taken to evaluate any term is bounded by a function that is linear in the number of `Lookup` operations invoked during its evaluation. Therefore, evaluation cost can be measured *asymptotically* by just counting uses of `Lookup`. This is the notion of cost used in their work, and we also adopt it for our system.

Formally, we write $M{\downarrow^n}$ if the abstract machine proceeds from the initial state $\langle \emptyset, M, \epsilon \rangle$ to some final state $\langle \Gamma, V, \epsilon \rangle$ with n uses of `Lookup`. Similarly, we write $M{\downarrow^{\leqslant n}}$ to mean that $M{\downarrow^m}$ for some $m \leqslant n$. Using this cost model we can now formalise the notion of improvement: a term M is *improved by* a term N, written $M \trianglerighteq N$, if the following holds for all contexts $\mathbb{C}$:

$$\mathbb{C}[M]{\downarrow^n} \implies \mathbb{C}[N]{\downarrow^{\leqslant n}}$$

That is, one term is improved by another if the latter takes no more `Lookup` operations to evaluate than the former in all contexts. In turn, we say that two terms M and N are *cost-equivalent*, written $M \lhd\!\rhd N$, if for all contexts $\mathbb{C}$:

$$\mathbb{C}[M]{\downarrow^n} \iff \mathbb{C}[N]{\downarrow^n}$$

As before, we need to keep track of evaluation costs explicitly. Our informal introduction viewed the tick operator as a syntactic construct that represents a unit time cost. Here we follow [2] and define tick as a derived operation:

$$\checkmark M \equiv \textbf{let}\ \{\, x = M \,\}\ \textbf{in}\ x \qquad (x \text{ fresh})$$

This definition takes precisely two steps to evaluate to M: one to add the binding to the heap, and the other to look it up. As one of these steps is a `Lookup` operation, the cost of evaluating M is increased by exactly one, as required. The following tick elimination rule still holds, but as before the reverse is not valid:

$$\checkmark M \trianglerighteq M \qquad (\checkmark\text{-elim})$$

The relation $\trianglerighteq$ formalises when one term is at least as efficient as another in all contexts, but this is a strong requirement. We use the notion of *weak improvement* [2] when one term is at least as efficient as another within a constant factor. Formally, M is *weakly improved by* N, written $M \mathrel{\reflectbox{$\trianglerighteq$}\!\!\!\!\reflectbox{$\approx$}} N$, if there exists a linear function $f(x) = kx + c$ (for $k, c \geqslant 0$) such that for all contexts $\mathbb{C}$:

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

This can be interpreted as "replacing M with N may make programs worse, but will not make them asymptotically worse" [2]. Analogous to cost-equivalence, we also have weak cost-equivalence, written $M \lhd\!\!\rhd\!\!\!\!\approx N$, which is defined in the obvious manner. As weak improvement ignores constant factors, we can introduce and eliminate ticks while preserving weak cost-equivalence:

$$M \lhd\!\!\rhd\!\!\!\!\approx \checkmark M \qquad (\checkmark\text{-intro})$$

3.4 Inequational Reasoning

When constructing improvement proofs, careful attention must be paid to their respective improvement relations. This is because the transformation rules we apply during reasoning steps are defined using the different notions of improvement ($\trianglerighteq, \gtrapprox, \lhd\rhd, \lessapprox\gtrapprox$), some of which may not entail the relation in question. For example, given that $\trianglerighteq \subseteq \gtrapprox$, any transformation defined using $\trianglerighteq$ automatically entails $\gtrapprox$, but the converse is not true. Similarly, removing a tick ($\checkmark$-elim) is an improvement, whereas unfolding a function's definition (3) is not.

In this instance, our system simplifies the necessary inequational reasoning by ensuring that each transformation rule applied by the user entails a particular improvement relation established prior to the start of the reasoning process. If the user attempts to apply a rule that does not entail this relation, the system will reject it and display an error message.

3.5 The Tick Algebra

We conclude this section by discussing the *tick algebra* [1], which is a collection of laws for propagating evaluation costs around terms while preserving or increasing efficiency. These laws make up a large proportion of the transformation rules that are provided by our system, and are a rich inequational theory that subsumes all axioms of the call-by-need calculus of Ariola et al. [16].

We refer the reader to [1] for the full tick algebra, but present two example laws below to illustrate their nature and complexity:

$$\begin{array}{ll} & \mathbf{let}\ \{\boldsymbol{x} = \boldsymbol{L}\}\ \mathbf{in}\ \mathbf{let}\ \{\boldsymbol{y} = \boldsymbol{M}\}\ \mathbf{in}\ N \\ \lhd\rhd & \quad \boldsymbol{x} \not= \boldsymbol{y} \quad \boldsymbol{y} \not= FV(\boldsymbol{L}) \\ & \mathbf{let}\ \{\boldsymbol{x} = \boldsymbol{L}, \boldsymbol{y} = \boldsymbol{M}\}\ \mathbf{in}\ N \end{array} \qquad \text{(let-flatten)}$$

$$\begin{array}{ll} & \checkmark\mathbf{let}\ \{x = z, \boldsymbol{y} = \boldsymbol{M}[z/w]\}\ \mathbf{in}\ N[z/w] \\ \trianglerighteq & \\ & \mathbf{let}\ \{x = z, \boldsymbol{y} = \boldsymbol{M}[x/w]\}\ \mathbf{in}\ N[x/w] \end{array} \qquad \text{(var-expand)}$$

The (let-flatten) rule is a cost-equivalence, and allows us to merge the binders of two **let**s modulo binder collisions and variable capture. In turn, (var-expand) is an improvement that allows us to replace a binding with its binder provided there is a tick in front of the **let** to pay for this expansion. Also included in the tick algebra are ($\checkmark$-$\mathbb{E}$) and (case-$\mathbb{E}$) introduced previously.

The laws discussed above are only a small fragment of the tick algebra, however, it should be evident from these examples that applying such rules manually can be a difficult task. In particular, the use of different improvement relations, different kinds of contexts, and each rule having a unique syntactic form, makes it challenging to know when a rule can be applied correctly. Furthermore, many laws have side conditions, often concerning free and bound variables as with (case-$\mathbb{E}$) and (let-flatten), which must be checked every time they are applied.

A key strength of our system is that it provides mechanical support for all of these tasks, and moreover, it will automatically perform, for example, alpha-renaming to enable rules such as (let-flatten) to be applied correctly. Thus, the system allows the user to focus on the key aspects of their improvement calculations by handling tedious but important technical details on their behalf.

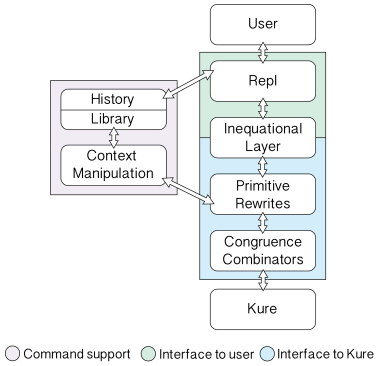


Fig. 4. Architecture of our system

4 System Architecture

The main components of our system are illustrated in Fig. 4. The *read-evaluate-print-loop* (Repl) handles interaction with the user. The *inequational layer* checks that transformation rules invoked by the user are safe to apply in the current proof state. The *primitive rewrites* and *congruence combinators* are basic building blocks used to define transformation rules in a modular manner [12], and are implemented using the Kure rewrite engine [17]. The *history* records successful commands entered by the user, and the resultant proof state of each command. In turn, the *library* maintains a collection of term, context, and cost-equivalent context definitions for use during proofs, together with a collection of command scripts that can be used to define transformation *sequences*. Finally, the *context manipulation* component supports the automatic generation, matching, and checking of the different kinds of program contexts.

4.1 Read-Eval-Print-Loop (Repl)

A necessary aspect of constructing interactive proofs is transforming sub-terms. We prioritise this requirement by maintaining a focus into the term being transformed, and providing navigation commands for changing the focus. Transformations are then applied to the sub-term currently in focus. By default, only the focused sub-term is displayed, which is updated each time a navigation command is executed. For situations when it may be beneficial to always display the whole term or some designated part, the system provides an option for the current focus to be highlighted. This feature is exhibited in Fig. 1.

4.2 Inequational Layer

Each time a transformation rule is invoked, the system checks that it is safe to apply in the current proof state. One aspect of this verification step is to ensure that the rule's corresponding operator entails the proof's improvement relation. If this is not the case, the transformation rule is rejected. Not only is this essential to ensuring well-formed calculations, it also permits users to safely experiment with improvement rules (for example, those from the tick algebra).

4.3 Primitive Rewrites, Congruence Combinators, and Kure

Similarly to the equational reasoning assistant Hermit [6], our system utilises the Kansas University Rewrite Engine (Kure) [17] for specifying and applying transformations to the abstract syntax of its operational model.

In brief, Kure is a strategic programming language [18] that provides a principled method for traversing and transforming data types. The fundamental idea behind Kure is to separate the implementations of traversals and the implementations of transformations. Traversal strategies and transformation rules can thus be reused and combined independently. For our system, this allows a sophisticated library of transformation rules, tailored to the needs of improvement theory, to be constructed by composing a small number of primitive operations using a selection of Kure's primitive combinators.

In addition, Kure's approach to datatype-generic programming [19] means that traversals can navigate to particular locations in order to apply type-specific transformations, giving fine control over when and where transformations are applied within a data type. This is vital for our implementation, as each reasoning step in an improvement proof typically transforms only a single sub-term.

Overall, our approach to implementing transformations rules using *primitive rewrites* and *congruence combinators* has been heavily inspired by the Hermit system, and builds on the work in [12,6]. We refer the reader to [6,17] for a detailed discussion of the relevant concepts.

4.4 Cost-Equivalent Contexts

The system maintains a library of *cost-equivalent contexts*. These are contexts whose syntactic forms do not satisfy the requirements for a particular kind of context (value, evaluation, applicative) but are nevertheless cost-equivalent to a context of this kind, and hence admit the same laws. Such contexts occur frequently in improvement proofs [1,2], as they lead to simplified reasoning steps. Once added by the user, cost-equivalent contexts are manipulated by the system in the same manner as other kinds of contexts (see section 6 for an example).

4.5 Context Manipulation

Managing contexts is one of the primary intricacies in constructing improvement proofs. In this section, we explain how this is handled by our system.

Context matching. In our system, a *context pattern* is simply a shorthand for a context, allowing sub-terms to be specified implicitly using wildcards and constructor patterns. For example, the context $\mathbf{let}\ \{x = a; y = b\}\ \mathbf{in}\ [-]$ may be described by any of the following context patterns (among others):

$$\mathbf{let}\ \{x = a; \_\}\ \mathbf{in}\ [-] \qquad \mathbf{let}\ \{x = \mathit{VAR}; \_\}\ \mathbf{in}\ [-] \qquad \mathbf{let}\ \_\ \mathbf{in}\ [-]$$

The underscores above are wildcards that match with any term, while VAR is a constructor pattern that matches with any variable.

Context patterns do not represent unique contexts, but when used in conjunction with a specific transformation rule, are often sufficient to determine a

unique context. In practice, they are used to simplify the amount of input required from a user interacting with the system. Recall the following rule, which allows ticks to be moved in and out of evaluation contexts:

$$\mathbb{E}[\checkmark M] \;\lhd\!\rhd\!\!\sim\; \checkmark\mathbb{E}[M] \qquad (\checkmark\text{-}\mathbb{E})$$

Suppose we wish to apply this rule to the term $\checkmark(a\ b\ c)$. To do so, we must determine an evaluation context $\mathbb{E}$ and a term M for which $\checkmark\mathbb{E}[M] \equiv \checkmark(a\ b\ c)$. In this case it is simple, such as by taking $\mathbb{E} = [-]\ b\ c$ and $M = a$. Applying $\checkmark$-$\mathbb{E}$ (right to left) then allows us to move the tick inside the context:

$$\begin{array}{ll} & \checkmark(a\ b\ c) \\ \lhd\!\rhd\!\!\sim & \{\ \checkmark\text{-}\mathbb{E} \quad [-]\ b\ c\ \} \\ & (\checkmark a)\ b\ c \end{array}$$

This transformation can be mirrored almost identically in our system:

```
unie> trans $`(a b c)$
✓(a b c)
[1]> untick-eval $[-] b c$
◁▷~ ✓a b c
```

Here the system uses the specified context $[-]\ b\ c$ and the given term $\checkmark(a\ b\ c)$ to verify the preconditions necessary for $\checkmark$-$\mathbb{E}$'s safe application. That is, it checks $[-]\ b\ c$ is a valid evaluation context and, by calculating the substituted term $M = a$, ensures the initial term has the required form $\checkmark\mathbb{E}[M]$. If any of these preconditions were not met, the transformation step would be rejected.

Suppose now that the context $\mathbb{E}$ from the above example was more complex, such as a **let** statement with multiple bindings. In this case, it would be impractical to expect the user to manually enter its full definition. Context patterns address this problem by allowing users to specify contexts by shorthand representations. The system uses these representations to automatically calculate valid contexts on the user's behalf, by *matching* the specified pattern against the syntax of the term being transformed. For example, we can use wildcard patterns to apply the above transformation in a simplified manner:

```
unie> trans $`(a b c)$
✓(a b c)
[1]> untick-eval $[-] _ _$
◁▷~ ✓a b c
```

Context generation. If we apply the same transformation rule as above, but without specifying a context parameter, the system will respond as follows:

```
unie> trans $`(a b c)$
✓(a b c)
[1]> untick-eval
Select a context/substitution option:
(1) E = [-],       M = a b c
(2) E = [-] c,     M = a b
(3) E = [-] b c,   M = a
```

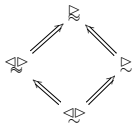
That is, three possible context/substitution pairs have been automatically *generated* by the system, each allowing $\checkmark$-$\mathbb{E}$ to be correctly applied to the given term. Option three corresponds to our previous choice:

```
[1]> 3
≂◁ ✓a b c
```

Context generation is available when applying any of the system's transformation rules. This feature has proven to be invaluable when validating proofs from the original improvement theory article [1], as the calculations in this article only specify the rules that are applied, and not how they are applied. Context generation often allows us to fill in these details automatically.

4.6 Inequational Reasoning

A central feature of our system is its support for inequational reasoning. The relationship between the different improvement relations that were introduced in section 3 are summarised in the following lattice:



An improvement proof is initiated by the user entering a proof statement, such as $\checkmark\!\!\checkmark x \trianglerighteq x$. The system uses this statement to establish a 'global' improvement relation, in this case $\trianglerighteq$. Each time a transformation is applied, the corresponding operator is checked to ensure that it entails this global relation in the above lattice. If this is not the case, the transformation is prevented from being applied and an error message is displayed. For example:

```
unie> trans $``x$ IMP $x$
Global relation set: ▷.
Transformation goal set: x.
✓✓x
[1]> tick-elim
▷ ✓x
[2]> untick-intro
Relation error: ◁ ⋣ ▷.
```

5 Worker/Wrapper Transformation

During the development of the Unie system, we were guided by the desire to mechanically verify all the results from the paper that renewed interest in improvement theory [2]. In this section, we review the main result of this paper, which shows that the worker/wrapper transformation is an improvement, and an example application of this result, which shows how a naive reverse function on lists can be improved to a more efficient version. In the next section, we will show how the latter result can be mechanised in our system.

5.1 Correctness

The worker/wrapper transformation is a technique for improving the performance of recursive programs by changing their types [20]. Given a recursive program of some type, the basic idea is to factorise the program into a *worker* program of a different type, together with a *wrapper* program that acts as an interface between the original program and the new worker. The intention is

that if the worker type supports more efficient operations than the original type, then this efficiency should result in a more efficient program overall.

More formally, suppose we are given a recursive program defined as the least fixed point *fix f* of a function f on some type A. Now consider a more efficient program that performs the same task, defined by first taking the least fixed point *fix g* of a function g on some other type B, and then migrating the resulting value back to the original type by applying a conversion function *abs*. The equivalence between the two programs is captured by the following equation:

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

This equation states that the original program *fix f* can be factorised into the application of a wrapper function *abs* to a worker program *fix g*. As one may expect, the validity of the equation depends on some properties of the functions f, g, and *abs*, together with a dual conversion function *rep*. These properties are summarised in the following worker/wrapper correctness theorem [21]: given functions $f : A \to A$, $g : B \to B$, $\mathit{abs} : B \to A$, and $\mathit{rep} : A \to B$ satisfying one of the assumptions (A–C) and one of the conditions (1–3),

$$\begin{array}{llll} \text{(A)} & \mathit{abs} \circ \mathit{rep} & = & \mathit{id}_A \\ \text{(B)} & \mathit{abs} \circ \mathit{rep} \circ f & = & f \\ \text{(C)} & \mathit{fix}\ (\mathit{abs} \circ \mathit{rep} \circ f) & = & \mathit{fix}\ f \end{array} \qquad \begin{array}{llll} \text{(1)} & g & = & \mathit{rep} \circ f \circ \mathit{abs} \\ \text{(2)} & g \circ \mathit{rep} & = & \mathit{rep} \circ f \\ \text{(3)} & f \circ \mathit{abs} & = & \mathit{abs} \circ g \end{array}$$

then we have the correctness equation *fix f* = *abs* (*fix g*).

5.2 Improvement

The previous section formalised that the worker/wrapper transformation is correct, in the sense that the original and new programs have the same denotational meaning. We now formalise that the transformation improves efficiency, in the sense that the new program improves the runtime performance of the original.

To reformulate the correctness theorem as an improvement theorem, we must first make some changes to the basic setup to take account of the differences between the underlying denotational and operational theories. In particular, functions are replaced by contexts, i.e. the functions f and g become contexts $\mathbb{F}$ and $\mathbb{G}$; the use of a *fix* operator is replaced by recursive **let** bindings, i.e. *fix f* becomes $\mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ x$; and the use of equality is replaced by an appropriate improvement relation, i.e. $=$ becomes $\trianglerighteq$, $\bigapprox$ or $\lhd\!\!\rhd$. Using these modifications, we have the following worker/wrapper improvement theorem [2]: given value contexts $\mathbb{F}$, $\mathbb{G}$, $\mathtt{Abs}$, and $\mathtt{Rep}$ satisfying one of the assumptions (where x is free)

$$\begin{array}{lll} \text{(A)} & \mathtt{Abs}[\mathtt{Rep}[x]] & \lhd\!\!\rhd\ x \\ \text{(B)} & \mathtt{Abs}[\mathtt{Rep}[\mathbb{F}[x]]] & \lhd\!\!\rhd\ \mathbb{F}[x] \\ \text{(C)} & \mathbf{let}\ x = \mathtt{Abs}[\mathtt{Rep}[\mathbb{F}[x]]]\ \mathbf{in}\ x & \lhd\!\!\rhd\ \mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ x \end{array}$$

and one of the conditions

$$\begin{array}{lll} \text{(1)} & \mathbb{G}[x] & \bigapprox\ \mathtt{Rep}[\mathbb{F}[\mathtt{Abs}[x]]] \\ \text{(2)} & \mathbb{G}[\checkmark\mathtt{Rep}[x]] & \lesssim\ \mathtt{Rep}[\checkmark\mathbb{F}[x]] \\ \text{(3)} & \mathtt{Abs}[\checkmark\mathbb{G}[x]] & \bigapprox\ \mathbb{F}[\checkmark\mathtt{Abs}[x]] \end{array}$$

then we have the improvement inequality $\mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ x \gtrapprox \mathbf{let}\ x = \mathbb{G}[x]\ \mathbf{in}\ \mathtt{Abs}[x]$. The assumptions and conditions above that ensure the original recursive program $\mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ x$ is improved by $\mathbf{let}\ x = \mathbb{G}[x]\ \mathbf{in}\ \mathtt{Abs}[x]$ are natural extensions of the corresponding properties for correctness. For example, correctness condition (1), $g = rep \circ f \circ abs$, is replaced by improvement condition (1), $\mathbb{G}[x] \lessapprox \mathtt{Rep}[\mathbb{F}[\mathtt{Abs}[x]]]$. Note that because improvement theory is untyped, there are no typing requirements on the contexts.

The proof of the above theorem utilises two other results: a 'rolling' rule and a fusion rule. Both are central to the worker/wrapper transformation [20], and can be proven using tick algebra laws. Consequently, the worker/wrapper improvement theorem is itself a direct result of the tick algebra's inequational theory. All aforementioned results have been verified using our system [13].

5.3 Example

Consider the following naive definition for the *reverse* function on lists:

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

Here the function is defined using a recursive **let** binding rather than explicit recursion, and the context `Revbody` captures the non-recursive part of the function's definition. This implementation is inefficient due to the use of the append operator $+\!\!+$, which takes linear time in the length of its first argument. We would like to use the worker/wrapper technique to improve it.

The first step is to select a new type to replace the original type $[\,a\,] \to [\,a\,]$, and define contexts to perform the conversions between the two types. In this case, we utilise the type $[\,a\,] \to [\,a\,] \to [\,a\,]$ that provides an additional argument that is used to accumulate the resulting list [14]. The contexts to convert between the original and new types are then defined as follows [2]:

$$\mathtt{Abs} = \lambda \mathit{xs}.[-]\ \mathit{xs}\ [\,] \qquad\qquad \mathtt{Rep} = \lambda \mathit{xs}.\lambda \mathit{ys}.[-]\ \mathit{xs} +\!\!+ \mathit{ys}$$

We must now verify that the conversion contexts `Abs` and `Rep` satisfy one of the worker/wrapper assumptions. We verify assumption (B) as follows:

$$
\begin{aligned}
& \mathtt{Abs}[\mathtt{Rep}[\mathtt{Revbody}[f]]] \\
\equiv\ & \{\ \text{apply definitions of } \mathtt{Abs} \text{ and } \mathtt{Rep}\ \} \\
& \lambda \mathit{xs}.(\lambda \mathit{xs}.\lambda \mathit{ys}.\mathtt{Revbody}[f]\ \mathit{xs} +\!\!+ \mathit{ys})\ \mathit{xs}\ [\,] \\
\lhd\!\rhd\ & \{\ \beta\text{-reduction}\ \} \\
& \lambda \mathit{xs}.\mathtt{Revbody}[f]\ \mathit{xs} +\!\!+ [\,] \\
\equiv\ & \{\ \text{apply definition of } \mathtt{Revbody}\ \} \\
& \lambda \mathit{xs}.(\lambda \mathit{xs}.\mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
& \quad [\,] \qquad\ \to [\,] \\
& \quad (y : \mathit{ys}) \to f\ \mathit{ys} +\!\!+ [\,y\,])\ \mathit{xs} +\!\!+ [\,] \\
\lhd\!\rhd\ & \{\ \beta\text{-reduction}\ \} \\
& \lambda \mathit{xs}.(\mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
& \quad [\,] \qquad\ \to [\,] \\
& \quad (y : \mathit{ys}) \to f\ \mathit{ys} +\!\!+ [\,y\,]) +\!\!+ [\,] \\
\lhd\!\rhd\ & \{\ \text{case-}\mathbb{E}\ \text{rule, where } \mathbb{E} \equiv [-] +\!\!+ [\,]\ \} \\
& \lambda \mathit{xs}.\mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
& \quad [\,] \qquad\ \to [\,] +\!\!+ [\,] \\
& \quad (y : \mathit{ys}) \to (f\ \mathit{ys} +\!\!+ [\,y\,]) +\!\!+ [\,] \\
\lhd\!\rhd\ & \{\ \text{associativity of } +\!\!+\ \} \\
& \lambda \mathit{xs}.\mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
& \quad [\,] \qquad\ \to [\,] +\!\!+ [\,] \\
& \quad (y : \mathit{ys}) \to f\ \mathit{ys} +\!\!+ ([\,y\,] +\!\!+ [\,]) \\
\lhd\!\rhd\ & \{\ \text{evaluate } [\,] +\!\!+ [\,] \text{ and } [\,y\,] +\!\!+ [\,]\ \} \\
& \lambda \mathit{xs}.\mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
& \quad [\,] \qquad\ \to [\,] \\
& \quad (y : \mathit{ys}) \to f\ \mathit{ys} +\!\!+ [\,y\,] \\
\equiv\ & \{\ \text{unapply definition of } \mathtt{Revbody}\ \} \\
& \mathtt{Revbody}[f]
\end{aligned}
$$

Note that the above calculation uses the fact that $+\!\!+$ is associative up to weak cost-equivalence, that is, $(xs +\!\!+ ys) +\!\!+ zs \lhd\!\!\wr\!\!\rhd xs +\!\!+ (ys +\!\!+ zs)$.

Next we must verify that one of the worker/wrapper conditions is satisfied. In this example, we can use condition (2) as a *specification* for the context $\mathbb{G}$, whose definition can then be calculated using laws from the tick algebra. We omit the details here for brevity, but they are included in the original paper [2], and result in the following context definition:

$$\begin{array}{l}\mathbb{G} = \lambda xs.\lambda ys.\mathbf{case}\ xs\ \mathbf{of}\\ \quad [\,] \quad\ \ \to ys\\ \quad (z : zs) \to \mathbf{let}\ ws = (z : ys)\ \mathbf{in}\ [-]\ zs\ ws\end{array}$$

The crucial step in the construction of $\mathbb{G}$ is applying property (2) from our example in section 2, which expresses that reassociating append to the right is an improvement, i.e. $(xs +\!\!+ ys) +\!\!+ zs \trianglerighteq xs +\!\!+ (ys +\!\!+ zs)$.

Finally, if we define $\mathit{fastrev} = \mathbf{let}\ x = \mathbb{G}[x]\ \mathbf{in}\ \mathtt{Abs}[f]$, then by applying the worker/wrapper improvement theorem, we have shown that the original version of *reverse* is improved by the new version, i.e. $\mathit{reverse} \wr\!\!\rhd \mathit{fastrev}$. Expanding out the definition of *fastrev* and renaming/simplifying the resulting **let** binding gives the familiar fast version of the original function:

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

6 Mechanising Fast Reverse

In this section, we demonstrate how to improve the naive reverse function mechanically using our system. In doing so, we illustrate a number of the system's key features, and show how it supports interactive reasoning using transformation and navigation rules. All of the interaction is taken directly from the system itself, with some minor reformatting for the paper-based medium.

As in the previous section, we focus on the proof of assumption (B). Prior to constructing the proof, we must ensure that the system has access to the definitions from section 5, which are required at different stages throughout. For convenience we have stored them in a file, which is imported into the system's library using the `import-lib` command, and the names of the new definitions displayed using `show-lib defs`. We have also included the definition of $+\!\!+$, as this is required in a number of proof steps involving evaluation.

```
unie> import-lib ./libs/reverse
Info: library updated.

unie> show-lib defs
Terms:    (++), reverse
Contexts: Abs, Rep, Revbody
```

We instruct the system to enter its transformation mode using `trans`. The relevant proof statement $\mathtt{Abs}[\mathtt{Rep}[\mathtt{Revbody}[f]]] \lhd\!\!\wr\!\!\rhd \mathtt{Revbody}[f]$ is supplied as a parameter, and determines the proof's global relation and goal. The global relation will prevent rules being applied whose operators do not entail weak cost-equivalence $\lhd\!\!\wr\!\!\rhd$, and we will be notified when the goal $\mathtt{Revbody}[f]$ is reached.

When entering terms into the system, the kinds of contexts must be specified. `Abs`, `Rep`, and `Revbody` are value contexts, so we use the `V_` prefix.

```
unie> trans $V_Abs[V_Rep[V_Revbody[f]]]$ WCE $V_Revbody[f]$
Global relation set: ◁▷.
Transformation goal set: V_Revbody[f].
```

As with the paper proof, we begin by applying the definitions of `Abs` and `Rep`, and beta-reducing inside the body of the outer lambda abstraction. In order to reduce the correct sub-terms, we must navigate using `left` and `right`, which move the focus to the current terms left and right child, respectively. The last step uses `top`, which restores focus to the full term.

```
V_Abs[V_Rep[V_Revbody[f]]]
[1]> apply-def 'Abs ; apply-def 'Rep
≡ λxs.(λxs.λys.(V_Revbody[f] xs) ++ ys) xs []
[3]> right
≡ (λxs.λys.(V_Revbody[f] xs) ++ ys) xs []
[4]> left
≡ (λxs.λys.(V_Revbody[f] xs) ++ ys) xs
[5]> beta
◁▷ λys.(V_Revbody[f] xs) ++ ys
[6]> up ; beta
◁▷ (V_Revbody[f] xs) ++ []
[8]> top
≡ λxs.(V_Revbody[f] xs) ++ []
```

Next we apply the definition of `Revbody` and beta-reduce the resulting redex. We then move `up` to focus on the application of append.

```
[9]> apply-def 'Revbody
≡ λxs.((λxs.case xs of
    []     → []
    (y:ys) → (f ys) ++ [y]) xs) ++ []
[10]> right ; left
≡ (++) ((λxs.case xs of
    []     → []
    (y:ys) → (f ys) ++ [y]) xs)
[12]> right
≡ (λxs.case xs of
    []     → []
    (y:ys) → (f ys) ++ [y]) xs
[13]> beta
◁▷ case xs of
    []     → []
    (y:ys) → (f ys) ++ [y]
[14]> up ; up
≡ (case xs of
    []     → []
    (y:ys) → (f ys) ++ [y]) ++ []
```

Now recall the case-$\mathbb{E}$ rule, which allows an evaluation context to be moved inside a **case** statement (subject to certain conditions regarding free and bound variables, which are automatically checked by our system):

$$\mathbb{E}[\textbf{case}\ M\ \textbf{of}\ \{pat_i \to N_i\}] \quad \lhd\!\rhd \quad \textbf{case}\ M\ \textbf{of}\ \{pat_i \to \mathbb{E}[N_i]\}$$

Here we would like to use this rule to move $+\!\!+\ []$ inside the **case** statement. We know that the system can generate evaluation contexts, so we can attempt to apply the rule without specifying a parameter:

```
[16]> case-eval
Error: no valid evaluation contexts.
```

However, an error results because the context $[-] +\!\!+ []$ we wish to use is not strictly speaking an evaluation context, but is only cost-equivalent to an evaluation context [2]. By default, only contexts of the correct syntactic form are accepted by the system, meaning that even if we manually specified the desired context as a parameter to `case-eval`, it would be rejected as invalid.

The solution is to add $[-] +\!\!+ []$ to the library of cost-equivalent evaluation contexts, which allows the system to treat it as if it were strictly an evaluation context. This is done using the `add-lib` command. In fact, the proof in [2] is more general than this particular example, and shows that $[-] +\!\!+ xs$ is cost-equivalent to an evaluation context for any list xs. This can be captured using the constructor pattern $LIST$ that matches with any list:

```
[16]> add-lib EVAL $[-] ++ LIST$
Info: library updated.
```

Cost-equivalent contexts are incorporated into the system's context generation and matching mechanisms, meaning that when we apply `case-eval` again without a parameter, the correct context is used automatically. Note that in this example, the context pattern $[-] +\!\!+ LIST$ has been instantiated to $[-] +\!\!+ []$ in order to apply the transformation rule correctly.

```
[16]> case-eval
◁▷≈ case xs of
      []     → [] ++ []
      (y:ys) → ((f ys) ++ [y]) ++ []
```

We have almost completed the proof. All that is left to do is evaluate the applications of append that have resulted from $+\!\!+ []$ being moved inside both alternatives in the **case** statement. Note that in the second alternative, we wish to evaluate $[y] +\!\!+ []$. In order to do so we must first reassociate the term using the fact that append is associative up to weak cost-equivalence.

```
[17]> right ; rhs
≡ [] ++ []
[19]> eval-wce
◁▷≈ []
[20]> up ; next ; rhs
≡ ((f ys) ++ [y]) ++ []
[23]> append-assoc-lr-wce
◁▷≈ (f ys) ++ ([y] ++ [])
[24]> right ; eval-wce
◁▷≈ [y]
[26]> top
≡ λxs.case xs of
      []     → []
      (y:ys) → (f ys) ++ [y]
```

Finally, we unapply the definition of `Revbody` and are notified that we have reached our goal. The proof of the property is complete:

```
[27]> unapply-def 'Revbody
Info: transformation goal reached!
≡ V_Revbody[f]
```

In conclusion, the above calculation demonstrates how improvement proofs can be constructed using our system. By following the same pattern as the original paper proof, with the addition of navigation steps to make the point of application of each rule clear, we were able to mechanise the calculation by simply entering the transformation rules as commands into the system. Behind the scenes, the technicalities of each proof step were administered automatically on our behalf to ensure the resulting proof is correct. Moreover, by entering commands without parameters, we allowed the system to simplify the development of proof steps by automatically generating the necessary contexts.

7 Related Work

Several tools have been developed to mechanise *equational* reasoning about Haskell programs [22,23,24,25]. Most relevant to our system is Hermit [6], which builds upon the Haskell Equational Reasoning Assistant (Hera) [26]. There appears to be no other systems in the literature that directly support *inequational* reasoning about Haskell programs, and to the best of our knowledge, our system is the first to provide mechanical support for improving Haskell programs.

In other languages, the Algebra of Programming in Agda (AoPA) library [27] is designed to encode relational program derivations, which supports a form of inequational reasoning. The Jape proof calculator [28,29] provides step-by-step interactive development of proofs in formal logics, and supports both equational and inequational reasoning. Improvement theory has not been explored within either of these settings, however. More generally, automated theorem provers [30,31] can be used to provide formal, machine-checked proofs of program properties, but require expertise in dependently-typed programming.

Other methods for reasoning about time performance in a lazy setting include [32,33]. Most notably is the work of Okasaki [34], who used a notion of *time credits* to analyse the amortized performance of a range of purely functional data structures. This approach has recently been implemented in Agda [35]. Research has also been conducted on type-based methods for cost analysis, for example in [36,37], but in general these frameworks do not incorporate call-by-need semantics. GHC itself provides *cost centres*, which can be used to annotate source code so that the GHC profiler can indicate which parts of a program cost the most to execute. A formal cost semantics for GHC core programs based on the notion of cost centres is presented in [38].

8 Conclusion and Further Work

In this article, we have presented the design and implementation of an inequational reasoning assistant that provides mechanical support for proofs of program improvement. In doing so, we have highlighted a number of difficulties in manually constructing improvement proofs, and described how the system has been developed to address these challenges. We have illustrated the applicability of our system by verifying a range of improvement results from the literature. Specifically, we have mechanised all proofs in [2], including the proof of the worker/wrapper improvement theorem, which relates to a highly general optimisation technique. We have also mechanically verified a number of proofs in [1]. All of

these proofs are freely available online as scripts that can be loaded into our system, along with the system itself [13].

We have three main avenues for further work. First of all, we would like to investigate higher-level support for navigating through terms during improvement proofs, for which we expect to be guided by our experience using the Hermit system [6]. Secondly, we would like to extend our system to produce proof objects that can be independently checked using an external proof assistant such as Coq or Agda, to provide a formal guarantee of their correctness. And finally, we are also interested in lightweight approaches to verifying improvement properties, for example, in a similar manner to which the QuickCheck [39] system supports lightweight verification of correctness properties.

Acknowledgements

We'd like to thank Jennifer Hackett and Neil Sculthorpe for many useful discussions, and the anonymous referees for their useful suggestions. This work was funded by EPSRC grant EP/P00587X/1, *Mind the Gap: Unified Reasoning About Program Correctness and Efficiency.*

References

1. Moran, A.K., Sands, D.: Improvement in a Lazy Context: An Operational Theory for Call-By-Need. Extended version of [40]. (1999)
2. Hackett, J., Hutton, G.: Worker/Wrapper/Makes It/Faster. In: ICFP. (2014)
3. Schmidt-Schauß, M., Sabel, D.: Improvements in a Functional Core Language with Call-by-need Operational Semantics. In: PPDP. (2015)
4. Hackett, J., Hutton, G.: Parametric Polymorphism and Operational Improvement. In preparation, University of Nottingham (2017)
5. Harper, R.: The Structure and Efficiency of Computer Programs. Carnegie Mellon University (2014)
6. Farmer, A.: Hermit: Mechanized Reasoning During Compilation in the Glasgow Haskell Compiler. PhD thesis, University of Kansas (2015)
7. Sculthorpe, N., Farmer, A., Gill, A.: The Hermit in the Tree: Mechanizing Program Transformations in the GHC Core Language. In: IFL. (2013)
8. Farmer, A., Höner zu Siederdissen, C., Gill, A.: The Hermit in the Stream. In: PEMP. (2014)
9. Farmer, A., Sculthorpe, N., Gill, A.: Reasoning with the Hermit: Tool Support for Equational Reasoning on GHC Core Programs. In: Haskell Symposium. (2015)
10. Adams, M.D., Farmer, A., Magalhães, J.P.: Optimizing SYB Is Easy! In: PEPM. (2014)
11. Adams, M.D., Farmer, A., Magalhães, J.P.: Optimizing SYB Traversals Is Easy! Science of Computer Programming (2015)
12. Farmer, A., Gill, A., Komp, E., Sculthorpe, N.: The Hermit in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In: Haskell Symposium. (2012)
13. Handley, M.A.T.: GitHub Repository for the University of Nottingham Improvement Engine (Unie) (2017) `https://github.com/mathandley/Unie`.
14. Wadler, P.: The Concatenate Vanishes. University of Glasgow (1987)
15. Sestoft, P.: Deriving a Lazy Abstract Machine. JFP **7**(3) (1997)
16. Ariola, Z.M., Maraist, J., Odersky, M., Felleisen, M., Wadler, P.: A Call-By-Need Lambda Calculus. In: POPL. (1995)

17. Sculthorpe, N., Frisby, N., Gill, A.: The Kansas University Rewrite Engine. JFP **24** (7 2014)
18. Lämmel, R., Visser, E., Visser, J.: The Essence of Strategic Programming. (2002)
19. Gibbons, J.: Datatype-Generic Programming. LNCS **419** (2006)
20. Gill, A., Hutton, G.: The Worker/Wrapper Transformation. JFP **19**(2) (2009)
21. Sculthorpe, N., Hutton, G.: Work It, Wrap It, Fix It, Fold It. JFP **24**(1) (2014)
22. Tullsen, M.A.: Path, A Program Transformation System for Haskell. Yale University, PhD Thesis. (2002)
23. Guttmann, W., Partsch, H., Schulte, W., Vullinghs, T.: Tool Support for the Interactive Derivation of Formally Correct Functional Programs. Journal of Universal Computer Science (2003)
24. Thompson, S., Li, H.: Refactoring Tools for Functional Languages. JFP **23**(3) (2013)
25. Li, H., Reinke, C., Thompson, S.: Tool Support for Refactoring Functional Programs. In: Haskell Workshop. (2003)
26. Gill, A.: Introducing the Haskell Equational Reasoning Assistant. In: Haskell Workshop. (2006)
27. Mu, S.C., Ko, H.S., Jansson, P.: Algebra of Programming in Agda: Dependent Types for Relational Program Derivation. JFP **19**(5) (2009)
28. Bornat, R., Sufrin, B.: Jape: A Calculator for Animating Proof-On-Paper. Automated Deduction (1997)
29. Bornat, R., Sufrin, B.: Animating Formal Proof at the Surface: The Jape Proof Calculator. The Computer Journal **42**(3) (1999)
30. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer (2013)
31. Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory. PhD thesis, Chalmers University of Technology (2007)
32. Wadler, P.: Strictness Analysis Aids Time Analysis. In: POPL. (1988)
33. Bjerner, B., Holmström, S.: A Composition Approach to Time Analysis of First Order Lazy Functional Programs. In: FPCA. (1989)
34. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1999)
35. Danielsson, N.A.: Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In: POPL. (2008)
36. Brady, E., Hammond, K.: A Dependently Typed Framework for Static Analysis of Program Execution Costs. In: IFL. (2005)
37. Çiçek, E., Barthe, G., Gaboardi, M., Garg, D., Hoffmann, J.: Relational Cost Analysis. In: POPL. (2017)
38. Sansom, P.M., Peyton Jones, S.L.: Formally Based Profiling for Higher-Order Functional Languages. TOPLAS (1997)
39. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. ICFP (2011)
40. Moran, A.K., Sands, D.: Improvement in a Lazy Context: An Operational Theory for Call-By-Need. In: POPL. (1999)