Parametric Polymorphism and Operational Improvement

JENNIFER HACKETT and GRAHAM HUTTON, University of Nottingham, UK

Parametricity, in both operational and denotational forms, has long been a useful tool for reasoning about program correctness. However, there is as yet no comparable technique for reasoning about program *improvement*, that is, when one program uses fewer resources than another. Existing theories of parametricity cannot be used to address this problem as they are agnostic with regard to resource usage. This article addresses this problem by presenting a new operational theory of parametricity that is sensitive to time costs, which can be used to reason about time improvement properties. We demonstrate the applicability of our theory by showing how it can be used to prove that a number of well-known program fusion techniques are time improvements, including fixed point fusion, map fusion and short cut fusion.

CCS Concepts: • **Software and its engineering → Functional languages**; **Polymorphism**; • **Theory of computation** → *Operational semantics*;

ACM Reference Format:
Jennifer Hackett and Graham Hutton. 2018. Parametric Polymorphism and Operational Improvement. N/A, ICFP (July 2018), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Parametric polymorphism is everywhere. In typed functional languages, many if not most of the built-in and user-defined functions are parametric. Because of this ubiquity, we must carefully study the properties of parametric functions. Chief among these properties is the *abstraction theorem* [Reynolds 1983], which shows that any well-typed term must satisfy a property that can be derived uniformly from its type. By instantiating this theorem for specific types, we obtain the so-called "free theorems" of Wadler [1989], properties held by any term of a given type.

The abstraction theorem was first presented by Reynolds [1983], who proved it using the notion of *relational* parametricity. In relational parametricity, one starts with a denotational semantics based on sets (or more generally, some form of domain) and builds on top of it another denotational semantics based on relations between those sets. It can then be shown that interpreting any term in related contexts must produce related results. Deriving the free theorem is then a matter of calculating the relational interpretation of the type in question.

The original denotational presentation of the abstraction theorem is promising, as it suggests that similar theorems will exist for any System F-like language. However, it is often not obvious what a parametric model of such a language should be. For this reason, it is helpful to investigate more *operational* notions of parametricity such as the version developed by Pitts [2000], where the relations we work with are between terms rather than between the interpretations of terms. The result is an abstraction theorem that respects observational equivalence, provided the relations involved satisfy the intuitive property of $\top\top$-closure.

Authors' address: Jennifer Hackett; Graham Hutton, School of Computer Science, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham, NG8 1BB, UK, {jennifer.hackett,graham.hutton}@nottingham.ac.uk.

Parametricity can be used to give correctness proofs of a number of useful program optimisations, most notably short cut fusion [Gill et al. 1993]. However, correctness is only one side of optimisations: we must also consider whether transformations *improve* performance. In order to carry correctness results into this setting we need a *resource-aware* theory of parametricity, to provide us with free theorems that include information about efficiency properties.

In this article we develop a new operational theory of parametricity that can be used to reason about *time improvement*, i.e. when one term can be replaced by another without increasing time cost. Our theory is built on a standard lazy abstract machine [Sestoft 1997], making it applicable to call-by-need languages such as Haskell. Specifically, we make the following contributions:

- We show how Pitts' notion of $\top\top$-closure can be adapted to produce a resource-aware notion that we call machine-closure. The key idea is that whereas $\top\top$-closed relations respect observational equivalence, machine-closed relations respect *time improvement*.
- We use the notion of machine-closure to prove an abstraction theorem for call-by-need programs that use recursion in a restricted manner, namely when the right-hand side of the recursive binding is in *value* form. The resulting theorem can be used to reason about time improvement properties in call-by-need languages such as Haskell.
- We demonstrate the application of our abstraction theorem by justifying a number of fusion-based optimisations as time improvements, including short cut fusion. We focus on fusion as most parametricity-based optimisations are instances of fusion.

This work has similar aims to that of Seidel and Voigtländer [2011], who investigate efficiency-based free theorems in a call-by-value language, but differs in setting and approach. Firstly, we consider call-by-need rather than call-by-value. Secondly, their work is based on a denotational semantics instrumented with costs and it is not clear how this can be applied in a call-by-need setting, so instead we use an operational semantics with an explicit stack and heap. Finally, our work builds on the call-by-need improvement theory of Moran and Sands [1999a], using an improvement relation to abstract away from explicit costs where possible.

This paper is part of a wider project to make questions of call-by-need efficiency more tractable [Hutton and Hackett 2016]. By developing new techniques for questions of improvement that are compatible with the existing techniques used to prove correctness, we seek to bring the two issues of correctness and improvement closer together, reducing the work that must be done to formally justify a particular program transformation. In this case, a technique based on parametricity for reasoning about improvement makes it easier to reason about the efficiency aspects of program transformations that rely on parametricity for their correctness properties.

2 BACKGROUND

We begin in this section with some background on the two key technical elements that underpin our work, namely parametric polymorphism and operational improvement.

2.1 Parametric Polymorphism

The viewpoint of parametric polymorphism is that a polymorphic function must do the *same* thing at every type. This contrasts with ad-hoc polymorphism [Strachey 2000], where it is only required that a function do *something* at every type. The result is that parametrically polymorphic functions are forced to respect the abstractions of the calling code, being prevented from inspecting the structure of the type at which they are called. This property was first observed by Reynolds [1983], who proved the *abstraction theorem* for the polymorphic λ-calculus.

Reynolds' abstraction theorem works by first defining a set-theoretic denotational semantics, where types are interpreted as sets and terms as elements of those sets, and then building on top of this a semantics of *logical relations*. These relations are built by defining relation-lifted definitions of the type constructors of the language. For example, two terms at a function type are related if they take related arguments to related results:

$$(f, g) \in R \rightarrow S \quad \Leftrightarrow \quad \forall\, (x, y) \in R.\ (f\ x, g\ y) \in S$$

Once we have this relational interpretation of types, it can be shown by a straightforward induction on the structure of type derivations that interpreting any term in related environments will give related results. This model can be extended with extra constructs such as general recursion and sequencing, which translate to adding restrictions on the relations [Johann and Voigtländer 2004]. However, this method is limited to languages with a natural denotational semantics, and so cannot be applied when the only natural semantics is operational.

Pitts [2000] addressed this issue by presenting an *operational* treatment of parametricity for a language called PolyPCF, a version of Plotkin's PCF [Plotkin 1977] extended with list types and polymorphism. The same technique of building relations from the structure of types is used, but these relations are between terms rather than elements of a denotational semantics. It is therefore necessary to require that the relations respect equivalence in the operational semantics.

To ensure that relations respect equivalence, Pitts introduced two notions for PolyPCF language. Firstly, there is the $\top$ relation that holds between stacks (which function as term contexts) and terms whenever the stack applied to the term will produce the empty list *Nil*. Secondly, there is the $\top\top$-closure operator for term relations, so called because it involves using the $\top$ relation twice: once to go from a relation on terms to one on stacks, and once to go back again. As a consequence of the definition, all $\top\top$-closed relations respect equivalence.

We can summarise the notion of $\top\top$-closure as follows. Given a relation $R : \tau_1 \leftrightarrow \tau_2$ between types τ_1 and τ_2, its $\top\top$-closure $R^{\top\top}$ is a relation of the same type. A pair (M_1, M_2) is in $R^{\top\top}$ if:

```
for all stacks $S_1$, $S_2$,
  if for all $(N_1, N_2) \in R$, $S_1 \top N_1 \Leftrightarrow S_2 \top N_2$
    then $S_1 \top M_1 \Leftrightarrow S_2 \top M_2$
```

Pitts shows that $\top\top$-closed relations are closed under the relational versions of function space formation, type abstraction and list type formation, thus demonstrating that they are a suitable notion of relation on terms. It can then be shown that any closed term is related to itself, and that open terms are related to themselves provided that the relational interpretations of the free type variables are all $\top\top$-closed. Applying this theorem is then a matter of instantiating with particular relations, provided these relations can be shown to be $\top\top$-closed.

2.2 Operational Improvement

In order to reason about the operational efficiency of programs, we need some model of the cost of terms. For call-by-value this is straightforward: in most cases it is enough simply to count the steps taken to evaluate the term to normal form; functions are slightly more complicated as we have both the cost to evaluate the function itself and the cost to compute its result, where the latter may depend on the argument to the function [Shultis 1985]. The result is a semantics of call-by-value λ-terms that can be used to reason about time costs.

The situation for call-by-need is more complicated, however. In this case, a subterm is only evaluated when it is *forced*, i.e. when its value is required, so the cost to evaluate a term to normal form is a poor measure of efficiency. For example, the terms $[\perp]$ and $[0]$ are both in normal form, but one is potentially more efficient than the other: e.g. *head* $[\perp]$ diverges while *head* $[0]$ returns almost immediately. One might consider making this analysis recursive, comparing the cost of evaluating subterms as well, but this method will identify the terms **let** $x = M$ **in** (x, x) and (M, M) even though one shares work in a way the other does not.

The solution of Moran and Sands [1999a] is to quantify over evaluation contexts. That is to say, a term M is *improved by* another term N, written $M \gtrsim N$, if for all contexts $\mathbb{C}$, the term $\mathbb{C}[M]$ takes at least as many steps to evaluate as $\mathbb{C}[N]$. By taking into account all possible uses of the initial terms, we automatically take into account cost savings introduced by sharing as well as the cost of computing subterms. Furthermore, this notion of efficiency is compositional by definition, making it amenable to techniques of inequational reasoning.

Moran and Sands' improvement theory has been used to justify general-purpose program improvements, such as the worker/wrapper transformation [Hackett and Hutton 2014]. This takes advantage of the parallels between the rules of call-by-need improvement and those of program *equivalence*, which makes it possible to adapt proofs of correctness into proofs of improvement. Subsequent work took this idea of having compatible equivalence and improvement proofs and placed it in a generalised setting of *preorder-enriched categories* [Hackett and Hutton 2015].

Improvement theory and related techniques have seen something of a resurgence in recent years. Breitner [2015] uses an operational approach to proving the safety of the call arity transformation based on counting the number of allocations made. Sergey et al. [2017] use a step-counting approach to show that floating **let** bindings into a lambda is an improvement provided the lambda expression is used only once. Schmidt-Schauß and Sabel [2015] use an operational semantics based on term rewriting rules to show that a number of local optimisations are improvements, including common subexpression elimination. Finally, Simões et al. [2012] define a cost model for Launchbury's natural semantics [Launchbury 1993], and use it to prove soundness of a cost analysis.

3 FROM EQUIVALENCE TO IMPROVEMENT

In this section, we show how to build a theory of parametricity that can be used to reason about program improvements. Essentially we want to do for improvement what Pitts did for equivalence. Unfortunately, Pitts' notion of $\top\top$-closure is too strong for our purposes: because $\top\top$-closed relations must respect program equivalence, they cannot be used to distinguish between observationally equivalent programs with different costs. In a sense, Pitts' notion of observation is too narrow for our purposes. What we need is a notion of closure that forces relations to respect only *cost-equivalence*, i.e. when two programs are interchangeable in terms of time costs.

3.1 Call-By-Need PolyPCF

We consider a simple language, a polymorphically-typed λ-calculus with recursive bindings. The type and term grammars are defined as follows:

$$\begin{array}{ll}
\alpha \in \mathit{TVar} & M,\ N \in \mathit{Term} ::= \\
x,\ y,\ xs \in \mathit{Var} & \quad x \\
A,\ B \in \mathit{Type} ::= \alpha & \quad \mid\ \lambda x.M \\
\qquad\qquad \mid\ A \to B & \quad \mid\ M\ x \\
\qquad\qquad \mid\ \mathit{List}\ A & \quad \mid\ \Lambda\alpha.M \\
\qquad\qquad \mid\ \forall\,\alpha.A & \quad \mid\ M\ A \\
 & \quad \mid\ \mathbf{let}\ \{\vec{x} = \vec{M}\}\ \mathbf{in}\ N \\
 & \quad \mid\ \mathit{Nil} \\
 & \quad \mid\ x :: xs \\
 & \quad \mid\ \mathbf{case}\ M\ \mathbf{of}\ \{\mathit{Nil} \to M_1; x :: xs \to M_2\}
\end{array}$$

The typing rules for the language are given in Figure 1, and comprise one rule for each language construct. Note that the typing context Γ contains both free type variables and type assignments for free term variables. We apply a standard well-formedness constraint to Γ.

Our chosen language is similar to the PolyPCF language studied by Pitts, but with two key differences. Firstly, we have added recursive let-bindings to the language and remove the fixed point combinator *fix*. We do this because let-binding-based presentations of recursion are better at capturing sharing. Secondly, we only allow functions and constructors to be applied to variables. This latter distinction is useful because it means that all sharing is explicitly introduced in let-bindings, making it easier to reason about call-by-need evaluation; a similar restriction applied to earlier versions of the internal language used in the Glasgow Haskell Compiler, GHC Core.

The operational semantics for this language is a small-step semantics based on Sestoft's Mark 1 abstract machine [Sestoft 1997], extended to handle type abstraction and application. A machine state is given by a triple $\langle H, M, S \rangle$ consisting of a heap H that binds term variables to terms, the current term to be evaluated M, and a stack S consisting of tokens that describe the context in which the result of evaluating M is to be used. There are four kinds of stack tokens: variable updates $\#x$ that signal when a variable is to be updated with the result of the computation; applications x that signify when the result should be given a variable as an argument; type applications $[A]$ that signify when the result should be given a *type* argument; and finally, alternatives *alts* that signify that the result should be pattern-matched and branched on.

A term is considered evaluated when it is in *value form*, that is, when it is the empty list, a cons cell or either kind of abstraction. Note that this is quite a restrictive form for lists, as cons cells only contain variables rather than terms; this reflects the fact that a fully-evaluated list will exist almost entirely on the heap. By convention, we denote value terms with letters such as V and W. The complete set of transition rules for the semantics are given in Figure 2. We assume that all bound variables are unique in the statement of these rules, which can be achieved by α-renaming to a fresh variable whenever an abstraction is opened.

Now we can define our notions of improvement and cost-equivalence. If for all heaps and stacks $\langle H, S \rangle$ we have that if $\langle H, M, S \rangle$ terminates after making n Lookup steps then $\langle H, N, S \rangle$ terminates after n or fewer Lookup steps, we say M is *improved* by N, written $M \underset{\sim}{\triangleright} N$. If M and N both improve each other we say that they are *cost-equivalent*, written $M \underset{\sim}{\triangleleft\triangleright} N$. Note that improvements capture the notion of *never-worse*, and do not guarantee that the resulting term is in practice *better*.

We only count Lookups as these are an appropriate measure of the total cost of evaluating a term. In particular, the total number of steps is bounded by a linear function of the number of Lookups [Moran and Sands 1999a]. Of course, it may be the case that an improvement that holds when counting Lookups fails when taking some other operational aspects into account. However, restricting the scope of the costs makes the theory more tractable. This is standard in improvement theory – no abstract model will perfectly match reality, and given that memory access often dominates, counting memory accesses is a reasonable approach.

3.2 Machine-Closure for Relations

Now that we have our language, we must develop a notion of relations between terms of that language. These relations will be between terms of particular types in the language; if R relates terms of type A to terms of type B, we say that R has the type $A \leftrightarrow B$. However, not all relations will have the desired properties of respecting costs. We must therefore have some notion of *permissible* relations that are all guaranteed to have this property.

As noted above, we cannot use Pitts' notion of $\top\top$-closure as it identifies all equivalent terms regardless of cost. However, we can modify $\top\top$-closure to produce another notion of closure that is more suitable for our purposes. We obtain this notion by replacing Pitts' stacks by *heap and*

$$\frac{}{\Gamma, x : A \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \textsc{Abs} \qquad \frac{\Gamma, x : A \vdash M : A \to B}{\Gamma, x : A \vdash M\,x : B}\ \textsc{App}$$

$$\frac{\Gamma, \alpha \vdash M : A}{\Gamma \vdash \Lambda\alpha.M : \forall\,\alpha.A}\ \textsc{TAbs} \qquad \frac{\Gamma \vdash M : \forall\,\alpha.B \qquad \mathit{ftv}\,(A) \subseteq \Gamma}{\Gamma \vdash M\,A : B\,[A/\alpha]}\ \textsc{TApp}$$

$$\frac{}{\Gamma \vdash \mathit{Nil} : \mathit{List}\ A}\ \textsc{Nil} \qquad \frac{}{\Gamma, x : A, \mathit{xs} : \mathit{List}\ A \vdash x :: \mathit{xs} : \mathit{List}\ A}\ \textsc{Cons}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \mathit{List}\ A \qquad \Gamma \vdash M_1 : B \\ \Gamma, x : A, \mathit{xs} : \mathit{List}\ A \vdash M_2 : B\end{array}}{\Gamma \vdash \textbf{case}\ M\ \textbf{of}\ \{\mathit{Nil} \to M_1; x :: \mathit{xs} \to M_2\} : B}\ \textsc{Case}$$

$$\frac{\begin{array}{c}\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash M_1 : T_1 \\ \vdots \\ \Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash M_n : T_n \\ \Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash M : T\end{array}}{\Gamma \vdash \textbf{let}\ \{x_1 = M_1; \ldots; x_n = M_n\}\ \textbf{in}\ M : T}\ \textsc{LetRec}$$

Fig. 1. The typing rules for our language

$$\begin{array}{lll}
\langle H\ \{x = M\},\ x,\ S\rangle & \to \langle H,\ M,\ \#x : S\rangle & \{\,\textsc{Lookup}\,\} \\
\langle H,\ V,\ \#x : S\rangle & \to \langle H\ \{x = V\},\ V,\ S\rangle & \{\,\textsc{Update}\,\} \\
\langle H,\ M\,x,\ S\rangle & \to \langle H,\ M,\ x : S\rangle & \{\,\textsc{Unwind}\,\} \\
\langle H,\ \lambda x.M,\ y : S\rangle & \to \langle H,\ M\,[y/x],\ S\rangle & \{\,\textsc{Subst}\,\} \\
\langle H,\ M\,A,\ S\rangle & \to \langle H,\ M,\ [A] : S\rangle & \{\,\textsc{TypeUnwind}\,\} \\
\langle H,\ \Lambda\alpha.M,\ [A] : S\rangle & \to \langle H,\ M\,[A/\alpha],\ S\rangle & \{\,\textsc{TypeSubst}\,\} \\
\langle H,\ \textbf{case}\ M\ \textbf{of}\ \mathit{alts},\ S\rangle & \to \langle H,\ M,\ \mathit{alts} : S\rangle & \{\,\textsc{Case}\,\} \\
\langle H,\ \mathit{Nil},\ \{\mathit{Nil} \to N_1; x :: \mathit{xs} \to N_2\} : S\rangle & \to \langle H,\ N_1,\ S\rangle & \{\,\textsc{BranchNil}\,\} \\
\langle H,\ y :: \mathit{ys},\ \{\mathit{Nil} \to N_1; x :: \mathit{xs} \to N_2\} : S\rangle & \to \langle H,\ N_2\,[y/x,\ \mathit{ys}/\mathit{xs}],\ S\rangle & \{\,\textsc{BranchCons}\,\} \\
\langle H,\ \textbf{let}\ \{\vec{x} = \vec{M}\}\ \textbf{in}\ N,\ S\rangle & \to \langle H\ \{\vec{x} = \vec{M}\},\ N,\ S\rangle & \{\,\textsc{Letrec}\,\}
\end{array}$$

Fig. 2. The call-by-need abstract machine

stack pairs that capture all of the state of the abstract machine besides the term itself, and adapting the $\top$ relation to take costs into account. We also follow Voigtländer and Johann [2007] and use one-directional implication in our definition, as this is more useful for reasoning about program orderings. We call the resulting notion of closure *machine-closure*. Given a relation $R : A \leftrightarrow B$, two terms $M_1 : A$ and $M_2 : B$ are related by the machine-closure of R, written $R^{\mathcal{M}}$, if:

for all heap and stack pairs $\langle H_1, S_1\rangle$ and $\langle H_2, S_2\rangle$,
  if for all $(N_1, N_2) \in R$, $\langle H_1, N_1, S_1\rangle \downarrow_n \Rightarrow \langle H_2, N_2, S_2\rangle \downarrow_n$
    then $\langle H_1, M_1, S_1\rangle \downarrow_n \Rightarrow \langle H_2, M_2, S_2\rangle \downarrow_n$

By $\langle H,\ N,\ S\rangle \downarrow_n$, we mean that if we start the abstract machine in state $\langle H,\ N,\ S\rangle$, then it will take at most n Lookup steps to finish evaluating. The key idea is that machine-closed relations should only be able to capture properties that can be observed by running terms in specific machine contexts and comparing the costs. In other words, they must be *observational improvement* properties.

We can prove a number of useful theorems about machine-closure. Firstly, we note that machine-closure is monotone, i.e. if $R \subseteq S$ then $R^{\mathcal{M}} \subseteq S^{\mathcal{M}}$. This follows from the fact that R appears twice under the left-hand side of an implication. Secondly, we show that machine-closure really *is* a notion of closure, i.e. $R \subseteq R^{\mathcal{M}} = (R^{\mathcal{M}})^{\mathcal{M}}$. That R is smaller than $R^{\mathcal{M}}$ follows from the definition, as when $(M_1,\ M_2) \in R$ then the criterion for membership in $R^{\mathcal{M}}$ becomes a tautology. Then, as any heap and stack pairs that identify pairs in R will also identify pairs in $R^{\mathcal{M}}$ by definition, we can conclude that any pair in $(R^{\mathcal{M}})^{\mathcal{M}}$ will also be in $R^{\mathcal{M}}$, so our closure operation is idempotent.

Finally, we show that machine-closed relations respect improvement, in the sense that whenever $(M_1,\ M_2) \in R$, $M_1' \mathrel{\underset{\sim}{\triangleright}} M_1$ and $M_2 \mathrel{\underset{\sim}{\triangleright}} M_2'$, then $(M_1',\ M_2') \in R$. Assuming this precondition, and assuming R is machine closed, we can then conclude that for any heap and stack pairs $\langle H_1,\ S_1\rangle$ and $\langle H_2,\ S_2\rangle$ such that for any $(N_1,\ N_2) \in R$, we have $\langle H_1,\ N_1,\ S_1\rangle \downarrow_n \Rightarrow \langle H_2,\ N_2,\ S_2\rangle \downarrow_n$:

$$\begin{array}{ll}
 & \langle H_1,\ M_1',\ S_1\rangle \downarrow_n \\
\Rightarrow & \{\,\text{improvement}\,\} \\
 & \langle H_1,\ M_1,\ S_1\rangle \downarrow_n \\
\Rightarrow & \{\,\text{assumption}\,\} \\
 & \langle H_2,\ M_2,\ S_2\rangle \downarrow_n \\
\Rightarrow & \{\,\text{improvement}\,\} \\
 & \langle H_2,\ M_2',\ S_2\rangle \downarrow_n
\end{array}$$

Therefore, we can conclude that $(M_1',\ M_2')$ is also in $R^{\mathcal{M}}$.

3.3 Actions on Relations

Now that we have defined an appropriate notion of relation on terms, we must in turn define how the type constructors of our language act on those relations. In particular, we need these actions to preserve the machine-closure property, in order that any relation built out of machine-closed relations using these constructors will also be machine-closed. We can define relational actions for the three type constructors of our language as follows:

- *Function spaces.* Given relations, $R\ :\ T_1 \leftrightarrow T_1'$ and $S\ :\ T_2 \leftrightarrow T_2'$, we can define the relation $R \rightarrow S\ :\ (T_1 \rightarrow T_2) \leftrightarrow (T_1' \rightarrow T_2')$. Two terms $M\ :\ T_1 \rightarrow T_2$ and $M'\ :\ T_1' \rightarrow T_2'$ are related by $R \rightarrow S$ iff for all collections of bindings B, B', we have the following implication:

  $$\begin{array}{l}
  (\mathbf{let}\ B\ \mathbf{in}\ y,\ \mathbf{let}\ B'\ \mathbf{in}\ y) \in R \\
  \quad \Rightarrow\ (\mathbf{let}\ B\ \mathbf{in}\ M\ y,\ \mathbf{let}\ B'\ \mathbf{in}\ M'\ y) \in S
  \end{array}$$

  In other words, they must take related arguments to related results.
- *List types.* Given $R\ :\ T \leftrightarrow T'$, we can define the relation $List\ R\ :\ List\ T \leftrightarrow List\ T'$ as the least fixed point of the following equation, for all collections of bindings B, B':

  $$\begin{array}{l}
  List\ R = (\{(Nil,\ Nil)\} \cup \\
  \quad \{\ (\mathbf{let}\ B\ \mathbf{in}\ y :: ys,\ \mathbf{let}\ B'\ \mathbf{in}\ y :: ys) \\
  \quad \mid (\mathbf{let}\ B\ \mathbf{in}\ y,\ \mathbf{let}\ B'\ \mathbf{in}\ y) \in R, \\
  \quad\ \ (\mathbf{let}\ B\ \mathbf{in}\ ys,\ \mathbf{let}\ B'\ \mathbf{in}\ ys) \in List\ R\})^{\mathcal{M}}
  \end{array}$$

In other words, *Nil* is related to itself, and non-empty lists are related if their heads and tails are related. The fixed point is guaranteed to exist because all operations in the definition are monotone and relations $List\ A \leftrightarrow List\ A'$ form a lattice.

- *Type abstraction.* Given a family of functions R indexed over types T_1, T_1' that map relations of type $T_1 \leftrightarrow T_1'$ to relations of type $T_2\ [T_1] \leftrightarrow T_2'\ [T_1']$, we can define the relation $\forall\ r.\ \ R\ (r)$. Two terms $M\ :\ \forall\ \alpha.T_2\ [\alpha]$ and $M'\ :\ \forall\ \alpha.T_2'\ [\alpha]$ are related by $\forall\ r.\ \ R\ (r)$ if and only if:

$$\begin{array}{l}\forall\ T_1,\ T_1' \in \mathit{Type},\ S\ :\ T_1 \leftrightarrow T_1'\ \ . \\ \quad (M\ T_1,\ M'\ T_1') \in R\ (S)\end{array}$$

  In other words, two polymorphic terms are related if a relation between two types can be lifted to a relation that relates the terms' instantiations at those types.

These three definitions are based on the relational actions from Pitts [2000], adapted to our setting. We use the technique of adding bindings to get around the limitation of applying terms and constructors only to variables, and to take account of sharing. There is an implicit requirement in the above definitions that all the bindings are type-correct. Note that our approach for lists differs from that of Pitts, who uses a *greatest* fixed point to make the use of coinduction easier. In contrast, we use a least fixed point to make the use of induction easier. In Pitts' setting, both definitions result in the same logical relation, and we conjecture the same is true in our setting. However, our theory does not depend on this being the case, and all of our results hold regardless.

The next step is to prove that all three of these constructions preserve the notion of machine-closure. That this is the case for the *List* construction is immediate, as any fixed-point of the defining equation is by definition machine-closed. For the other two, the proof is a little more involved:

Lemma 3.1 (Function Space Preserves Machine-Closure).

(i) Given bindings B, B' and heap and stack pairs $\langle H,\ S\rangle$, $\langle H',\ S'\rangle$, if we have

(a) $(\mathbf{let}\ B\ \mathbf{in}\ y,\ \mathbf{let}\ B'\ \mathbf{in}\ y) \in R$
(b) $\forall\ (N,\ N') \in R'.\ \langle H,\ N,\ S\rangle \downarrow_n \Rightarrow \langle H',\ N',\ S'\rangle \downarrow_n$

then we also have

$$\begin{array}{l}\forall\ (M,\ M') \in R \rightarrow R'\ \ . \\ \quad \langle H + B,\ M,\ y\ :\ S\rangle \downarrow_n \Rightarrow \langle H' + B',\ M',\ y\ :\ S'\rangle \downarrow_n\end{array}$$

(By $H + B$, we mean the heap gained by adding the bindings B to the heap H.)

(ii) Given machine-closed relations R and R', then $R \rightarrow R'$ is also machine-closed.

Proof. For part (i), take arbitrary $(M,\ M') \in R \rightarrow R'$. We note that $(\mathbf{let}\ B\ \mathbf{in}\ M\ y,\ \mathbf{let}\ B'\ \mathbf{in}\ M'\ y) \in R'$ by assumption (a) and the definition of $R \rightarrow R'$, and reason as follows:

$$\begin{array}{ll} & \langle H + B,\ M,\ y\ :\ S\rangle \downarrow_n \\ \Leftrightarrow & \{\ \text{Unwind and Letrec steps are free}\ \} \\ & \langle H,\ \mathbf{let}\ B\ \mathbf{in}\ M\ y,\ S\rangle \downarrow_n \\ \Rightarrow & \{\ \text{assumption (b)}\ \} \\ & \langle H',\ \mathbf{let}\ B'\ \mathbf{in}\ M'\ y,\ S'\rangle \downarrow_n \\ \Leftrightarrow & \{\ \text{Unwind and Letrec steps are free}\ \} \\ & \langle H' + B',\ M',\ y\ :\ S'\rangle \downarrow_n \end{array}$$

For part (ii), we assume $(M,\ M') \in (R \rightarrow R')^{\ \mathcal{M}}$ for machine-closed R and R', and aim to prove $(M,\ M') \in R \rightarrow R'$. Assuming type-correct bindings B and B' such that $(\mathbf{let}\ B\ \mathbf{in}\ y,\ \mathbf{let}\ B'\ \mathbf{in}\ y) \in R$, we must show that ($\mathbf{let}\ B\ \mathbf{in}\ M\ y$, $\mathbf{let}\ B'\ \mathbf{in}\ M'\ y$) $\in R'$. Letting $\langle H, S\rangle$, $\langle H', S'\rangle$ be arbitrary heap-stack pairs such that $\forall\ (N, N') \in R'.\ \langle H, N, S\rangle \downarrow_n \Rightarrow \langle H', N', S'\rangle \downarrow_n$, we have:

$$\begin{array}{ll} & \langle H, \mathbf{let}\ B\ \mathbf{in}\ M\ y, S\rangle \downarrow_n \\ \Leftrightarrow & \{\ \textsc{Unwind}\text{ and }\textsc{Letrec}\text{ steps are free}\ \} \\ & \langle H + B, M, y : S\rangle \downarrow_n \\ \Rightarrow & \{\ \text{part (i), definition of machine-closure}\ \} \\ & \langle H' + B', M', y : S\rangle \downarrow_n \\ \Leftrightarrow & \{\ \textsc{Unwind}\text{ and }\textsc{Letrec}\text{ steps are free}\ \} \\ & \langle H', \mathbf{let}\ B'\ \mathbf{in}\ M'\ y, S\rangle \downarrow_n \end{array}$$

Because these heap-stack pairs were arbitrary, we can conclude ($\mathbf{let}\ B\ \mathbf{in}\ M\ y$, $\mathbf{let}\ B'\ \mathbf{in}\ M'\ y$) $\in R'^{\mathcal{M}}$, which implies the desired result by the assumption that R' is machine-closed. □

Lemma 3.2 (Type Abstraction Preserves Machine-Closure).

(i) Given heap and stack pairs $\langle H, S\rangle$, $\langle H', S'\rangle$, if we have

$$\begin{array}{l} \forall\ r : T \leftrightarrow T', (N, N') \in R\ (r)\ . \\ \quad \langle H, N, S\rangle \downarrow_n \Rightarrow \langle H', N', S'\rangle \downarrow_n \end{array}$$

then we also have

$$\begin{array}{l} \forall\ (M, M') \in \forall\ r.\ R\ (r)\ . \\ \quad \langle H, M, [T] : S\rangle \downarrow_n \Rightarrow \langle H, M', [T'] : S'\rangle \downarrow_n \end{array}$$

(ii) For any types T_2, T_2', given a family of functions R indexed over types T_1, T_1' that map relations of type $T_1 \leftrightarrow T_1'$ to relations of type $T_2\ [T_1] \leftrightarrow T_2'\ [T_1']$, if all the relations $R\ (r)$ are machine-closed then the relation $\forall\ r.R\ (r)$ will also be machine-closed.

Proof. Similarly to Lemma 3.1, but with TypeUnwind steps rather than Unwind/Letrec. □

3.4 The Logical Relation

Given the above relational actions, it is possible to define a family of relations indexed over types, abstracted over the interpretations of type variables. By convention this family is called the *logical relation*, denoted Δ. The idea is that this family of relations will relate every term to itself, and so by calculating the relation for a particular type we will get a property of terms of that type. This property is called the *abstraction theorem*. Using the constructions from the previous section, we can define our logical relation recursively on the structure of types:

$$\begin{array}{lcl} \Delta\ (\vec{R}/\vec{\alpha})\ (\alpha_i) & = & R_i \\ \Delta\ (\vec{R}/\vec{\alpha})\ (T_1 \to T_2) & = & \Delta\ (\vec{R}/\vec{\alpha})\ (T_1) \to \Delta\ (\vec{R}/\vec{\alpha})\ (T_2) \\ \Delta\ (\vec{R}/\vec{\alpha})\ (\forall\ \alpha.T) & = & \forall\ r.\ \Delta\ (r^{\mathcal{M}}/\alpha, \vec{R}/\vec{\alpha})\ (T) \\ \Delta\ (\vec{R}/\vec{\alpha})\ (List\ T) & = & List\ (\Delta\ (\vec{R}/\vec{\alpha})\ (T)) \end{array}$$

A simple proof by induction shows that $\Delta\ (\vec{R}/\vec{\alpha})\ (T)$ is machine-closed whenever all the $\vec{R}$ are. Furthermore, another proof by induction shows that if all of $\vec{R}$ are subsets of $\underset{\sim}{\triangleright}$ then $\Delta\ (\vec{R}/\vec{\alpha})\ (T)$ is also a subset of $\underset{\sim}{\triangleright}$. This corresponds to Reynolds' *identity extension lemma* [Reynolds 1983].

3.5 The Abstraction Theorem (for Non-Recursive Programs)

Now we can prove a version of the abstraction theorem. In this section we will prove the theorem for a non-recursive version of our language, but in the next section we will show how to extend the theorem to deal with recursive programs. We proceed in this manner because dealing with recursion requires applying some limitations to our language that we do not wish to apply to non-recursive programs. For the purposes of this section, we use the following weaker version of the Let rule that disallows the use of recursion:

$$\dfrac{\Gamma \vdash M_1 : T_1 \qquad \ldots \qquad \Gamma \vdash M_n : T_n \\ \Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash M : T}{\Gamma \vdash \textbf{let}\ \{x_1 = M_1; \ldots; x_n = M_n\}\ \textbf{in}\ M : T}\ \textsc{Let'}$$

Now we state the abstraction theorem:

Theorem 3.3 (Abstraction for Non-Recursive Programs). *Given a closed term M and closed type A such that* $\vdash M : A$ *(using the weaker* Let' *rule rather than* Let*), we have* $(M, M) \in \Delta\ ()\ (A)$.

To prove this theorem we must actually prove a stronger theorem, extending the logical relation Δ to terms with free variables. Assuming $\Gamma = \vec{\alpha}, x_1 : T_1, \ldots, x_n : T_n$, we define $\Gamma \vdash M\ \Delta\ M' : T$ to mean that for all bindings B, B' that close the terms M, M' respectively, and machine-closed relations $\vec{R} : T_i \leftrightarrow T_i'$ (where *length* $\vec{R}$ = *length* $\vec{\alpha}$), we have:

$$\begin{array}{l} \quad (\forall\ i \in [1\,..\,n]\ . \\ \quad\quad (\textbf{let}\ B\ \textbf{in}\ x_i, \\ \quad\quad\quad \textbf{let}\ B'\ \textbf{in}\ x_i) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_i)) \\ \Rightarrow \\ \quad (\textbf{let}\ B\ \textbf{in}\ M\ [\vec{T}/\vec{\alpha}], \\ \quad\quad \textbf{let}\ B'\ \textbf{in}\ M'\ [\vec{T}'/\vec{\alpha}]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T) \end{array}$$

Theorem 3.3 then follows immediately from the following lemma:

Lemma 3.4. *Given a context* Γ*, term M and type A such that* $\Gamma \vdash M : T$ *(using the weaker* Let' *rule instead of* Let*), we have that* $\Gamma \vdash M\ \Delta\ M : T$. (The proof is given in the Appendix.)

4 DEALING WITH RECURSIVE BINDINGS

In order to extend our treatment to deal with the full Let rule, we must have some way to reason about the behaviour of recursive bindings. The usual technique, as used by Pitts [2000] and Moran and Sands [1999a], is to define some notion of unwinding and to consider a fixed point as a limit to the sequence of increasingly deep unwindings. This notion of limit is made precise by an *unwinding lemma* that relates the behaviour of recursive terms to that of their finite non-recursive unwindings. For example, the meaning of **let** $x = M$ **in** N is generally taken to be the limit of the sequence:

$$\textbf{let}\ x_0 = \bot\ \textbf{in}\ N\ [x_0/x]$$

$$\textbf{let}\ x_1 = M\ [x_0/x]; x_0 = \bot\ \textbf{in}\ N\ [x_1/x]$$

$$\textbf{let}\ x_2 = M\ [x_1/x]; x_1 = M\ [x_0/x]; x_0 = \bot\ \textbf{in}\ N\ [x_2/x]$$

$$\vdots$$

We denote the nth element of this sequence as $\textbf{let}\ x =^n M\ \textbf{in}\ N$

This technique cannot be applied as-is in a call-by-need setting, as there is work shared in the recursive definition that is not shared in any of the unwindings. We follow the approach taken by Moran and Sands [1999a] and restrict ourselves to cases where the right-hand side is a value, in which case the problem does not arise. We return to this assumption in the concluding section.

First of all, we must state and prove our version of the unwinding lemma. In this case, we want to know that machine-closed relations behave well with respect to limits of unwindings, which can be regarded as a kind of continuity property of relations.

Lemma 4.1 (Unwinding).

(i) A machine state $\langle H\ \{x = V\},\ N,\ S\rangle$ terminates in k steps iff there is some non-negative integer n such that the machine state $\langle H\ \{x =^n V\},\ N,\ S\rangle$ terminates in k steps.

(ii) Given a pair of closed terms $\textbf{let}\ x = V\ \textbf{in}\ N$ and $\textbf{let}\ x = V'\ \textbf{in}\ N'$ and a machine-closed relation R, the membership relation $(\textbf{let}\ x = V\ \textbf{in}\ N,\ \textbf{let}\ x = V'\ \textbf{in}\ N') \in R$ holds iff the membership relation $(\textbf{let}\ x =^n V\ \textbf{in}\ N,\ \textbf{let}\ x =^n V'\ \textbf{in}\ N') \in R$ holds for all non-negative n.

Proof. Part (i) follows easily from the fact that a binding can only recurse finitely many times in a given execution. For part (ii), the $\Rightarrow$ direction follows from (i) and the definition of machine-closure. For the $\Leftarrow$ direction, suppose we have heap and stack pairs $\langle H_1, S_1\rangle$ and $\langle H_2, S_2\rangle$ such that for any $(P, P') \in R$, we have $\langle H_1, P, S_1\rangle \downarrow_n \Rightarrow \langle H_2, P', S_2\rangle \downarrow_n$. We prove that $\langle H_1, \textbf{let}\ x = V\ \textbf{in}\ N, S_1\rangle \downarrow_n \Rightarrow \langle H_2, \textbf{let}\ x = V'\ \textbf{in}\ N', S_2\rangle \downarrow_n$ by case analysis on the termination behaviour of the left-hand side of this result, considering the two cases in turn:

- *Termination.* If $\langle H_1, \textbf{let}\ x = V\ \textbf{in}\ N, S_1\rangle$ terminates in k steps, then by part (i) there must be some non-negative integer n such that $\langle H_1, \textbf{let}\ x =^n V\ \textbf{in}\ N, S_1\rangle$ terminates in k steps, in which case we also have that $\langle H_2, \textbf{let}\ x =^n V'\ \textbf{in}\ N', S_2\rangle$ terminates in k steps, and then by part (i) we can conclude that $\langle H_2, \textbf{let}\ x = V'\ \textbf{in}\ N', S_2\rangle$ terminates in k steps.
- *Non-termination.* If $\langle H_1, \textbf{let}\ x = V\ \textbf{in}\ N, S_1\rangle$ does not terminate, $\langle H_1, \textbf{let}\ x = V\ \textbf{in}\ N, S_1\rangle \downarrow_n$ $\Rightarrow \langle H_2, \textbf{let}\ x = V'\ \textbf{in}\ N', S_2\rangle \downarrow_n$ is vacuously true. □

Now we can extend our proof of Lemma 3.4 to deal with recursive bindings. Only the Let case differs. For simplicity, we illustrate the case for a single recursive binding, but the result can be generalised easily to the case of multiple recursive bindings.

Lemma 4.2 (Recursive bindings). *If $\Gamma, x : T_1 \vdash V \,\Delta\, V : T_1$ and $\Gamma, x : T_1 \vdash N \,\Delta\, N : T_2$ both hold, then $\Gamma \vdash \textbf{let}\ x = V\ \textbf{in}\ N \,\Delta\, \textbf{let}\ x = V\ \textbf{in}\ N : T_2$ will also hold.*

Proof. Let B and B' be bindings that close $\textbf{let}\ x = M\ \textbf{in}\ N$ (but don't contain x; we can ensure this with alpha-renaming) and let $\vec{R} : \vec{T} \leftrightarrow \vec{T}'$ be a list of machine-closed relations with length equal to the number of free type variables in Γ. Assume that for any $y : A \in \Gamma$, we have $(\textbf{let}\ B\ \textbf{in}\ y, \textbf{let}\ B'\ \textbf{in}\ y') \in \Delta\ (\vec{R}/\vec{\alpha})\ (A)$. We aim to prove $(\textbf{let}\ B\ \textbf{in}\ \textbf{let}\ x = V\ [\vec{T}/\alpha]\ \textbf{in}\ N\ [\vec{T}/\alpha], \textbf{let}\ B'\ \textbf{in}\ \textbf{let}\ x = V\ [\vec{T}'/\alpha]\ \textbf{in}\ N\ [\vec{T}'/\alpha]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_2)$.

It suffices to prove $(\textbf{let}\ B; x = V\ [\vec{T}/\alpha]\ \textbf{in}\ x, \textbf{let}\ B'; x = V\ [\vec{T}'/\alpha]\ \textbf{in}\ x) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_1)$, which implies our goal by the assumption $\Gamma, x : T_1 \vdash N \,\Delta\, N : T_2$. First, we prove that for all n, we have $(\textbf{let}\ B; x =^n V\ [\vec{T}/\alpha]\ \textbf{in}\ x_n, \textbf{let}\ B'; x =^n V\ [\vec{T}'/\alpha]\ \textbf{in}\ x_n) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_1)$. We do this by induction on n. The base case is simple, as both sides diverge and all machine-closed relations relate diverging terms to each other. For the inductive case, we reason as follows. Our induction hypothesis is that $(\textbf{let}\ B; x =^n V\ [\vec{T}/\alpha]\ \textbf{in}\ x_n, \textbf{let}\ B'; x =^n V\ [\vec{T}'/\alpha]\ \textbf{in}\ x_n) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_1)$, so by the assumption $\Gamma, x : T_1 \vdash V \,\Delta\, V : T_1$ we know that $(\textbf{let}\ B; x =^n V\ [\vec{T}/\alpha]\ \textbf{in}\ V\ [x_n/x], \textbf{let}\ B'; x =^n$

$V\ [\vec{T}/\alpha]$ **in** $V\ [x_n/x]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_1)$. But these terms are cost-equivalent to (**let** $B; x =^{n+1} V\ [\vec{T}/\alpha]$ **in** x_n, **let** $B'; x =^{n+1} V\ [\vec{T}'/\alpha]$ **in** x_n), so we are done.

Finally, we apply Lemma 4.1 to give (**let** $B; x = V\ [\vec{T}/\alpha]$ **in** x, **let** $B'; x = V\ [\vec{T}'/\alpha]$ **in** $x) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_1)$, which completes the proof of this lemma. □

5 APPLICATIONS

In this section, we apply our abstraction theorem to a range of examples to derive *free theorems for improvement*: theorems about the operational time efficiency of terms that rely only on their polymorphic types. As this theory is built on the same language and cost model as Moran and Sands [1999a], all the same theorems hold, but now we have the added tool of parametricity. In order to apply this tool, however, we need ways to construct interesting machine-closed relations.

In Moran and Sands' theory, there is the special operator ✓ (which is pronounced as 'tick') that adds one unit of cost. In our context, it is useful to reify this into the machine model, as this makes reasoning easier. We therefore add a tick token ✓ to our stacks: it is safe to do this because our reasoning was generic in the possible stack items. We introduce the following transition rule for ✓ to the abstract machine semantics that we defined in Figure 2:

$$\langle H,\ M,\ \checkmark\ :\ S\rangle \rightarrow \langle H,\ M,\ S\rangle \quad \{\textsc{Tick}\}$$

and count these steps as well in our cost semantics, so the total cost of evaluation is now given by the number of Lookup and Tick steps. We can then formulate the following theorem:

Theorem 5.1. *Given a context $\mathbb{C}$ such that $\mathbb{C}\ [\bot] \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \bot$, the following relations are machine-closed:*

(i) $\{(M,\ M') \mid \mathbb{C}\ [M] \mathrel{\underset{\sim}{\triangleright}} M'\}$

(ii) $\{(M,\ M') \mid M \mathrel{\underset{\sim}{\triangleright}} \mathbb{C}\ [M']\}$

By $\bot$, we mean some arbitrarily-chosen divergent term.

Proof. We take an arbitrary pair $(M,\ M')$ in the machine-closure of the relation and show it is in the original relation. For (i) we must show $\mathbb{C}\ [M] \mathrel{\underset{\sim}{\triangleright}} M'$, which by definition is equivalent to:

$$\forall \langle H,\ S\rangle,\ \langle H,\ \mathbb{C}\ [M],\ S\rangle \downarrow_n \Rightarrow \langle H,\ M',\ S\rangle \downarrow_n$$

Take an arbitrary $\langle H,\ S\rangle$. Starting in the state $\langle H,\ \mathbb{C}\ [x],\ S\rangle$ where x is fresh, the machine will either evaluate to $\langle H',\ x,\ S'\rangle$ in k steps, or it will diverge, because otherwise it would violate the assumption $\mathbb{C}\ [\bot] \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \bot$. If $\langle H,\ \mathbb{C}\ [x],\ S\rangle$ diverges then so does $\langle H,\ \mathbb{C}\ [M],\ S\rangle$, so $\langle H,\ \mathbb{C}\ [M],\ S\rangle \downarrow_n \Rightarrow \langle H,\ M',\ S\rangle \downarrow_n$ is vacuously true and we are done. Otherwise, take an arbitrary $(N,\ N')$ such that $\mathbb{C}\ [N] \mathrel{\underset{\sim}{\triangleright}} N'$. We know that $\langle H,\ \mathbb{C}\ [N],\ S\rangle$ evaluates to $\langle H',\ N,\ S'\rangle$ in k steps by construction, so we can conclude $\langle H',\ N,\ {}^{k}\checkmark\ :\ S'\rangle \downarrow_n \Rightarrow \langle H,\ \mathbb{C}\ [N],\ S\rangle \downarrow_n$ (where ${}^{k}\checkmark$ represents k copies of ✓). From this and the assumption $\mathbb{C}\ [N] \mathrel{\underset{\sim}{\triangleright}} N'$ we can conclude $\langle H',\ N,\ {}^{k}\checkmark\ :\ S'\rangle \downarrow_n \Rightarrow \langle H,\ N',\ S\rangle \downarrow_n$. Because N and N' were arbitrary, by the definition of machine-closure this implies $\langle H',\ M,\ {}^{k}\checkmark\ :\ S'\rangle \downarrow_n \Rightarrow \langle H,\ M',\ S\rangle \downarrow_n$, which by construction implies $\langle H,\ \mathbb{C}\ [M],\ S\rangle \downarrow_n \Rightarrow \langle H,\ M',\ S\rangle \downarrow_n$.

The second case (ii) follows the same pattern. □

5.1 Example: Fixed Point Fusion

Given a fixed point combinator $\mathit{fix}\ :\ \forall\ \alpha.\ (\alpha \rightarrow \alpha) \rightarrow \alpha$, the *fixed point fusion* rule [Meijer et al. 1991] states that if h is a strict function such that $h \cdot f = g \cdot h$, then we have $\mathit{fix}\ f = h\ (\mathit{fix}\ g)$. We can actually prove an improving version of this rule using parametricity, solely from the type. Given a term $\mathit{fix}\ :\ \forall\ \alpha.\ (\alpha \rightarrow \alpha) \rightarrow \alpha$, the abstraction theorem implies the following:

```
for any bindings $B_2, B_2'$ and machine-closed relation $R : T \Leftrightarrow T'$,
  if for any bindings $B_1, B_1'$ we have
    $(\textbf{let } B_1 \textbf{ in } x, \textbf{let } B_1' \textbf{ in } x) \in R \Rightarrow$
      $(\textbf{let } B_1 \textbf{ in } (\textbf{let } B_2 \textbf{ in } f)\ x, \textbf{let } B_1' \textbf{ in } (\textbf{let } B_2' \textbf{ in } f)\ x) \in R$
  then $(\textbf{let } B_2 \textbf{ in } \mathit{fix}\ T\ f, \textbf{let } B_2' \textbf{ in } \mathit{fix}\ T'\ f) \in R$
```

Simplifying, by letting $B_2 = \{f = M_1\}$, $B_2' = \{f = M_2\}$, we obtain:

```
for any machine-closed relation $R$,
  if for any bindings $B_1, B_1'$ we have
    $(\textbf{let } B_1 \textbf{ in } x, \textbf{let } B_1' \textbf{ in } x) \in R \Rightarrow (\textbf{let } B_1 \textbf{ in } M_1\ x, \textbf{let } B_1' \textbf{ in } M_2\ x) \in R$
  then $(\textbf{let } f = M_1 \textbf{ in } \mathit{fix}\ T\ f, \textbf{let } g = M_2 \textbf{ in } \mathit{fix}\ T'\ g) \in R$
```

Now if we let $R = \{(N, N') \mid \mathbb{C}[N] \mathrel{\underset{\sim}{\triangleright}} N'\}$, we obtain:

```
for any context $\mathbb{C}$ such that $\mathbb{C}[\bot] \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \bot$,
  if for any bindings $B_1, B_1'$ we have
    $\textbf{let } B_1 \textbf{ in } \mathbb{C}[x] \mathrel{\underset{\sim}{\triangleright}} \textbf{let } B_1' \textbf{ in } x \Rightarrow \textbf{let } B_1 \textbf{ in } \mathbb{C}[M_1\ x] \mathrel{\underset{\sim}{\triangleright}} \textbf{let } B_1' \textbf{ in } M_2\ x$
  then $\textbf{let } f = M_1 \textbf{ in } \mathbb{C}[\mathit{fix}\ T\ f] \mathrel{\underset{\sim}{\triangleright}} \textbf{let } g = M_2 \textbf{ in } \mathit{fix}\ T'\ g$
```

Finally, we observe that the precondition $\textbf{let } B_1 \textbf{ in } \mathbb{C}[x] \mathrel{\underset{\sim}{\triangleright}} \textbf{let } B_1' \textbf{ in } x \Rightarrow \textbf{let } B_1 \textbf{ in } \mathbb{C}[M_1\ x] \mathrel{\underset{\sim}{\triangleright}} \textbf{let } B_1' \textbf{ in } M_2\ x$ is implied by $\textbf{let } B_1 \textbf{ in } \mathbb{C}[M_1\ x] \mathrel{\underset{\sim}{\triangleright}} \textbf{let } B_1, y = \mathbb{C}[x] \textbf{ in } M_2\ y$, so we obtain:

```
for any context $\mathbb{C}$ such that $\mathbb{C}[\bot] \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \bot$,
  if for any bindings $B_1$ we have
    $\textbf{let } B_1 \textbf{ in } \mathbb{C}[M_1\ x] \mathrel{\underset{\sim}{\triangleright}} \textbf{let } B_1, y = \mathbb{C}[x] \textbf{ in } M_2\ y$
  then $\textbf{let } f = M_1 \textbf{ in } \mathbb{C}[\mathit{fix}\ T\ f] \mathrel{\underset{\sim}{\triangleright}} \textbf{let } g = M_2 \textbf{ in } \mathit{fix}\ T'\ g$
```

Here, the role of the function h is played by the context $\mathbb{C}$, and the requirement $\mathbb{C}[\bot] \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \bot$ states that this context must be strict. The fixed functions f and g correspond to the terms M_1 and M_2, and the requirement that $\textbf{let } B_1 \textbf{ in } \mathbb{C}[M_1\ x] \mathrel{\underset{\sim}{\triangleright}} \textbf{let } B_1, y = \mathbb{C}[x] \textbf{ in } M_2\ y$ is simply a spelling-out of $h \cdot f = g \cdot h$. Thus we see that our abstraction theorem implies an improving version of the standard fixed point fusion rule. We can also take $R = \{(N, N') \mid N \mathrel{\underset{\sim}{\triangleright}} \mathbb{C}[N']\}$ to obtain a version of this rule where the improvement goes in the other direction:

```
for any context $\mathbb{C}$ such that $\mathbb{C}[\bot] \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} \bot$,
  if for any bindings $B_1$ we have
    $\textbf{let } B_1, y = \mathbb{C}[x] \textbf{ in } M_1\ y \mathrel{\underset{\sim}{\triangleright}} \textbf{let } B_1 \textbf{ in } \mathbb{C}[M_2\ x]$
  then $\textbf{let } f = M_1 \textbf{ in } \mathit{fix}\ T\ f \mathrel{\underset{\sim}{\triangleright}} \textbf{let } g = M_2 \textbf{ in } \mathbb{C}[\mathit{fix}\ T'\ g]$
```

The first rule tells us that fusion is an improvement when the fusion precondition is an improvement in one direction. The second tells us that reverse fusion ("fission") is an improvement when the fusion precondition is an improvement in the other direction. Together, these imply that if the fusion precondition is a cost equivalence, then fusion itself is a cost equivalence.

5.2 Example: Map Fusion

For our second example, consider the *map* function on lists:

```
$map : \forall\ \alpha\ \beta.\ (\alpha \to \beta) \to List\ \alpha \to List\ \beta$
$map = \Lambda\alpha\ \beta.\lambda f\ xs \to$
  case $xs$ of
    $Nil \qquad\ \to Nil$
    $(y :: ys) \to$ let $y' = f\ y; ys' = map\ \alpha\ \beta\ f\ ys$ in $y' :: ys'$
```

A common optimisation concerning this function is *map fusion*, where two successive applications of *map* are fused into one. The correctness of this transformation is a consequence of the free theorem for *map*, so it makes sense to ask if we can justify it as an improvement in a similar way. The abstraction theorem applied to *map* gives us the following:

$$\begin{array}{l}
\text{for any bindings } B_2, B_2' \text{ and machine-closed relations } R_1 : T_1 \leftrightarrow T_1', R_2 : T_2 \leftrightarrow T_2', \\
\quad \text{if for any bindings } B_1, B_1' \text{ we have} \\
\qquad (\textbf{let } B_1 \textbf{ in } x, \textbf{let } B_1' \textbf{ in } x) \in R_1 \Rightarrow \\
\qquad\quad (\textbf{let } B_1 \textbf{ in } (\textbf{let } B_2 \textbf{ in } f)\ x, \textbf{let } B_1' \textbf{ in } (\textbf{let } B_2' \textbf{ in } f)\ x) \in R_2 \\
\quad \text{and } (\textbf{let } B_2 \textbf{ in } xs, \textbf{let } B_2' \textbf{ in } xs) \in \mathit{List}\ R_1 \\
\quad \text{then } (\textbf{let } B_2 \textbf{ in } \mathit{map}\ T_1\ T_2\ f\ xs, \textbf{let } B_2' \textbf{ in } \mathit{map}\ T_1'\ T_2'\ f\ xs) \in \mathit{List}\ R_2
\end{array}$$

As before, we can use theorem 5.1 to instantiate our relations R_1 and R_2 with contexts $\mathbb{C}$ and $\mathbb{D}$. However, it is not clear that the *List* action applied to a relation based on a context will still be a relation based on a context. To proceed, we need the following result:

Theorem 5.2. *Given a context* $\mathbb{C}$ *such that* $\mathbb{C}[\bot] \underset{\sim}{\lhd\!\rhd} \bot$ *and* $\Gamma \vdash M : T_1 \Rightarrow \Gamma \vdash \mathbb{C}[M] : T_2$,

$$\begin{array}{l}
\quad \mathit{List}\ \{(M_1, M_2) \mid \mathbb{C}[M_1] \underset{\sim}{\rhd} M_2\} \\
\subseteq \\
\quad \{(L1, L2) \mid \textbf{let } \mathit{map} = \ldots; f = \lambda x.\mathbb{C}[x]; l = L1 \textbf{ in } \mathit{map}\ T_1\ T_2\ f\ l \underset{\sim}{\rhd} L2\}
\end{array}$$

Proof. From the definition of *List*, we know that *List* $\{(M_1, M_2)$ C[M1] $\rangle M_2\}$ is the greatest solution to the following equation:

$$\begin{array}{l}
X = (\{(\mathit{Nil}, \mathit{Nil})\} \cup \\
\qquad \{\ (\textbf{let } B \textbf{ in } y :: ys, \textbf{let } B' \textbf{ in } y :: ys) \\
\qquad \mid \textbf{let } B \textbf{ in } \mathbb{C}[y] \underset{\sim}{\rhd} \textbf{let } B' \textbf{ in } y, \\
\qquad\ \ (\textbf{let } B \textbf{ in } ys, \textbf{let } B' \textbf{ in } ys) \in X\})^{\mathcal{M}}
\end{array}$$

We proceed by fixed point induction on X. In the base case, X is empty, so is clearly included on the right hand side. For the inductive step, we assume $X \subseteq \{(L1, L2) \mid \textbf{let } \mathit{map} = \ldots; f = \lambda x.\mathbb{C}[x]; l = L1 \textbf{ in } \mathit{map}\ T_1\ T_2\ f\ l \underset{\sim}{\rhd} L2\}$ and try to prove the same for the right hand side of the recursive equation. Because both sides of our inclusion are machine-closed, it suffices to show that the following relation is included in the right hand side:

$$\begin{array}{l}
\{(\mathit{Nil}, \mathit{Nil})\} \cup \\
\{\ (\textbf{let } B \textbf{ in } y :: ys, \textbf{let } B' \textbf{ in } y :: ys) \\
\ \mid \textbf{let } B \textbf{ in } \mathbb{C}[y] \underset{\sim}{\rhd} \textbf{let } B' \textbf{ in } y, \\
\quad (\textbf{let } B \textbf{ in } ys, \textbf{let } B' \textbf{ in } ys) \in X\}
\end{array}$$

We proceed by case analysis on the elements of this relation. First of all, the pair $(\mathit{Nil}, \mathit{Nil})$ is included in $\{(L1, L2) \mid \textbf{let } \mathit{map} = \ldots; f = \lambda x.\mathbb{C}[x]; l = L1 \textbf{ in } \mathit{map}\ T_1\ T_2\ f\ l \underset{\sim}{\rhd} L2\}$ simply by applying the definition of *map*. In turn, for $(\textbf{let } B \textbf{ in } y :: ys, \textbf{let } B' \textbf{ in } y :: ys)$ we know the following two properties from the way in which the relation is constructed:

$$\begin{array}{l}
\textbf{let } B \textbf{ in } \mathbb{C}[y] \underset{\sim}{\rhd} \textbf{let } B' \textbf{ in } y \\
(\textbf{let } B \textbf{ in } ys, \textbf{let } B' \textbf{ in } ys) \in X
\end{array}$$

From the inductive hypothesis, the second of these two properties implies:

$$\textbf{let } \mathit{map} = \ldots; f = \lambda x.\mathbb{C}[x]; B \textbf{ in } \mathit{map}\ T_1\ T_2\ f\ ys \underset{\sim}{\rhd} \textbf{let } B' \textbf{ in } ys$$

We then reason as follows:

$$\begin{array}{l}
\quad \textbf{let } \mathit{map} = \ldots; f = \lambda x.\mathbb{C}\,[x];\, l = \textbf{let } B \textbf{ in } y :: \mathit{ys} \textbf{ in } \mathit{map}\; T_1\; T_2\; f\; l \\
\underset{\sim}{\lhd\rhd} \quad \{ \text{flattening } \textbf{lets} \} \\
\quad \textbf{let } \mathit{map} = \ldots; f = \lambda x.\mathbb{C}\,[x];\, B;\, l = y :: \mathit{ys} \textbf{ in } \mathit{map}\; T_1\; T_2\; f\; l \\
\underset{\sim}{\rhd} \quad \{ \text{applying definitions of } \mathit{map}, f \} \\
\quad \textbf{let } \mathit{map} = \ldots; f = \lambda x.\mathbb{C}\,[x];\, B;\, y' = \mathbb{C}\,[y];\, \mathit{ys}' = \mathit{map}\; T_1\; T_2\; f\; \mathit{ys} \textbf{ in } y' :: \mathit{ys}' \\
\underset{\sim}{\rhd} \quad \{ \textbf{let } B \textbf{ in } \mathbb{C}\,[y] \mathrel{\underset{\sim}{\rhd}} \textbf{let } B' \textbf{ in } y, \text{ and} \\
\qquad\quad \textbf{let } \mathit{map} = \ldots; f = \lambda x.\mathbb{C}\,[x];\, B \textbf{ in } \mathit{map}\; T_1\; T_2\; f\; \mathit{ys} \mathrel{\underset{\sim}{\rhd}} \textbf{let } B' \textbf{ in } \mathit{ys} \} \\
\quad \textbf{let } B' \textbf{ in } y :: \mathit{ys}
\end{array}$$

Hence this pair is also included in the right-hand side. □

Now we can specialise our free theorem for *map* to something more immediately useful. Letting $T_1 = T_1'$, $R_1 = \{(M_1, M_2) \mid M_1 \mathrel{\underset{\sim}{\rhd}} M_2\}$ and $R_2 = \{(M_1, M_2) \mid \mathbb{C}\,[M_1] \mathrel{\underset{\sim}{\rhd}} M_2\}$, we obtain the following:

$$\begin{array}{l}
\text{for any bindings } B_2, B_2', \\
\quad \text{if for any bindings } B_1, B_1' \text{ we have} \\
\qquad \textbf{let } B_1 \textbf{ in } x \mathrel{\underset{\sim}{\rhd}} \textbf{let } B_1' \textbf{ in } x \Rightarrow \textbf{let } B_1 \textbf{ in } \mathbb{C}\,[(\textbf{let } B_2 \textbf{ in } f)\; x] \mathrel{\underset{\sim}{\rhd}} \textbf{let } B_1' \textbf{ in } (\textbf{let } B_2' \textbf{ in } f)\; x \\
\quad \text{and } (\textbf{let } B_2 \textbf{ in } \mathit{xs}, \textbf{let } B_2' \textbf{ in } \mathit{xs}) \in \mathit{List}\; R_1 \\
\quad \text{then } (\textbf{let } B_2 \textbf{ in } \mathit{map}\; T_1\; T_2\; f\; \mathit{xs}, \textbf{let } B_2' \textbf{ in } \mathit{map}\; T_1\; T_2'\; f\; \mathit{xs}) \in \mathit{List}\; R_2
\end{array}$$

We note that $\mathit{List}\; R_1$ contains the improvement relation, and simplify:

$$\begin{array}{l}
\text{for any bindings } B_2, B_2', \\
\quad \text{if for any bindings } B_1 \text{ we have} \\
\qquad \textbf{let } B_1 \textbf{ in } \mathbb{C}\,[(\textbf{let } B_2 \textbf{ in } f)\; x] \mathrel{\underset{\sim}{\rhd}} \textbf{let } B_1 \textbf{ in } (\textbf{let } B_2' \textbf{ in } f)\; x \\
\quad \text{and } \textbf{let } B_2 \textbf{ in } \mathit{xs} \mathrel{\underset{\sim}{\rhd}} \textbf{let } B_2' \textbf{ in } \mathit{xs} \\
\quad \text{then } \textbf{let } h = \lambda x.\mathbb{C}\,[x];\, l = \textbf{let } B_2 \textbf{ in } \mathit{map}\; T_1\; T_2\; f\; \mathit{xs} \textbf{ in } \mathit{map}\; T_2\; T_2'\; h\; l \mathrel{\underset{\sim}{\rhd}} \\
\textbf{let } B_2' \textbf{ in } \mathit{map}\; T_1\; T_2'\; f\; \mathit{xs}
\end{array}$$

Finally, we let $B_2 = \{f = P,\ \mathit{xs} = M\}$, $B_2' = \{f = Q,\ \mathit{xs} = M\}$ and obtain:

$$\begin{array}{l}
\text{if for any bindings } B_1 \text{ we have} \\
\quad \textbf{let } B_1 \textbf{ in } \mathbb{C}\,[P\; x] \mathrel{\underset{\sim}{\rhd}} \textbf{let } B_1 \textbf{ in } Q\; x \\
\text{then } \textbf{let } h = \lambda x.\mathbb{C}\,[x];\, f = P;\, \mathit{xs} = M;\, l = \mathit{map}\; T_1\; T_2\; f\; \mathit{xs} \textbf{ in } \mathit{map}\; T_2\; T_2'\; h\; l \\
\qquad \mathrel{\underset{\sim}{\rhd}} \textbf{let } g = Q;\, \mathit{xs} = M \textbf{ in } \mathit{map}\; T_1\; T_2'\; g\; \mathit{xs}
\end{array}$$

Note that B_1 is quantified over *all* bindings, including those that bind variables free in $\mathbb{C}$, P and Q; however, in practice $\mathbb{C}$, P and Q will typically not contain free variables. The result states that if the function P can be fused with the context $\mathbb{C}$ to make an improved function Q, then the combination of $\mathit{map}\; (\lambda x.\mathbb{C}\,[x])$ with $\mathit{map}\; P$ is improved by $\mathit{map}\; Q$. This establishes that map fusion is indeed an efficiency optimisation in terms of time performance.

5.3 Example: Fold Fusion

Now consider the *fold* function on lists, with the following type:

$$\mathit{fold} : \forall\, \alpha\; \beta.(\alpha \to \beta \to \beta) \to \beta \to \mathit{List}\; \alpha \to \beta$$

Applying our abstraction theorem, we obtain:

$$\begin{array}{l}
\text{for any bindings } B_2, B_2' \text{ and machine-closed relations } R_1 : T_1 \leftrightarrow T_1', R_2 : T_2 \leftrightarrow T_2', \\
\quad \text{if for any bindings } B_1, B_1' \text{ we have} \\
\qquad (\textbf{let } B_1 \textbf{ in } x, \textbf{let } B_1' \textbf{ in } x) \in R_1 \text{ and } (\textbf{let } B_1 \textbf{ in } y, \textbf{let } B_1' \textbf{ in } y) \in R_2 \\
\qquad\quad \text{together imply } (\textbf{let } B_1; B_2 \textbf{ in } h\; x\; y, \textbf{let } B_1'; B_2' \textbf{ in } h'\; x\; y) \in R_2
\end{array}$$

and $(\textbf{let } B_2 \textbf{ in } n,\ \textbf{let } B_2' \textbf{ in } n') \in R_2$
and $(\textbf{let } B_2 \textbf{ in } l,\ \textbf{let } B_2' \textbf{ in } l') \in \mathit{List}\ R_1$
then $(\textbf{let } B_2 \textbf{ in } \mathit{fold}\ T_1\ T_2\ h\ n\ l,\ \textbf{let } B_2' \textbf{ in } \mathit{fold}\ T_1'\ T_2'\ h'\ n'\ l') \in R_2$

Letting $T_1 = T_1'$, $R_1 = \{(M_1, M_2) \mid M_1 \underset{\sim}{\triangleright} M_2\}$, $R_2 = \{(M_1, M_2) \mid \mathbb{C}\,[M_1] \underset{\sim}{\triangleright} M_2\}$, we obtain:

for any bindings B_2, B_2',
if for any bindings B_1, B_1' we have
$\textbf{let } B_1 \textbf{ in } x \underset{\sim}{\triangleright} \textbf{let } B_1' \textbf{ in } x$ and $\textbf{let } B_1 \textbf{ in } \mathbb{C}\,[y] \underset{\sim}{\triangleright} \textbf{let } B_1' \textbf{ in } y$
together imply $\textbf{let } B_1; B_2 \textbf{ in } \mathbb{C}\,[h\ x\ y] \underset{\sim}{\triangleright} \textbf{let } B_1'; B_2' \textbf{ in } h'\ x\ y$
and $\textbf{let } B_2 \textbf{ in } \mathbb{C}\,[n] \underset{\sim}{\triangleright} \textbf{let } B_2' \textbf{ in } n'$
then $\textbf{let } B_2 \textbf{ in } \mathbb{C}\,[\mathit{fold}\ T_1\ T_2\ h\ n\ l] \underset{\sim}{\triangleright} \textbf{let } B_2' \textbf{ in } \mathit{fold}\ T_1\ T_2'\ h'\ n'\ l$

Simplifying further, we obtain:

for any bindings B_2, B_2',
if for any bindings B_1 we have
$\textbf{let } B_1; B_2 \textbf{ in } \mathbb{C}\,[h\ x\ y] \underset{\sim}{\triangleright} \textbf{let } B_1; B_2'; y' = \mathbb{C}\,[y] \textbf{ in } h'\ x\ y'$
and $\textbf{let } B_2 \textbf{ in } \mathbb{C}\,[n] \underset{\sim}{\triangleright} \textbf{let } B_2' \textbf{ in } n'$
then $\textbf{let } B_2 \textbf{ in } \mathbb{C}\,[\mathit{fold}\ T_1\ T_2\ h\ n\ l] \underset{\sim}{\triangleright} \textbf{let } B_2' \textbf{ in } \mathit{fold}\ T_1\ T_2'\ h'\ n'\ l$

This is an improving version of the usual fusion rule for *fold* [Meijer et al. 1991], much like the improving version of *fix* fusion above, telling us that when the fusion precondition is an improvement in one direction then fusion itself is also an improvement.

5.4 Example: Short Cut Fusion

Short cut fusion, as introduced by Gill et al. [1993], is a general purpose transformation that eliminates the use of intermediate lists in functional programs. The transformation itself is based around *fold* function from the previous example, coupled with a new function called *build*:

$$\begin{aligned} &\mathit{build}\ :\ \forall\,\alpha.(\forall\,\beta.(\alpha \to \beta \to \beta) \to \beta \to \beta) \to \mathit{List}\ \alpha \\ &\mathit{build} = \Lambda\alpha.\lambda g \to g\ (\mathit{List}\ \alpha)\ (\lambda x\ xs \to x :: xs)\ \mathit{Nil} \end{aligned}$$

Essentially, *build* is used to produce lists by abstracting over the two list constructors, while *fold* is used to consume the resulting lists in a structured manner by replacing the two constructors. These two functions are linked by the following equation:

$$\mathit{fold}\ h\ n\ (\mathit{build}\ g) \quad = \quad g\ h\ n$$

Short cut fusion involves applying this equation anywhere the left-hand side appears, thereby removing the intermediate list. The strength of this technique comes from the generality of *build*. In particular, many library functions that produce lists can be implemented using *build*, and any function implemented in such a way then becomes a candidate for short cut fusion.

Unlike the previous examples, which could in theory be proved from the structure of the function in question, the structure of g is not known ahead of time. As such, parametricity is *essential* in the proof of correctness of short cut fusion. This suggests that we may be able to prove a result about its efficiency properties using our new abstraction theorem. First of all, instantiating the abstraction theorem for the type of our function g gives:

for any bindings B_2, B_2' and machine-closed relation $R\ :\ T_1 \leftrightarrow T_1'$,
if for any bindings B_1, B_1' we have
$(\textbf{let } B_1 \textbf{ in } x,\ \textbf{let } B_1' \textbf{ in } x) \in \Delta\ ()\ (A)$ and $(\textbf{let } B_1 \textbf{ in } y,\ \textbf{let } B_1' \textbf{ in } y) \in R$
together imply $(\textbf{let } B_1; B_2 \textbf{ in } f\ x\ y,\ \textbf{let } B_1'; B_2' \textbf{ in } f\ x\ y) \in R$

and ($\textbf{let } B_2 \textbf{ in } c$, $\textbf{let } B_2' \textbf{ in } c) \in R$
then ($\textbf{let } B_2 \textbf{ in } g\ f\ c$, $\textbf{let } B_2' \textbf{ in } g\ f\ c) \in R$

In this case, we let $T_1 = \mathit{List}\ A$, $R = \{(M_1, M_2) \mid \textbf{let } xs = M_1 \textbf{ in } \mathit{fold}\ h\ n\ xs \underset{\sim}{\rhd} M_2\}$, $B_2 = \{f = \lambda x\ xs.x :: xs; c = \mathit{Nil}\}$ and $B_2' = \{f = h; c = n\}$. The fact that R is machine-closed follows from the strictness of the *fold* function on lists. We obtain:

if for any bindings B_1, B_1' we have
($\textbf{let } B_1 \textbf{ in } x$, $\textbf{let } B_1' \textbf{ in } x) \in \Delta\ ()\ (A)$ and $\textbf{let } B_1 \textbf{ in } \mathit{fold}\ h\ n\ y \underset{\sim}{\rhd} \textbf{let } B_1' \textbf{ in } y$
together imply $\textbf{let } B_1; xs = x :: y \textbf{ in } \mathit{fold}\ h\ n\ xs \underset{\sim}{\rhd} \textbf{let } B_1' \textbf{ in } h\ x\ y$
and $\mathit{fold}\ h\ n\ \mathit{Nil} \underset{\sim}{\rhd} n$
then $\textbf{let } f = \lambda x\ xs.x :: xs; c = \mathit{Nil} \textbf{ in } \mathit{fold}\ h\ n\ (g\ f\ c) \underset{\sim}{\rhd} \textbf{let } f = h; c = n \textbf{ in } g\ f\ c$

We now aim to prove the precondition. We assume ($\textbf{let } B_1 \textbf{ in } x$, $\textbf{let } B_1' \textbf{ in } x) \in \Delta\ ()\ (A)$ and $\textbf{let } B_1 \textbf{ in } \mathit{fold}\ h\ n\ y \underset{\sim}{\rhd} \textbf{let } B_1' \textbf{ in } y$, and attempt to prove $\textbf{let } B_1; xs = x :: y \textbf{ in } \mathit{fold}\ h\ n\ xs \underset{\sim}{\rhd} \textbf{let } B_1' \textbf{ in } h\ x\ y$. Note that by our equivalent of the identity extension lemma the first assumption implies $\textbf{let } B_1 \textbf{ in } x \underset{\sim}{\rhd} \textbf{let } B_1' \textbf{ in } x$. We reason as follows:

$$
\begin{array}{ll}
& \textbf{let } B_1; xs = x :: y \textbf{ in } \mathit{fold}\ h\ n\ xs \\
\underset{\sim}{\rhd} & \{ \text{definition of } \mathit{fold} \} \\
& \textbf{let } B_1 \textbf{ in } h\ x\ (\mathit{fold}\ h\ n\ y) \\
\underset{\sim}{\rhd} & \{ \text{assumptions} \} \\
& \textbf{let } B_1' \textbf{ in } h\ x\ y
\end{array}
$$

We can therefore conclude that $\textbf{let } f = \lambda x\ xs.x :: xs; c = \mathit{Nil} \textbf{ in } \mathit{fold}\ h\ n\ (g\ f\ c) \underset{\sim}{\rhd} g\ h\ n$, which establishes that applying short cut fusion is indeed in an improvement.

6 CONCLUSION AND FURTHER WORK

We have shown that a parametricity result holds for a call-by-need operational semantics with a notion of program cost, extending the work of Pitts [2000]. The resulting abstraction theorem can be used to derive free theorems that give results about program efficiency, giving conditions under which program transformations are guaranteed to maintain or improve the time performance of programs. We have applied our theorem to a range of examples, including the well-known short cut fusion of Gill et al. [1993], showing that these particular transformations are all *safe*, in the sense that they do not degrade performance. This is an important step towards making formal reasoning about the performance of call-by-need programs tractable.

This work considers the time performance of call-by-need programs. However, *space* behaviour of call-by-need programs can also be counterintuitive, and this raises the question of whether a similar parametricity result will hold. This work was based on call-by-need time improvement as developed by [Moran and Sands 1999a]; corresponding work for space costs could be based on the theory of [Gustavsson and Sands 1999, 2001]. Ultimately, it would be best to have a unified theory for both space and time based on some abstract notion of resource, so that this asymmetry can be avoided. The work of Sands [1997] may offer a way forward here, and we are also in the process of developing a generic foundation for program improvement [Hackett and Hutton 2018], based upon the use of metric spaces to abstract over various aspects including the cost model.

Another way to extend this work would be to consider selective strictness, where an operator such as Haskell's *seq* can be used to force evaluation of subterms. Parametricity in the presence of *seq* has already been investigated both denotationally [Johann and Voigtländer 2004] and operationally in terms of program equivalence and partial equivalence [Voigtländer and Johann 2007], so it would be a natural next step to attempt to do the same for program improvement.

Improvement-theory based techniques can show that a transformation does not have a negative impact on performance, but there are other questions we may wish to ask about our optimisations. In particular, we would like to quantify *how much* performance is improved, and whether this change is merely a constant factor or an actual asymptotic improvement. This would also allow us to verify the properties of optimisations that are not simple improvements, such as those that improve performance in all but a few pathological cases.

This work inherits the limitation of Moran and Sands [1999a], in that it can only be used to reason about certain forms of recursive definition, namely those where the right-hand side is in a *value* form. This covers the usual case of recursive functions, because the right-hand side is then a lambda-term and hence in value form, but it fails to address a number of programs written in the so-called "knot-tying" style. For example, the function *cycle* that takes a list and produces the list of infinite repetitions of that list, can naturally be written in two ways:

(i) $cycle\ xs = xs +\!\!+\ cycle\ xs$

(ii) $cycle\ xs = \mathbf{let}\ r = xs +\!\!+\ r\ \mathbf{in}\ r$

Our theory allows us to reason about the first definition, but not the second. The problem is the unwinding lemma (4.1) fails when the right-hand side of the definition is unrestricted, as when $x = M\ [x]$ is unwound to the infinite sequence $x_0 = \bot; x_1 = M\ [x_0]; x_2 = M\ [x_1] \ldots$ we lose sharing between the different levels of recursion. We believe this limitation can be removed by considering *bindings with fuel* instead of unwound bindings, where $x =^n M$ is treated as a binding that allows x to be looked up at most n times. However, while the unwinding lemma holds for this form of binding, we have yet to prove the corresponding Let case for the abstraction theorem.

Deriving free theorems can be time consuming. Generation of standard free theorems has been mechanised for a sublanguage of Haskell [Böhme 2007], so the work of this article could potentially also be mechanised. This would benefit from further work on machine-closed relations, so as to increase the number of ways we can specialise these theorems. In particular, it would be useful to know whether the inclusion from Theorem 5.2 can be strengthened to an equality.

Recently, we have developed the UNIE system [Handley and Hutton 2018], an inequational reasoning assistant that provides mechanical support for improvement proofs. This tool takes care of the administrative work in improvement proofs, allowing users to focus on the high-level structure of their proofs. At present, UNIE is based on the standard untyped improvement theory, but it could be extended to include a type system and parametricity-based results.

Our concept of machine-closure is based on execution costs in an abstract machine. Similar ideas have been explored in quantitative realizability models [Brunel 2015; Brunel and Gaboardi 2015], where the notion of a *ρ-behavior* has a similar structure to our notion of a machine-closed relation (as well as Pitts' $\top\top$-closed relations). It would be interesting to explore this connection further.

Acknowledgements

We would like to thank Martin Handley and the referees for many useful comments and suggestions. This work was funded by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/P00587X/1, *Mind the Gap: Unified Reasoning About Program Correctness and Efficiency.*

REFERENCES

Sascha Böhme. 2007. *Free Theorems for Sublanguages of Haskell.* Master's thesis. Technische Universität Dresden.

Joachim Breitner. 2015. Formally Proving a Compiler Transformation Safe. In *Haskell Symposium.*

Aloïs Brunel. 2015. Quantitative Classical Realizability. *Information and Computation* 241 (2015).

Aloïs Brunel and Marco Gaboardi. 2015. Realizability Models for a Linear Dependent PCF. *Theoretical Computer Science* 585 (2015).

Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Functional Programming Languages and Computer Architecture*.

Jörgen Gustavsson and David Sands. 1999. A Foundation for Space-Safe Transformations of Call-by-Need Programs. *Electronic Notes on Theoretical Computer Science* 26 (1999).

Jörgen Gustavsson and David Sands. 2001. Possibilities and Limitations of Call-by-Need Space Improvement. In *International Conference on Functional Programming*.

Jennifer Hackett and Graham Hutton. 2014. Worker/Wrapper/Makes It/Faster. In *International Conference on Functional Programming*.

Jennifer Hackett and Graham Hutton. 2015. Programs for Cheap!. In *Logic in Computer Science*.

Jennifer Hackett and Graham Hutton. 2018. A Generic Foundation for Program Improvement. (2018). In preparation.

Martin Handley and Graham Hutton. 2018. Improving Haskell. In *Trends in Functional Programming*.

Graham Hutton and Jennifer Hackett. 2016. Mind the Gap: Unified Reasoning About Program Correctness and Efficiency. (2016). Engineering and Physical Sciences Research Council grant EP/P00587X/1.

Patricia Johann and Janis Voigtländer. 2004. Free Theorems in the Presence of *seq*. In *Principles of Programming Languages*.

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Principles of Programming Languages*.

Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture '91 (Lecture Notes in Computer Science)*, Vol. 523. Springer.

Andrew Moran and David Sands. 1999a. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. (1999). Extended version of [Moran and Sands 1999b], available at http://tinyurl.com/ohuv8ox.

Andrew Moran and David Sands. 1999b. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *Principles of Programming Languages*.

Andrew M. Pitts. 2000. Parametric Polymorphism and Operational Equivalence. *Mathematical Structures in Computer Science* 10, 03 (2000).

Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theoretical Computer Science* 5, 3 (1977).

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. *Proceedings of the IFIP 9th World Computer Congress* (1983).

David Sands. 1997. From SOS Rules to Proof Principles: An Operational Metatheory for Functional Languages. In *Principles of Programming Languages*.

Manfred Schmidt-Schauß and David Sabel. 2015. Improvements in a Functional Core Language with Call-By-Need Operational Semantics. In *Principles and Practice of Declarative Programming*.

Daniel Seidel and Janis Voigtländer. 2011. Improvements for Free. In *Quantitative Aspects of Programming Languages*.

Ilya Sergey, Dimitrios Vytiniotis, Simon L Peyton Jones, and Joachim Breitner. 2017. Modular, Higher Order Cardinality Analysis in Theory and Practice. *Journal of Functional Programming* 27 (2017).

Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. *Journal of Functional Programming* 7, 3 (1997).

Jon Shultis. 1985. *On the Complexity of Higher-Order Programs*. Technical Report. University of Colorado.

Hugo Simões, Pedro Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. 2012. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *International Conference on Functional Programming*.

Christopher Strachey. 2000. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation* 13, 1-2 (2000).

Janis Voigtländer and Patricia Johann. 2007. Selective Strictness and Parametricity in Structural Operational Semantics, Inequationally. *Theoretical Computer Science* 388, 1-3 (2007).

Philip Wadler. 1989. Theorems for Free!. In *Functional Programming Languages and Computer Architecture*.

A PROOF OF LEMMA 3.4

We proceed by induction on the derivation of the typing judgement $\Gamma \vdash M : T$ according to the rules in Figure 1, considering each of the nine cases in turn.

Var Follows immediately from the definition of Δ.

Abs In this case, M is of the form $\lambda y.M'$ and T is of the form $T_1 \to T_2$. The inductive hypothesis is that $\Gamma,\ y : T_1 \vdash M' \ \Delta\ M' : T_2$. We must prove that $\Gamma \vdash \lambda y.M' \ \Delta\ \lambda y.M' : T_1 \to T_2$.

We let B_1, B_2 be bindings that close $\lambda y.M'$ and let $\vec{R} : \vec{T} \leftrightarrow \vec{T}'$ be a list of machine-closed relations with length equal to the number of free type variables in Γ. Assume that for all $x : A$ in Γ, $(\textbf{let } B_1 \textbf{ in } x,\ \textbf{let } B_2 \textbf{ in } x) \in \Delta\ (\vec{R}/\vec{\alpha})\ (A)$, and let B_1' and B_2' be extensions of B_1 and B_2 such that $(\textbf{let } B_1' \textbf{ in } y,\ \textbf{let } B_2' \textbf{ in } y) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_1)$ also. We reason as follows:

$$
\begin{array}{l}
\quad \{\text{ induction hypothesis }\} \\
\Rightarrow \\
\quad (\textbf{let } B_1' \textbf{ in } M'\ [\vec{T}/\vec{\alpha}], \\
\qquad \textbf{let } B_2' \textbf{ in } M'\ [\vec{T}'/\vec{\alpha}]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_2) \\
\Leftrightarrow \quad \{\ \beta\text{-reduction, machine-closure }\} \\
\quad (\textbf{let } B_1' \textbf{ in } ((\lambda y.M')\ y)\ [\vec{T}/\vec{\alpha}], \\
\qquad \textbf{let } B_2' \textbf{ in } ((\lambda y.M')\ y)\ [\vec{T}'/\vec{\alpha}]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_2) \\
\Leftrightarrow \quad \{\text{ definition of} \to \text{action on relations }\} \\
\quad (\textbf{let } B_1' \textbf{ in } (\lambda y.M')\ [\vec{T}/\vec{\alpha}], \\
\qquad \textbf{let } B_2' \textbf{ in } (\lambda y.M')\ [\vec{T}'/\vec{\alpha}]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_1 \to T_2) \\
\Leftrightarrow \quad \{\text{ removing extra bindings }\} \\
\quad (\textbf{let } B_1 \textbf{ in } (\lambda y.M')\ [\vec{T}/\vec{\alpha}], \\
\qquad \textbf{let } B_2 \textbf{ in } (\lambda y.M')\ [\vec{T}'/\vec{\alpha}]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_1 \to T_2)
\end{array}
$$

The removing bindings step is valid because $\textbf{let } B_1 \textbf{ in } (\lambda y.M')$ and $\textbf{let } B_2 \textbf{ in } (\lambda y.M')$ were already closed, so the extra bindings in the extended versions cannot affect evaluation. Because all relations are machine-closed, this means that the terms before and after removing bindings must be treated equally. As this reasoning was generic in $\vec{R}$, B_1 and B_2, we can conclude that $\Gamma \vdash \lambda y.M' \ \Delta\ \lambda y.M' : T_1 \to T_2$, as required.

App Follows straightforwardly from the definitions of Δ and the relational action of $\to$.

TAbs In this case, M is of the form $\Lambda\alpha.M'$ and T is of the form $\forall\ \alpha.T'$. The inductive hypothesis is that $\Gamma,\ \alpha \vdash M' \ \Delta\ M' : T'$. We must prove that $\Gamma \vdash \Lambda\alpha.M' \ \Delta\ \Lambda\alpha.M' : \forall\ \alpha.T'$.

We let B_1, B_2 be bindings that close M' and let $\vec{R} : \vec{T} \leftrightarrow \vec{T}'$ be a list of machine-closed relations with length equal to the number of free type variables in Γ. Assume that for all $x : A$ in Γ, $(\textbf{let } B_1 \textbf{ in } x,\ \textbf{let } B_2 \textbf{ in } x) \in \Delta\ (\vec{R}/\vec{\alpha})\ (A)$, and let X, X' be arbitrary types with an arbitrary machine-closed relation $R : X \leftrightarrow X'$. We reason as follows:

$$
\begin{array}{l}
\quad \{\text{ induction hypothesis }\} \\
\Rightarrow \\
\quad (\textbf{let } B_1 \textbf{ in } M'\ [X/\alpha,\ \vec{T}/\vec{\alpha}], \\
\qquad \textbf{let } B_2 \textbf{ in } M'\ [X'/\alpha,\ \vec{T}'/\vec{\alpha}]) \in \Delta\ (R/\alpha,\ \vec{R}/\vec{\alpha}) \\
\Leftrightarrow \quad \{\text{ type-}\beta\text{-reduction, machine-closure }\}
\end{array}
$$

$$\begin{aligned}&(\textbf{let } B_1 \textbf{ in } ((\Lambda\alpha.M')\ X)\ [\vec{T}/\vec{\alpha}],\\ &\quad \textbf{let } B_2 \textbf{ in } ((\Lambda\alpha.M')\ X')\ [\vec{T}'/\vec{\alpha}]) \in \Delta\ (R/\alpha,\ \vec{R}/\vec{\alpha})\ (T')\end{aligned}$$

Because this reasoning is generic in X, X' and $R : X \leftrightarrow X'$, and because R is machine-closed, we can conclude:

$$\begin{aligned}&(\textbf{let } B_1 \textbf{ in } (\Lambda\alpha.M')\ [\vec{T}/\vec{\alpha}],\\ &\quad \textbf{let } B_2 \textbf{ in } (\Lambda\alpha.M')\ [\vec{T}'/\vec{\alpha}]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (\forall\ \alpha.T')\end{aligned}$$

Finally, because all of the above reasoning was generic in $\vec{R}$, B_1 and B_2, we can conclude that $\Gamma \vdash \Lambda\alpha.M' \ \Delta\ \Lambda\alpha.M' : \forall\ \alpha.T'$, as required.

TApp Follows straightforwardly from the definitions of Δ and the relational action of $\forall$.

Nil Using the fact that **let** B **in** $Nil \underset{\sim}{\lhd\!\rhd} Nil$, this case follows from the definition of the logical relation Δ and the relational action of *List*.

Cons Follows immediately from the definition of Δ and the relational action of *List*.

Case In this case, we know that M is of the form **case** M' **of** $\{Nil \rightarrow M_1; x :: xs \rightarrow M_2\}$. The induction hypotheses are that $\Gamma \vdash M'\ \Delta\ M' : List\ T'$, $\Gamma \vdash M_1\ \Delta\ M_1 : T$, $\Gamma, x : T', xs : List\ T' \vdash M_2 : T$ for some type T'.

We assume that x, xs are fresh, which can be ensured by alpha-renaming. We let B_1, B_2 be bindings that close M and let $\vec{R} : \vec{T} \leftrightarrow \vec{T}'$ be a list of machine-closed relations with length equal to the number of free type variables in Γ.

To prove $\Gamma \vdash$ **case** M' **of** $\{\dots\}\ \Delta$ **case** M' **of** $\{\dots\} : T$ we assume that for all $y : A \in \Gamma$ we have

$$\begin{aligned}&(\textbf{let } B_1 \textbf{ in } y,\\ &\quad \textbf{let } B_2 \textbf{ in } y) \in \Delta\ (\vec{R}/\vec{\alpha})\ (A)\end{aligned}$$

and try to prove

$$\begin{aligned}&(\textbf{let } B_1 \textbf{ in case } M'\ [\vec{T}/\vec{\alpha}] \textbf{ of } \{Nil \rightarrow M_1\ [\vec{T}/\vec{\alpha}]; x :: xs \rightarrow M_2\ [\vec{T}/\vec{\alpha}]\},\\ &\quad \textbf{let } B_2 \textbf{ in case } M'\ [\vec{T}'/\vec{\alpha}] \textbf{ of } \{Nil \rightarrow M_1\ [\vec{T}'/\vec{\alpha}]; x :: xs \rightarrow M_2\ [\vec{T}'/\vec{\alpha}]\})\\ &\qquad \in \Delta\ (\vec{R}/\vec{\alpha})\ (T)\end{aligned}$$

To prove this, we need the following sub-lemma:

Lemma A.1. *Consider machine-closed relations R_1 R_2, heap and stack pairs $\langle H_1, S_1\rangle$, $\langle H_2, S_2\rangle$, bindings B_1, B_2 and terms M_1, M_1', M_2, M_2' that satisfy the following assumptions:*

(i) (**let** B_1 **in** M_1, **let** B_2 **in** M_1') $\in R_2$

(ii) For all bindings B_1', B_2' and variables y, ys, if

(**let** B_1' **in** y, **let** B_2' **in** y) $\in R_1$

and

(**let** B_1' **in** ys, **let** B_2' **in** ys) $\in List\ R_1$

then it follows that

(**let** B_1' **in** $M_2\ [y/x,\ ys/xs]$, **let** B_2' **in** $M_2'\ [y/x,\ ys/xs]$) $\in R_2$

(iii) For any pair of terms $(N, N') \in R_2$, $\langle H_1, N, S_1\rangle \downarrow_n \Rightarrow \langle H_2, N', S_2\rangle \downarrow_n$

Then for any pair of terms $(L, L') \in List\ R_1$, we have:

$$\langle H_1 + B_1,\ L,\ \{Nil \to M_1; x :: xs \to M_2\}\ :\ S_1\rangle \downarrow_n\ \Rightarrow$$
$$\quad \langle H_2 + B_2,\ L',\ \{Nil \to M_1'; x :: xs \to M_2'\}\ :\ S_2\rangle \downarrow_n$$

Proof. Because $List\ R_1$ is the machine-closure of $Rel = \{Nil, Nil\} \cup \{(\textbf{let}\ B_1\ \textbf{in}\ y :: ys,\ \textbf{let}\ B_2\ \textbf{in}\ y :: ys) \mid (\textbf{let}\ B_1\ \textbf{in}\ y,\ \textbf{let}\ B_2\ \textbf{in}\ y) \in R_1,\ (\textbf{let}\ B_1\ \textbf{in}\ ys,\ \textbf{let}\ B_2\ \textbf{in}\ ys) \in List\ R_1\}$, the definition of machine-closure implies that our result holds if $\langle H_1 + B_1,\ L,\ \{Nil \to M_1; x :: xs \to M_2\}\ :\ S_1\rangle \downarrow_n \Rightarrow \langle H_2 + B_2,\ L',\ \{Nil \to M_1'; x :: xs \to M_2'\}\ :\ S_2\rangle \downarrow_n$ for any (L, L') in the underlying set Rel. This (L, L') will be of one of two forms.

Firstly, we consider the case of (Nil, Nil). In this case, we reason as follows:

$$\begin{array}{ll}
& \langle H_1 + B_1,\ Nil,\ \{Nil \to M_1; x :: xs \to M_2\}\ :\ S_1\rangle \downarrow_n \\
\Leftrightarrow & \{\ \textsc{BranchNil}\ \} \\
& \langle H_1 + B_1,\ M_1,\ S_1\rangle \downarrow_n \\
\Rightarrow & \{\ \text{conditions (i) and (iii)}\ \} \\
& \langle H_2 + B_2,\ M_2,\ S_2\rangle \downarrow_n \\
\Leftrightarrow & \{\ \textsc{BranchNil}\ \} \\
& \langle H_2 + B_2,\ Nil,\ \{Nil \to M_1'; x :: xs \to M_2'\}\ :\ S_2\rangle \downarrow_n
\end{array}$$

Next, we consider the case of $(\textbf{let}\ B_1'\ \textbf{in}\ y :: ys,\ \textbf{let}\ B_2'\ \textbf{in}\ y :: ys)$. In this case, we know that $(\textbf{let}\ B_1'\ \textbf{in}\ y,\ \textbf{let}\ B_2'\ \textbf{in}\ y) \in R_1$ and $(\textbf{let}\ B_1'\ \textbf{in}\ ys,\ \textbf{let}\ B_2'\ \textbf{in}\ ys) \in List\ R_1$. We reason as follows:

$$\begin{array}{ll}
& \langle H_1 + B_1,\ \textbf{let}\ B_1'\ \textbf{in}\ y :: ys,\ \{Nil \to M_1; x :: xs \to M_2\}\ :\ S_1\rangle \downarrow_n \\
\Leftrightarrow & \{\ \textsc{Letrec}\ \} \\
& \langle H_1 + B_1 + B_1',\ y :: ys,\ \{Nil \to M_1; x :: xs \to M_2\}\ :\ S_1\rangle \downarrow_n \\
\Leftrightarrow & \{\ \textsc{BranchCons}\ \} \\
& \langle H_1 + B_1 + B_1',\ M_2\ [y/x,\ ys/xs],\ S_1\rangle \downarrow_n \\
\Leftrightarrow & \{\ \textsc{Letrec}\ \} \\
& \langle H_1 + B_1,\ \textbf{let}\ B_1'\ \textbf{in}\ M_2\ [y/x,\ ys/xs],\ S_1\rangle \downarrow_n \\
\Rightarrow & \{\ \text{conditions (ii) and (iii)}\ \} \\
& \langle H_2 + B_2,\ \textbf{let}\ B_2'\ \textbf{in}\ M_2'\ [y/x,\ ys/xs],\ S_2\rangle \downarrow_n \\
\Leftrightarrow & \{\ \textsc{Letrec}\ \} \\
& \langle H_2 + B_2 + B_2',\ M_2'\ [y/x,\ ys/xs],\ S_2\rangle \downarrow_n \\
\Leftrightarrow & \{\ \textsc{BranchCons}\ \} \\
& \langle H_2 + B_2 + B_2',\ y :: ys,\ \{Nil \to M_1'; x :: xs \to M_2'\}\ :\ S_2\rangle \downarrow_n \\
\Leftrightarrow & \{\ \textsc{Letrec}\ \} \\
& \langle H_2 + B_2,\ \textbf{let}\ B_2'\ \textbf{in}\ y :: ys,\ \{Nil \to M_1'; x :: xs \to M_2'\}\ :\ S_2\rangle \downarrow_n
\end{array}$$

□

Now we can prove this case of the main lemma. Let $\langle H_1, S_1\rangle$ and $\langle H_2, S_2\rangle$ be heap and stack pairs such that for any pair of terms $(N, N') \in \Delta\ (\vec{R}/\vec{\alpha})\ (T)$, we have $\langle H_1, N, S_1\rangle \downarrow_n \Rightarrow \langle H_2, N', S_2\rangle \downarrow_n$. We then reason as follows:

$$\begin{array}{ll}
& \langle H_1,\ \textbf{let}\ B_1\ \textbf{in case}\ M'\ [\vec{T}/\vec{\alpha}]\ \textbf{of}\ \{Nil \to M_1\ [\vec{T}/\vec{\alpha}]; x :: xs \to M_2\ [\vec{T}/\vec{\alpha}]\},\ S_1\rangle \downarrow_n \\
\Leftrightarrow & \{\ \textsc{Letrec}\ \} \\
& \langle H_1 + B_1,\ \textbf{case}\ M'\ [\vec{T}/\vec{\alpha}]\ \textbf{of}\ \{Nil \to M_1\ [\vec{T}/\vec{\alpha}]; x :: xs \to M_2\ [\vec{T}/\vec{\alpha}]\},\ S_1\rangle \downarrow_n \\
\Leftrightarrow & \{\ \textsc{Case}\ \} \\
& \langle H_1 + B_1,\ M'\ [\vec{T}/\vec{\alpha}],\ \{Nil \to M_1\ [\vec{T}/\vec{\alpha}]; x :: xs \to M_2\ [\vec{T}/\vec{\alpha}]\}\ :\ S_1\rangle \downarrow_n \\
\Rightarrow & \{\ \text{lemma; conditions (i) and (ii) follow from IHs, (iii) holds by assumption}\ \}
\end{array}$$

$$\begin{array}{ll} & \langle H_2 + B_2,\ M'\ [\vec{T}'/\vec{\alpha}],\ \{\mathit{Nil} \rightarrow M_1\ [\vec{T}'/\vec{\alpha}]; x :: xs \rightarrow M_2\ [\vec{T}'/\vec{\alpha}]\}\ :\ S_2\rangle \downarrow_n \\ \Leftrightarrow & \{\ \textsc{Case}\ \} \\ & \langle H_2 + B_2,\ \textbf{case}\ M'\ [\vec{T}'/\vec{\alpha}]\ \textbf{of}\ \{\mathit{Nil} \rightarrow M_1\ [\vec{T}'/\vec{\alpha}]; x :: xs \rightarrow M_2\ [\vec{T}'/\vec{\alpha}]\},\ S_2\rangle \downarrow_n \\ \Leftrightarrow & \{\ \textsc{Letrec}\ \} \\ & \langle H_2,\ \textbf{let}\ B_2\ \textbf{in case}\ M'\ [\vec{T}'/\vec{\alpha}]\ \textbf{of}\ \{\mathit{Nil} \rightarrow M_1\ [\vec{T}'/\vec{\alpha}]; x :: xs \rightarrow M_2\ [\vec{T}'/\vec{\alpha}]\},\ S_2\rangle \downarrow_n \end{array}$$

But this reasoning was generic in $\langle H_1, S_1\rangle$, $\langle H_2, S_2\rangle$, so by machine-closure we have:

$$\begin{array}{l}(\textbf{let}\ B_1\ \textbf{in case}\ M'\ [\vec{T}/\vec{\alpha}]\ \textbf{of}\ \{\mathit{Nil} \rightarrow M_1\ [\vec{T}/\vec{\alpha}]; x :: xs \rightarrow M_2\ [\vec{T}/\vec{\alpha}]\}, \\ \quad \textbf{let}\ B_2\ \textbf{in case}\ M'\ [\vec{T}'/\vec{\alpha}]\ \textbf{of}\ \{\mathit{Nil} \rightarrow M_1\ [\vec{T}'/\vec{\alpha}]; x :: xs \rightarrow M_2\ [\vec{T}'/\vec{\alpha}]\}) \\ \qquad \in \Delta\ (\vec{R}/\vec{\alpha})\ (T)\end{array}$$

Finally, since this reasoning was generic in $\vec{R}$, B_1 and B_2, we can then conclude that $\Gamma \vdash \textbf{case}\ M'\ \textbf{of}\ \{\ldots\}\ \Delta\ \textbf{case}\ M'\ \textbf{of}\ \{\ldots\}\ :\ T$, as required.

Let' In this case, M is of the form $\textbf{let}\ \vec{x} = \vec{M}\ \textbf{in}\ M'$. The induction hypotheses are that $\Gamma \vdash M_i\ \Delta\ M_i\ :\ T_i$ for all i, and $\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash M'\ \Delta\ M'\ :\ T$.

We let B_1, B_2 be bindings that close $\textbf{let}\ \vec{x} = \vec{M}\ \textbf{in}\ M'$ and let $\vec{R} : \vec{T} \leftrightarrow \vec{T}'$ be a list of machine-closed relations with length equal to the number of free type variables in Γ.

To prove $\Gamma \vdash \textbf{let}\ \vec{x} = \vec{M}\ \textbf{in}\ M'\ \Delta\ \textbf{let}\ \vec{x} = \vec{M}\ \textbf{in}\ M'\ :\ T$, we must assume that for all $y : A \in \Gamma$ it is the case that:

$$\begin{array}{l}(\textbf{let}\ B_1\ \textbf{in}\ y, \\ \quad \textbf{let}\ B_2\ \textbf{in}\ y) \in \Delta\ (\vec{R}/\vec{\alpha})\ (A)\end{array}$$

and try to prove:

$$\begin{array}{l}(\textbf{let}\ B_1\ \textbf{in let}\ \vec{x} = \vec{M}\ [\vec{T}/\vec{\alpha}]\ \textbf{in}\ M'\ [\vec{T}/\vec{\alpha}], \\ \quad \textbf{let}\ B_2\ \textbf{in let}\ \vec{x} = \vec{M}\ [\vec{T}'/\vec{\alpha}]\ \textbf{in}\ M'\ [\vec{T}'/\vec{\alpha}]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T)\end{array}$$

We note that the induction hypotheses for the M_i imply:

$$\begin{array}{l}(\textbf{let}\ B_1\ \textbf{in}\ M_i\ [\vec{T}/\vec{\alpha}], \\ \quad \textbf{let}\ B_2\ \textbf{in}\ M_i\ [\vec{T}'/\vec{\alpha}]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_i)\end{array}$$

In turn, by machine closure, this implies that

$$\begin{array}{l}(\textbf{let}\ B_1; \vec{x} = \vec{M}\ [\vec{T}/\vec{\alpha}]\ \textbf{in}\ x_i, \\ \quad \textbf{let}\ B_2; \vec{x} = \vec{M}\ [\vec{T}'/\vec{\alpha}]\ \textbf{in}\ x_i) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T_i)\end{array}$$

because each term takes exactly one Lookup step to replace x_i with $M_i\ [\vec{T}/\vec{\alpha}]$, and the extra bindings serve no purpose beyond this. These, combined with the inductive hypothesis of $\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash M'\ \Delta\ M'\ :\ T$, imply

$$\begin{array}{l}(\textbf{let}\ B_1; \vec{x} = \vec{M}\ [\vec{T}/\vec{\alpha}]\ \textbf{in}\ M'\ [\vec{T}/\vec{\alpha}], \\ \quad \textbf{let}\ B_2; \vec{x} = \vec{M}\ [\vec{T}'/\vec{\alpha}]\ \textbf{in}\ M'\ [\vec{T}'/\vec{\alpha}]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T)\end{array}$$

which implies

$$\begin{array}{l}(\textbf{let}\ B_1\ \textbf{in let}\ \vec{x} = \vec{M}\ [\vec{T}/\vec{\alpha}]\ \textbf{in}\ M'\ [\vec{T}/\vec{\alpha}], \\ \quad \textbf{let}\ B_2\ \textbf{in let}\ \vec{x} = \vec{M}\ [\vec{T}'/\vec{\alpha}]\ \textbf{in}\ M'\ [\vec{T}'/\vec{\alpha}]) \in \Delta\ (\vec{R}/\vec{\alpha})\ (T)\end{array}$$

Because these are each cost-equivalent to the terms in the previous relation and machine-closed relations respect cost-equivalence. Finally, we note that this reasoning was generic in $\vec{R}$, B_1 and B_2, so we can conclude $\Gamma \vdash \mathbf{let}\ \vec{x} = \vec{M}\ \mathbf{in}\ M' \mathrel{\Delta} \mathbf{let}\ \vec{x} = \vec{M}\ \mathbf{in}\ M' : T$.

This completes the proof. □