A Generic Foundation for Program Improvement

JENNIFER HACKETT and GRAHAM HUTTON, University of Nottingham, UK

Improvement theory is an approach to reasoning about program efficiency that allows inequational proofs of improvement to be conducted in a similar manner to equational proofs of correctness. Unfortunately, there is currently no unified framework for reasoning about efficiency using this approach, with each choice of language, semantics and cost model requiring its own custom theory. This article develops an approach based on metric spaces that is generic in all three of these aspects, allowing us to reason not only about time and space usage, but about any notion of resource. This theory also indicates why space efficiency has been problematic in the past, and shows us how to avoid these problems. We develop our theory and show how it can be used both to establish generic improvement results and to verify specific programming examples. This work establishes metric spaces as the “essence” of improvement theory.

CCS Concepts: • Theory of computation → Operational semantics; Denotational semantics; • Mathematics of computing → Point-set topology;

ACM Reference Format:

1 INTRODUCTION

Declarative programming is about writing programs in terms of what they are, instead of what they do. This has the benefit of conceptual simplicity, making it easier to build programs out of smaller parts in a compositional style. However, if we ever want to run our programs, what they do will matter at some point. It is here that the conceptual purity of the declarative approach can become an impediment, obscuring the operational behaviour of our programs. We would like to be able to use the same sorts of techniques to reason about efficiency as we do about correctness.

One approach to addressing these issues is improvement theory [Sands 1991]. This is an inequational theory, which matches well with the equational character of proofs of correctness. However, improvement theory’s main drawback is that it is tied to specific languages, semantics and cost models, requiring a new theory for any new combination of these factors. The most well-developed improvement theory is for time efficiency of call-by-need functional languages [Moran and Sands 1999a], but there is also a theory addressing space in the same setting [Gustavsson and Sands 1999, 2001], as well as theories for other evaluation strategies. For the most part, each new version of improvement theory has been built from scratch, making it difficult to see how to integrate different theories into a unified framework. In particular, there are powerful proof techniques available for reasoning about time that are not available in the space theory.

In this paper, we address this problem by introducing a new foundation for improvement theory based on metric spaces [Searcóid 2006]. This is inspired by the use of metric spaces in program semantics, in particular the metric model of PCF developed by Escardó [1999]. The key idea we
exploit is the same, namely that terms that take a larger number of resources to distinguish should be "closer" to each other than those that can be more easily separated. The result is a denotational semantics with enough operational information to capture improvement properties.

Our framework is highly generic, making minimal assumptions about the underlying language, semantics and cost model. Furthermore, it is quite intuitive and straightforward to use, despite the use of the perhaps unfamiliar concept of metric spaces. We demonstrate the utility of our theory by showing how it can be used both to establish generic improvement results, and to verify specific programming examples. No prior knowledge of improvement theory or metric spaces is assumed, and we begin with a brief review of these two concepts.

2 IMPROVEMENT THEORY

Improvement theory refers to a class of inequational theories that can be used to reason about intensional properties of programs, most importantly their resource usage. We say a program term $M$ is "improved" by another term $N$, written $M \triangleright N$, if $N$ is "better" (in some sense) than $M$ in any program context. Formally, we define the relation $\triangleright$ as follows:

$$M \triangleright N \iff \forall C. \llbracket M \rrbracket \downarrow^\leq n \implies \llbracket N \rrbracket \downarrow^\leq n$$

That is, "for any context $C$, if $C[M]$ terminates with cost $n$, then $C[N]$ will terminate with cost at most $n". This notion of cost can refer to the number of steps of evaluation (time cost), the amount of stack and/or heap used (space cost), or more generally any form of resource. We also have the symmetric notion of "cost equivalence", written $M \lhd N$, defined by $M \triangleright N \land N \triangleright M$.

The original motivation for improvement theory was the problem of reasoning about the time cost of non-strict functional programs [Sands 1991]. In this setting, simple step-counting is inadequate, because a non-strict evaluation strategy only evaluates those parts of the output that are required by the surrounding context. This leads to the idea of quantifying over all contexts. However, improvement theory also has applications to program correctness, as it can be used to justify the total correctness of transformations on recursive programs [Sands 1995].

Improvement theory was initially developed in the 1990s, but has recently been the subject of renewed interest and progress, with general-purpose program optimisations such as the worker/wrapper transformation [Hackett and Hutton 2014], common subexpression elimination [Schmidt-Schauß and Sabel 2015] and short cut fusion [Hackett and Hutton 2018] being formally shown to be improvements. Other recent work includes the development of tool support for improvement proofs [Handley and Hutton 2018], a categorical view of improvement theory based on preorder-enriched categories [Hackett and Hutton 2015] and an operational theory of parametricity that gives "improvements for free" [Hackett and Hutton 2018].

3 METRIC SPACES

Metric spaces generalise ordinary Euclidean spaces to an abstract notion of sets equipped with a distance measure. Formally, a metric on a set $X$ is a function $d : X \times X \rightarrow \mathbb{R}^+$ (where $\mathbb{R}^+$ is the non-negative real numbers) satisfying the following properties [Searcóid 2006]:

$$d(x, y) = 0 \iff x = y \quad \text{(identity of indiscernibles)}$$
$$d(x, y) = d(y, x) \quad \text{(symmetry)}$$
$$d(x, z) \leq d(x, y) + d(y, z) \quad \text{(triangle inequality)}$$

Given this definition, a metric space is simply a pair $(X, d)$ where $d$ is a metric on $X$. Sometimes we will require the strong triangle inequality, $d(x, z) \leq \max\{d(x, y), d(y, z)\}$. In this case, we say
that \( d \) is an ultrametric and \((X, d)\) is an ultrametric space. We will refer to \( X \) on its own as a metric or ultrametric space when our choice of the function \( d \) is apparent.

Metric spaces allow us to generalise the usual notion of limit to arbitrary sets. A sequence of values \( x_1, x_2, x_3 \ldots \) in a metric space \( X \) is called a Cauchy sequence if the values get arbitrarily close together, i.e. if for any \( \epsilon > 0 \) there is some natural number \( N \) such that for any \( m \geq n \geq N \), \( d(x_m, x_n) \leq \epsilon \). In this case, if there is an \( x \) that the \( x_i \) get arbitrarily close to, we say that \( x \) is the limit of the sequence. Note that any sequence with a limit will be a Cauchy sequence, as if the terms get arbitrarily close to some fixed value then they will also get arbitrarily close to each other. A metric space where all Cauchy sequences have limits is called a complete metric space.

There are multiple possible notions of morphisms between metric spaces. In this work, we focus on two in particular: non-expansive maps and contractions. A non-expansive map between metric spaces \( X \) and \( Y \) is a function \( f : X \rightarrow Y \) such that for any \( x_1, x_2 \in X \), we have \( d(f(x_1), f(x_2)) \leq d(x_1, x_2) \). Non-expansive maps are closed under composition, and the identity function is trivially non-expansive, so metric spaces with non-expansive maps form a category.

In turn, a function \( f : X \rightarrow Y \) is a contraction if there is a constant \( 0 < c < 1 \) such that for any \( x_1, x_2 \in X \), we have \( d(f(x_1), f(x_2)) \leq c \cdot d(x_1, x_2) \). Clearly, any contraction is also a non-expansive map, but there are non-expansive maps that are not contractions, in particular the identity function. Therefore, contractions do not form a category. However, they do have the important property that in a complete metric space, all contractions have unique fixed points. This fact is known as the Banach fixed point theorem, after its originator [Banach 1922].

Metric spaces can be used to give a denotational semantics of programs, with the advantage that a program’s meaning can be defined uniquely as the fixed point of a set of recursive equations. This contrasts with the competing approach of domain theory based upon lattices or complete partial orders, in which the meaning of a program is the least solution to its defining equations. A key way in which domain theory differs from metric space semantics is that it lacks the uniqueness property that we obtain from the Banach fixed point theorem. However, the domain-theoretic ordering on programs often corresponds to some natural ordering by definedness. In this article our semantic domain is both a metric space and a lattice, and we apply techniques from both approaches.

4 A METRIC ON PROGRAMS

We can use the notion of “improvement in all contexts” to define a metric on the interpretations of programs. The advantage of this approach is that we can be generic in our notion of cost, requiring only some abstract “cost function” rather than any concrete semantics.

In order to realise this idea, let \( \text{Trm} \) be the set of valid terms in a programming language, \( \text{Ctx} \) be the set of program contexts (terms with holes), and let us write \( M \Downarrow^n \) if evaluation of the term \( M \) terminates with cost \( n \in \mathbb{N} \). In turn, we define a function cost : \( \text{Trm} \rightarrow \mathbb{N}_\omega \) (where \( \mathbb{N}_\omega = \mathbb{N} \cup \{\omega\} \)) that assigns to every terminating term the cost of evaluating it, and to every non-terminating term the symbol \( \omega \) that represents a value greater than any finite cost. That is, we have:

\[
\text{cost}(M) = \begin{cases} 
  n & \text{if } M \Downarrow^n \\
  \omega & \text{if } M \text{ diverges}
\end{cases}
\]

We interpret terms in the semantic domain \( \text{Ctx} \rightarrow \mathbb{N}_\omega \), which forms a complete lattice under \( \leq^* \), the pointwise lifting of \( \leq \) on \( \mathbb{N}_\omega \), and define our interpretation \([\cdot] : \text{Trm} \rightarrow (\text{Ctx} \rightarrow \mathbb{N}_\omega)\) by simply taking the cost of evaluating the given term in the supplied context:

\[
\lceil M \rceil(C) = \text{cost}(C[M])
\]

Note that there are values in the semantic domain that do not correspond to any term in the source language, such as the constant function that always returns the cost zero. However, these
impossible values are harmless, existing only to ensure the completeness of the lattice. Using these definitions, it is straightforward to show that $M \succeq N$ if and only if $[N] \leq^* [M]$:

\[
\begin{align*}
[N] &\leq^* [M] \\
&\iff \{\text{definition of } \leq^*\} \\
&\forall C. [N](C) \leq [M](C) \\
&\iff \{\text{definition of } [-]\} \\
&\forall C. \text{cost}(C[N]) \leq \text{cost}(C[M]) \\
&\iff \{\text{definition of cost, properties of } \leq\} \\
&\forall C. C[M] \uplus^n \Rightarrow C[N] \uplus^\leq^n \\
&\iff \{\text{definition of } \succ\} \\
M &\succeq N
\end{align*}
\]

Similarly, we can also show that $M \preceq N$ if and only if $[M] = [N]$.

We can now define a metric on the interpretation of terms. Let $r \in \mathbb{R}$ be an arbitrary but fixed number $0 < r < 1$. We define a distance function $d$ on the set $\text{Ctx} \to \mathbb{N}_\omega$ as follows:

\[
d(p, q) = \sup \{ \min\{\rho(C), \omega(q(C))\} \mid C \in \text{Ctx}, \rho(C) \neq q(C)\}
\]

(Note that we define $r^0 = 0$.) This definition expresses that to find the distance between two terms, we first consider all contexts where their behaviours differ in terms of cost. The smaller of the two costs is the amount of resources required to distinguish these behaviours. We then exponentiate with base $r$ to convert this value to a distance between 0 and 1, with greater costs being mapped to smaller distances. Finally, we take the supremum of all of these distances, as the greatest distance will be from the context that requires the fewest resources to distinguish the terms.

In this way, we capture the idea due to Escardó [1999] that the greater the cost required to distinguish terms, the closer they are. It is straightforward to show that under the above definition our semantic domain forms a complete metric space, with limits computed pointwise.

Each context $C$ induces a map $\lbrack C\rbrack$ on the set $\text{Ctx} \to \mathbb{N}_\omega$, defined by $\lbrack C\rbrack(p) = \lambda D. \rho(D[C[-]])$. A simple calculation shows that this map is non-expansive:

\[
d([C](p), [C](q)) = \{\text{definitions of } d, [C]\} \\
= \sup \{ \min\{\rho(D[C[-]], \omega(D[C[-])]\} \mid D \in \text{Ctx}, \rho(D[C[-]]) \neq q(D[C[-]])\} \\
= \{\text{letting } C' = D[C[-]]\} \\
= \sup \{ \min\{\rho(C'), \omega(q(C'))\} \mid C' \in \text{Ctx}, \exists D. C' = D[C[-]], \rho(C') \neq q(C')\} \\
\leq \{\text{sup is monotone increasing}\} \\
= \{\text{renaming } C' \text{ to } C\} \\
= \sup \{ \min\{\rho(C), \omega(q(C))\} \mid C \in \text{Ctx}, \rho(C) \neq q(C)\} \\
= \{\text{definition of } d\} \\
d(p, q)
\]

The map $\lbrack C\rbrack$ is the denotational analogue of the context $C$, in that $\lbrack C\rbrack([M]) = [C[M]]$. 

Jennifer Hackett and Graham Hutton
A Generic Foundation for Program Improvement

Finally, there is a special function $\delta$ on $\text{Ctx} \rightarrow \mathbb{N}_\omega$ that adds precisely one unit of cost in all contexts, defined by $\delta(p) = \lambda C. p(C) + 1$. We can show that this function is a contraction:

$$d(\delta(p), \delta(q))$$

$$= \begin{cases} 
\text{definitions of } d, \delta \\
\sup\{r \min\{p(C) + 1, q(C) + 1\} | C \in \text{Ctx}, \ p(C) + 1 \neq q(C) + 1\} \\
\text{arithmetic} \\
\sup\{r \cdot r \min\{p(C), q(C)\} | C \in \text{Ctx}, \ p(C) \neq q(C)\} \\
\{\text{definition of } d\} \\
r \cdot d(p, q)
\end{cases}$$

This function plays the same role as the $\surd$ operator of improvement theory.

5 IMPROVING WITH METRICS

The metric-based approach allows us to prove theorems relating to improvement in a generic way. In this section, we prove a number of such theorems, illustrating the utility of our framework.

5.1 Syntactic Continuity

In operational semantics, it is generally useful to have some theorems that characterise the behaviour of a recursive term in terms of a sequence of finite unwindings. This kind of theorem is often called “syntactic continuity”, being a syntactic analogue of the domain-theoretic concept of continuity. For example, for a term $\text{let } x = M \text{ in } N$ we might have the following sequence of unwindings:

\[
\begin{align*}
\text{let } x_0 = \bot & \text{ in } N[x_0/x] \\
\text{let } x_0 = \bot; x_1 = M[x_0/x] & \text{ in } N[x_1/x] \\
\text{let } x_0 = \bot; x_1 = M[x_0/x]; x_2 = M[x_1/x] & \text{ in } N[x_2/x] \\
& \vdots
\end{align*}
\]

The symbol $\bot$ refers to some suitable bottom element to start the chain.

From now on, we will use the terms $M_0, M_1, M_2, \ldots$ to refer to the sequence of unwindings of some term $M$. In the setting of our metric space approach, a continuity theorem is a theorem that states that $[M_0], [M_1], [M_2], \ldots$ is a Cauchy sequence with limit $[M]$. The precise definition of the sequence $M_n$ and the class of terms $M$ that the theorem holds for will vary depending on the particular language and the notion of resource that is being considered.

For the case of time costs, a continuity theorem will hold for terms of the form $\text{let } x = M \text{ in } N$ with the unwinding sequence given above. That the interpretations form a Cauchy sequence follows from the fact that each successive term will run for longer and terminate in more contexts, so the distances will continually decrease, trending towards zero. That the limit of this sequence is $[\text{let } x = M \text{ in } N]$ follows from the fact that for any context $C$, we have

$$\text{cost}(C[\text{let } x = M \text{ in } N]) = \lim_{n \to \infty} \text{cost}(C[\text{let } x = M \text{ in } N])$$
where \( \text{let } x^n = M \text{ in } N \) is the \( n \)th element of the sequence of unwindings. This is because only finitely many levels of recursion will be used in any given execution.

The above reasoning is valid in a call-by-name setting, but in a call-by-need setting it loses sharing between the different levels of recursion. To make it valid for call-by-need, we can either restrict \( M \) to be a value, or modify our definition of unwindings to a notion of "bindings with fuel", where the fuel (cf. Haynes and Friedman [1984]) is an integer annotation on bindings that is decremented when the bound variable is looked up. Such annotations put an upper bound on the recursion depth in the same way as unwinding, but without losing sharing.

The space theory of Gustavsson and Sands [1999, 2001] has a syntactic continuity theorem, but it is not a continuity theorem in our sense. In particular, while it does characterise terms of the form \( \text{let } f = V \text{ in } C[f] \) as limits of their sequences of finite unwindings, it is not clear that the sequence of unwindings is Cauchy when our cost function measures space rather than time. The reason is that while it takes more time to distinguish successive unwindings of a recursive binding, if the recursive call is a tail call or occurs within a thunk, then the space used could be the same. However, if the recursive call in the binding \( f = V \) is scrutinised by a case expression then each successive unwinding will push information about the context of the recursive call onto the stack, increasing the space usage. Therefore, continuity holds for this subset of recursive bindings.

Having made this observation, we can see that if we re-incorporate time into our cost function we can recover continuity for all recursive bindings. If we have some monotone function \( f : \mathbb{N}_\omega \times \mathbb{N}_\omega \rightarrow \mathbb{N}_\omega \) such that \( f(t, s) < f(t + 1, s) \) for any \( t \) and \( s \), we can define

\[
\text{cost}(M) = f(\text{cost}_{\text{time}}(M), \text{cost}_{\text{space}}(M))
\]

where \( \text{cost}_{\text{time}} \) and \( \text{cost}_{\text{space}} \) are our cost functions for time and space respectively, and the resulting theory will have continuity for the same set of terms as the theory that considers time alone (as any Cauchy sequence for the time theory will remain Cauchy in the combined theory). To put it into words, we can combine time and space costs in nearly any way that we please, but we cannot ignore time altogether if we wish to retain continuity.

5.2 Unwindings and Improvement

If a continuity theorem holds for our particular choice of cost model, we can use the unwindings of two terms to prove an improvement between them. Suppose that we have two terms \( M \) and \( N \) for which a continuity theorem holds, and let \( M_0, M_1, M_2, \ldots \) and \( N_0, N_1, N_2, \ldots \) be their sequences of unwindings. Then the following inference rule is valid:

\[
\exists k. \forall n \geq k. M_n \triangleright N_n \\
M \triangleright N
\]

This result follows from the well-known fact that standard numeric limits preserve (non-strict) inequalities. First of all, let \( \mathbb{C} \) be an arbitrary context. Then because \( M \) is the limit of the sequence of \( M_n \), we have \( \lim_{n \rightarrow \infty} [M_n]\,(\mathbb{C}) = [M](\mathbb{C}) \), and similarly for \( N \). Therefore, \( [N](\mathbb{C}) \leq [M](\mathbb{C}) \), because \( [N_n]\,(\mathbb{C}) \leq [M_n]\,(\mathbb{C}) \) for all \( n \geq k \). The above reasoning was generic in our choice of \( \mathbb{C} \), so we can conclude \( [N] \leq^* [M] \) and hence that we have \( M \triangleright N \).

A corollary of this result is an analogue to the well-known \textit{squeeze theorem} for limits of sequences. Given three terms \( M, N \) and \( P \) with unwindings \( M_n, N_n \) and \( P_n \) respectively, we have:

\[
\exists k. \forall n \geq k. M_n \triangleright N_n \triangleright P_n \\
M \triangleright N \triangleright P
\]

That is, if the unwindings of \( N \) are always between the unwindings of \( M \) and \( P \), and \( M \) and \( P \) are cost equivalent, then \( N \) must also be cost equivalent to \( M \) and \( P \).
5.3 The Improvement Theorem

The following rule, due to Moran and Sands [1999a], is known as the improvement theorem:

\[
\text{let } f = V \text{ in } V \vdash \text{let } f = V \text{ in } W \\
\text{let } f = V \text{ in } N \vdash \text{let } f = W \text{ in } N
\]

Using our metric approach, we can give a proof of the validity of this inference rule for call-by-need time improvements. Consider the following sequence of terms:

\[
\text{let } f_0 = V[f_0/f] \text{ in } N[f_0/f] \\
\text{let } f_0 = V[f_0/f]; f_1 = W[f_0/f] \text{ in } N[f_1/f] \\
\text{let } f_0 = V[f_0/f]; f_1 = W[f_0/f]; f_2 = W[f_1/f] \text{ in } N[f_2/f] \\
\vdots
\]

The interpretations of these terms form a Cauchy sequence for the same reason that the sequence starting at \( \bot \) is Cauchy. The initial element of the sequence is cost equivalent to \( \text{let } f = V \text{ in } N \).

Furthermore, the limit of the sequence will be \([\text{let } f = W \text{ in } N]\), as each term in the sequence will take more levels of recursion than the previous to diverge from this. Therefore, it suffices to show that this sequence is a chain, as the limit of a chain will be an upper bound to all the elements and thus greater than the initial element. We calculate as follows:

\[
\text{let } f_0 = V[f_0/f]; f_1 = W[f_0/f]; \ldots; f_n = W[f_{n-1}/f] \text{ in } N[f_n/f] \\
\vdash \{\text{unrolling one level}\} \\
\text{let } f_{n-1} = V[f_{n-1}/f]; f_0 = V[f_{n-1}/f]; f_1 = W[f_0/f]; \ldots; f_n = W[f_{n-1}/f] \text{ in } N[f_n/f] \\
\vdash \{\text{let } f = V \text{ in } V \vdash \text{let } f = V \text{ in } W\} \\
\text{let } f_{n-1} = V[f_{n-1}/f]; f_0 = W[f_{n-1}/f]; f_1 = W[f_0/f]; \ldots; f_n = W[f_{n-1}/f] \text{ in } N[f_n/f] \\
\vdash \{\text{renaming}\} \\
\text{let } f_0 = V[f_0/f]; f_1 = W[f_0/f]; \ldots; f_{n+1} = W[f_n/f] \text{ in } N[f_{n+1}/f]
\]

Thus, we see that the improvement theorem comes elegantly out of our metric space framework. This idea of constructing chains of improvements is a highly versatile proof technique.

5.4 Improvement Induction

The improvement induction principle of Moran and Sands [1999a] is a powerful technique that allows us to show an improvement between two terms by showing that they are both in some sense “fixed points” of a particular context. Letting \( \checkmark \) be a context such that \([\checkmark] = \delta\), improvement induction is formally captured by the following inference rule:

\[
M \vdash \checkmark \cap [M] \quad \checkmark \cap [N] \nleq N \quad \frac{}{M \nleq N}
\]

To prove this rule, we formalise the idea of fixed points using the machinery of metric spaces. First, we note that the second precondition implies that \([N]\) is the fixed point of \( \delta \cdot [\checkmark] \). Because \( \delta \) is a contraction and \([\checkmark]\) is non-expansive, their composition is also a contraction and so by the Banach fixed point theorem \([N]\) is the only such fixed point. Because our domain is a complete lattice, this will also be the least pre-fixed point. But the first precondition implies that \([M]\) is a pre-fixed point of \( \delta \cdot [\checkmark] \), and hence it must be greater than the least pre-fixed point \([N]\), so we can conclude that \([N] \leq^* [M]\). As noted earlier, this implies \(M \nleq N\), which completes the proof.
In fact, we can make improvement induction more general, by replacing the use of cost equivalence in the second precondition with simple cost improvement:

\[ M \triangleright \sqrt{C}[M] \quad \sqrt{C}[N] \triangleright N \]

In this case, \([N]\) is no longer the unique fixed point, only a post-fixed point. However, a unique fixed point in a complete lattice is also the greatest post-fixed point, so the result follows from a simple modification of the above proof and the transitive property of \(\triangleright\).

The space improvement theory of Gustavsson and Sands [1999, 2001] lacked this kind of induction rule. Our proof here can provide an intuition as to why: none of the so-called “space gadgets” of that theory would correspond to our function \(\delta\). This is because the increase in space usage they cause is only temporary, allowing the space used to be re-used in certain contexts. Therefore, if we wish to use improvement induction in the context of space efficiency, we need to add some operation that permanently takes up space, increasing space usage by one in any context.

6 EXAMPLES

In this section we illustrate the practical application of our metric-based approach to improvement. We begin by showing how our generic theory can be instantiated to verify some well-known improvement results, before going on to show how it can be used to establish new improvement results that go beyond what was possible with the previous theories.

6.1 Associativity of Append

Consider the following recursive definition for the append operator on lists:

\[
\begin{align*}
[] & \mathbin{++} ys = ys \\
(x : xs) & \mathbin{++} ys = x : (xs \mathbin{++} ys)
\end{align*}
\]

Using this definition, a left-associated append \((xs \mathbin{++} ys) \mathbin{++} zs\) will intuitively take more steps to evaluate than a right-associated append \(xs \mathbin{++} (ys \mathbin{++} zs)\). This behaviour arises from the fact that \(\mathbin{++}\) is defined by traversing its first argument, which means that in the left-associated case the list \(xs\) is traversed twice, whereas in the right-associated case it is only traversed once.

We can formally verify this result using improvement induction. The proof proceeds in essentially the same manner as using regular improvement theory, and demonstrates how our new, generic theory can be used in the same way as the original, specific version. In this case, the cost function counts the number of heap lookup steps in a suitable abstract machine such as [Sestoft 1997], which is a faithful measure of the total number of evaluation steps [Moran and Sands 1999a].

To verify the improvement \((xs \mathbin{++} ys) \mathbin{++} zs \triangleright xs \mathbin{++} (ys \mathbin{++} zs)\), we begin by defining the following context, in which \([-\]) denotes the hole where a term will be substituted:

\[
\begin{align*}
C &= \text{case } xs \text{ of} \\
[] & \rightarrow ys \mathbin{++} zs \\
x : xs & \rightarrow x : [-]
\end{align*}
\]

By the definition of append, \(xs \mathbin{++} (ys \mathbin{++} zs)\) is cost equivalent to \(\sqrt{C}[xs \mathbin{++} (ys \mathbin{++} zs)]\). Hence, by improvement induction, it suffices to show that \((xs \mathbin{++} ys) \mathbin{++} zs \triangleright \sqrt{C}[xs \mathbin{++} ys] \mathbin{++} zs\), which can itself be verified by the following calculation:

\[
\begin{align*}
(xs \mathbin{++} ys) \mathbin{++} zs \\
\triangleright \quad \{\text{definition of append}\} \\
\sqrt{\text{case } xs ++ ys \text{ of}}
\end{align*}
\]
A Generic Foundation for Program Improvement

\[
\begin{align*}
[ & \to zs \\
      rs & \to r : (rs ++ zs) \\
\end{align*}
\]

\[
\left\{ \text{definition of append} \right\}
\]

\[
\begin{align*}
\checkmark \text{case } & \checkmark \text{ case } xs \text{ of} \\
[ & \to ys \\
      x : xs & \to x : (xs ++ ys) \\
\end{align*}
\]

\[
\begin{align*}
\checkmark \text{ case } & \checkmark \text{ case } xs \text{ of} \\
[ & \to zr \\
      r : rs & \to r : (rs ++ zs) \\
\end{align*}
\]

\[
\left\{ \text{pushing tick inward} \right\}
\]

\[
\begin{align*}
\checkmark \text{ case } & \checkmark \text{ case } xs \text{ of} \\
[ & \to \checkmark ys \\
      x : xs & \to \checkmark x : (xs ++ ys) \\
\end{align*}
\]

\[
\begin{align*}
\checkmark \text{ case } & \checkmark \text{ case } xs \text{ of} \\
[ & \to zr \\
      r : rs & \to r : (rs ++ zs) \\
\end{align*}
\]

\[
\left\{ \text{strict contexts distribute over case} \right\}
\]

\[
\begin{align*}
\checkmark \text{ case } & \checkmark \text{ case } xs \text{ of} \\
[ & \to zr \\
      r : rs & \to r : (rs ++ zs) \\
\end{align*}
\]

\[
\left\{ \text{eliminating a tick} \right\}
\]

\[
\begin{align*}
\checkmark \text{ case } & \checkmark \text{ case } xs \text{ of} \\
[ & \to zr \\
      r : rs & \to r : (rs ++ zs) \\
\end{align*}
\]

\[
\left\{ \text{case of known constructor} \right\}
\]

\[
\begin{align*}
\checkmark \text{ case } & \checkmark \text{ case } xs \text{ of} \\
[ & \to zr \\
      r : rs & \to r : (rs ++ zs) \\
\end{align*}
\]

\[
\left\{ \text{pulling tick out of strict context} \right\}
\]

\[
\begin{align*}
\checkmark \text{ case } & \checkmark \text{ case } xs \text{ of} \\
[ & \to zr \\
      r : rs & \to r : (rs ++ zs) \\
\end{align*}
\]
Thus we have shown that associativity of append is indeed a time improvement.

### 6.2 Fast Reverse

Consider a simple recursive definition for the \textit{reverse} function on lists,

\[
\begin{align*}
\text{reverse} \; [] &= [] \\
\text{reverse} \; (x : xs) &= \text{reverse} \; xs ++ [x]
\end{align*}
\]

Together with a more efficient version \textit{reverse'} that uses an accumulator:

\[
\begin{align*}
\text{reverse'} \; xs &= \text{revcat} \; xs \; [] \\
\text{where} \\
\text{revcat} \; [] \; ys &= ys \\
\text{revcat} \; (x : xs) \; ys &= \text{revcat} \; xs \; (x : ys)
\end{align*}
\]

In order to formally relate the time performance of these two functions, we begin by showing that \textit{reverse} \; \textit{xs} ++ \textit{ys} \triangleright \textit{revcat} \; \textit{xs} \; \textit{ys}, which establishes the relationship between \textit{reverse} and the auxiliary function \textit{revcat}. To this end, we define the following context:

\[
C = \text{case} \; xs \; \text{of} \\
[] \rightarrow ys \\
x : xs \rightarrow \text{let} \; ys = x : ys \; \text{in} \; [-]
\]

(The let binding here is non-recursive, ensuring that \textit{ys} is replaced by \(x : ys\) in the hole.) By the definition of \textit{revcat}, we have \(\checkmark C[\textit{revcat} \; \textit{xs} \; \textit{ys}] \triangleright \textit{revcat} \; \textit{xs} \; \textit{ys}\), where the \(\checkmark\) pays for the step of applying the definition. Hence, by the generalised form of improvement induction, it suffices to show \(\text{reverse} \; \textit{xs} ++ \textit{ys} \triangleright \checkmark C[\textit{reverse} \; \textit{xs} ++ \textit{ys}]\), which proceeds as follows:

\[
\begin{align*}
\text{reverse} \; \textit{xs} ++ \textit{ys} \\
\triangleright \{\text{definition of reverse}\} \\
\checkmark\text{(case} \; \textit{xs} \; \text{of} \\
[] \rightarrow [] \\
x : xs \rightarrow \text{reverse} \; \textit{xs} ++ [x]) \; ++ \textit{ys} \\
\triangleright \{\text{strict operations distribute over case}\} \\
\checkmark\text{(case} \; \textit{xs} \; \text{of} \\
[] \rightarrow [] \; ++ \textit{ys} \\
x : xs \rightarrow (\text{reverse} \; \textit{xs} ++ [x]) \; ++ \textit{ys} \\
\triangleright \{\text{associativity of append is an improvement}\} \\
\checkmark\text{(case} \; \textit{xs} \; \text{of} \\
[] \rightarrow [] \; ++ \textit{ys} \\
x : xs \rightarrow \text{reverse} \; \textit{xs} ++ ([x] \; ++ \textit{ys}) \\
\triangleright \{\text{evaluating and renaming}\}
\end{align*}
\]
\( \begin{align*}
\text{✓ case } & \text{xs of} \\
[] & \rightarrow ys \\
x : \text{xs} & \rightarrow \text{let } ys = x : ys \text{ in } \text{reverse xs } ++ ys \\
\equiv & \{ \text{definition of context} \} \\
\text{✓ C}[\text{reverse } \text{xs } ++ ys]
\end{align*} \)

Thus we have shown that \( \text{reverse } \text{xs } ++ ys \triangleright \text{revcat } \text{xs } yss \). Finally, if we take \( yss = [] \), this result implies that \( \text{reverse } \text{xs } ++ [] \) is improved by \( \text{reverse’ } \text{xs} \), which means that the performance of \( \text{reverse’ } \text{xs} \) can be no worse than that of \( \text{reverse } \text{xs} \) plus the cost of \( ++ [] \), which is linear. Note that we do not have \( \text{reverse } \text{xs } \triangleright \text{reverse’ } \text{xs} \), because this result is invalid for small lists (empty and singletons) due to the overhead of calling the auxiliary function \( \text{revcat} \).

### 6.3 Compact Reverse

The fast version of the \( \text{reverse} \) function also has another advantage: it uses less space. However, we cannot formally verify this in the same way as we did for time, for two reasons. First of all, associativity of append is not a space improvement [Gustavsson and Sands 2001]. And secondly, our proof used the improvement induction principle, which does not hold for space costs.

Instead, we will construct a chain of unfoldings, starting with the more efficient \( \text{reverse’ } \) and ending with the less efficient \( \text{reverse} \). In this case, the cost function measures the space (in terms of heap and stack usage) required to evaluate a term. Our chain of unfoldings is then as follows:

\[
\begin{align*}
M_0 &= \text{reverse’ } \text{xs} \\
M_1 &= \text{case } \text{xs } \text{of} \\
&[\phantom{} ] \rightarrow [\phantom{} ] \\
x : \text{xs}’ &\rightarrow \text{reverse’ } \text{xs}’ ++ [x] \\
M_2 &= \text{case } \text{xs } \text{of} \\
&[\phantom{} ] \rightarrow [\phantom{} ] \\
x : \text{xs}’ &\rightarrow (\text{case } \text{xs}’ \text{ of} \\
&[\phantom{} ] \rightarrow [\phantom{} ] \\
x’ : \text{xs}” &\rightarrow \text{reverse’ } \text{xs}” ++ [x’]) ++ [x] \\
\vdots
\end{align*}
\]

This sequence is Cauchy, because each level of nested append will push more information onto the stack, and the limit of the interpretations of the sequence is \([\text{reverse } \text{xs}]\), because each subsequent term will mimic the space behaviour of \( \text{reverse} \) for one more level of recursion. Therefore, it suffices to show that the sequence forms a chain of reversed improvements, i.e. \( M_{n+1} \triangleright M_n \) for all \( n \geq 0 \). The ordering is reversed because the improved version \( \text{reverse’ } \) is at the start of the chain.

This result can be proved by induction on \( n \). The inductive case follows from the fact that improvement is preserved by contexts, and it then remains to verify the base case, \( M_1 \triangleright M_0 \). Our proof of the base case makes use of a lemma, namely that \( \text{revcat } \text{xs } []++[x] \triangleright \text{revcat } \text{xs } [x] \). We use the following auxiliary definition:

\[
\text{revcat’ } x \text{ xs } yss = \text{case } \text{xs } \text{of} \\
&[\phantom{} ] \rightarrow [\phantom{} ] ++ [x] \\
x’ : \text{xs}’ &\rightarrow \text{revcat’ } x \text{ xs’ } (x’ : yss)
\]
Note that \( \text{revcat} \; xs \; [] \; ++ \; [x] \) is cost-equivalent to \( \text{revcat}' \; x \; xs \; [] \), as unrolling \( \text{revcat} \; xs \; [] \) and distributing \( ++ \; [x] \) over the cases will result in the unrollings of the latter.

Next, we consider the following chain of unfoldings:

\[
N_0 = \text{revcat} \; xs \; [x]
\]

\[
N_1 = \text{case} \; xs \; \text{of}
\]

\[
[\] \rightarrow \begin{array}{c}
[\] ++ [x] \\
x' : xs' \rightarrow \text{revcat} \; xs' \; ([x'] ++ [x])
\end{array}
\]

\[
N_2 = \text{case} \; xs \; \text{of}
\]

\[
[\] \rightarrow \begin{array}{c}
[\] ++ [x] \\
x' : xs' \rightarrow \text{case} \; xs' \; \text{of}
\end{array}
\]

\[
[\] \rightarrow [x] ++ [x]
\]

\[
x" : xs" \rightarrow \text{revcat} \; xs" \; ([x", x'] ++ [x])
\]

\[
\vdots
\]

This forms a reversed chain of improvements, starting at \( \text{revcat} \; xs \; [x] \). That this is a reversed chain follows from the fact that each term is simply an unrolling of the previous with some of the work of the appends undone. This implies that the distance between terms will keep decreasing, and hence the sequence is Cauchy. Finally, the limit is \( \text{revcat}' \; x \; xs \; [] \), as each successive term approximates it more closely, and this is cost-equivalent to \( \text{revcat} \; xs \; [] \; ++ \; [x] \). Hence, we can conclude that \( \text{revcat} \; xs \; [] \; ++ \; [x] \) is improved by \( \text{revcat} \; xs \; [x] \), as required.

Now we return to our proof obligation, \( M_1 \triangleright M_0 \). We reason as follows:

\[
M_1 \equiv \text{definition of } M_1
\]

\[
\text{case} \; xs \; \text{of}
\]

\[
[\] \rightarrow [\]
\]

\[
x : xs' \rightarrow \text{reverse}' \; xs' \; ++ \; [x]
\]

\[
\triangleright \text{definition of } \text{reverse}'
\]

\[
\text{case} \; xs \; \text{of}
\]

\[
[\] \rightarrow [\]
\]

\[
x : xs' \rightarrow \text{revcat} \; xs' \; [\] \; ++ \; [x]
\]

\[
\triangleright \text{above lemma}
\]

\[
\text{case} \; xs \; \text{of}
\]

\[
[\] \rightarrow [\]
\]

\[
x : xs' \rightarrow \text{revcat} \; xs' \; [x]
\]

\[
\triangleright \text{definition of } \text{revcat}
\]

\[
\text{revcat} \; xs \; []
\]

\[
\triangleright \text{definition of } \text{reverse}'
\]

\[
\text{reverse}' \; xs
\]

\[
\equiv \text{definition of } M_0
\]

\[
M_0
\]

Finally, because every step in the chain \( M_n \) is a reversed improvement, the first element of the chain \( \text{reverse}' \; xs \) must be an improvement of its limit \( \text{reverse} \; xs \), which establishes that the fast version of reverse is indeed a space improvement over the standard version.
6.4 Internal Nondeterminism

We can also use our framework to reason about resources other than space and time. One interesting example is internal nondeterminism. Many programs that produce deterministic results can naturally be specified using nondeterminism internally, but when it comes to implementation we would usually like to remove as much of the nondeterminism as possible. This kind of approach is often seen in the relational style of program calculation [Bird and de Moor 1997].

In order to apply our theory in this setting, we need a language and a cost model. We consider a simple functional language with two extra operators: choice and fail. The first operator nondeterministically “chooses” between one of two options, the second allows for the program to backtrack if it turns out one of the previous choices was incorrect, and our cost function will simply count the number of times that choice is invoked during evaluation.

Consider the following non-deterministic sorting program (sometimes called bogosort), which proceeds by randomly permuting the input list and checking if the result is ordered:

\[
\text{sort} = \text{checkOrdered} \cdot \text{permute}
\]

The function that permutes the input can itself be defined using an auxiliary function that nondeterministically selects a random element from a list using the choice operator:

\[
\text{permute}\,xs = \begin{cases}
null\,xs & \text{then } [] \\
\text{else} & \\
\text{let } (r,rs) = \text{selectRand}\,xs \text{ in } r : \text{permute}\,rs
\end{cases}
\]

\[
\text{selectRand}\,[x] = (x,[])
\]
\[
\text{selectRand}\,(x:xs) = \text{choice}\,(x,xs) (\text{let } (r,rs) = \text{selectRand}\,xs \text{ in } (r, x:rs))
\]

In turn, the function that checks if a list is ordered can be defined as follows, in which the use of the fail operator results in backtracking to another permutation if the check fails:

\[
\text{checkOrdered}\,[] = []
\]
\[
\text{checkOrdered}\,(x:xs) = \begin{cases}
\text{if } \text{all} (\geq x)\,xs \text{ then } x \text{ else fail}
\end{cases}
\]

While the above definition for sort has the desired behaviour, the "generate and test" approach that is used does not result in a practical sorting methodology. However, it can be regarded as a specification of a sorting program, and used as the basis for deriving a more practical implementation that removes the internal nondeterminism. In order to achieve this, we first fuse the two functions checkOrdered and permute together, calculating a version of sort that recurses directly:

\[
\text{sort}\,xs
\]

\[
\begin{array}{l}
\Rightarrow \{ \text{definition of sort} \}
\text{checkOrdered}\,(\text{permute}\,xs)
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow \{ \text{definition of permute} \}
\begin{array}{l}
\text{if } \text{null}\,xs \text{ then checkOrdered}\,[] \text{ else} \\
\text{let } (r,rs) = \text{selectRand}\,xs \text{ in } \text{checkOrdered}\,(r : \text{permute}\,rs)
\end{array}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow \{ \text{definition of checkOrdered} \}
\begin{array}{l}
\text{if } \text{null}\,xs \text{ then } [] \text{ else} \\
\text{let } (r,rs) = \text{selectRand}\,xs; \,rs' = \text{permute}\,rs \text{ in} \\
\text{if } \text{all} (\geq r)\,rs' \text{ then } r : \text{checkOrdered}\,rs' \text{ else } \text{fail}
\end{array}
\end{array}
\]

\[
\Rightarrow \{ \text{all is order-independent} \}
\text{if } \text{null}\,xs \text{ then } [] \text{ else}
\]

let \((r, rs) = \text{selectRand } xs\) \(rs' = \text{permute } rs\) in

\[
\text{if all } (\geq r) \text{ rs then } r : \text{checkOrdered } rs' \text{ else fail}
\]

\(\Leftrightarrow\) \{substituting \(rs'\)\}

\[
\text{if null } xs \text{ then } [] \text{ else}
\]

\[
\text{let } (r, rs) = \text{selectRand } xs \text{ in}
\]

\[
\text{if all } (\geq r) \text{ rs then } r : \text{checkOrdered } (\text{permute } rs) \text{ else fail}
\]

\(\Leftrightarrow\) \{definition of sort\}

\[
\text{if null } xs \text{ then } [] \text{ else}
\]

\[
\text{let } (r, rs) = \text{selectRand } xs \text{ in}
\]

\[
\text{if all } (\geq r) \text{ rs then } r : \text{sort } rs \text{ else fail}
\]

In summary, we have shown that \(\text{sort}\) is a pre-fixed point of the following context:

\[
C = \lambda x. \rightarrow \text{if null } xs \text{ then } [] \text{ else}
\]

\[
\text{let } (r, rs) = \text{selectRand } xs \text{ in}
\]

\[
\text{if all } (\geq r) \text{ rs then } r : [\ldots] \text{ rs else fail}
\]

That is, \(\text{sort} \not\succ C[\text{sort}]\). Moreover, due to the use of the nondeterministic function \(\text{selectRand}\), the interpretation of this context is a contraction, and hence it has a unique fixed point.

An implementation of sorting can now be characterised as a deterministic function \(f\) that improves the nondeterministic function \(\text{sort}\), i.e. for which \(\text{sort} \not\succ f\). However, because the context \(C\) has a unique fixed point, by a similar argument to that used for improvement induction it is sufficient to show that \(f\) is a post-fixed point of \(C\), i.e. that we have \(C[f] \not\succ f\).

We can use this proof obligation as a starting point to derive a program for \(\text{selection sort}\), based on a function \(\text{selectMin}\) that satisfies the following specification:

\[
\text{let } (r, rs) = \text{selectRand } xs \text{ in}
\]

\[
\text{if all } (\geq r) \text{ rs then } (r, rs) \text{ else fail}
\]

\(\succ\)

\(\text{selectMin } xs\)

This specification states that \(\text{selectMin}\) behaves in a similar manner to \(\text{selectRand}\), with the key difference that it will only return values \(\text{r}\) that are minimal in the input list, whereas \(\text{selectRand}\) returns any value from the input list. Based on such a function, we now proceed to derive our sorting function, starting from the proof obligation \(C[f] \not\succ f:\)

\[
\equiv \{\text{definition of } C\}
\]

\[
\lambda x. \rightarrow \text{if null } xs \text{ then } [] \text{ else}
\]

\[
\text{let } (r, rs) = \text{selectRand } xs \text{ in}
\]

\[
\text{if all } (\geq r) \text{ rs then } r : f \text{ rs else fail}
\]

\(\succ\)\{factoring out \(r : f \text{ rs}\)\}

\[
\lambda x. \rightarrow \text{if null } xs \text{ then } [] \text{ else}
\]

\[
\text{let } (r, rs) = \text{selectRand } xs \text{ in}
\]

\[
\text{let } (r', rs') = \text{if all } (\geq r) \text{ rs then } (r, rs) \text{ else fail}
\]

\[
\text{in } r' : f \text{ rs'}
\]

\(\Leftrightarrow\) \{rearranging lets\}

\[
\lambda x. \rightarrow \text{if null } xs \text{ then } [] \text{ else}
\]
let \((r', rs') = \text{let } (r, rs) = \text{selectRand } xs \text{ in }\)

\[
\begin{align*}
\text{if all } (\geq r) rs & \text{ then } (r, rs) \text{ else fail} \\
\text{ in } r' : f \text{ rs'}
\end{align*}
\]

\(\triangleright \{\text{specification of selectMin}\}\)

\(\lambda xs \rightarrow \text{if null } xs \text{ then } [] \text{ else }\)

\[
\text{let } (r', rs') = \text{selectMin } xs \text{ in } r' : f \text{ rs'}
\]

\(\triangleright \{\text{defining } f \text{ to be this term}\}\)

\(f\)

Finally, some simple renaming results in the following definition of selection sort:

\[
\text{ssort } xs = \text{if null } xs \text{ then } [] \text{ else } \text{let } (r, rs) = \text{selectMin } xs \text{ in } r : \text{ssort } rs
\]

Thus we have shown how our metric-based approach to program improvement can be used to derive selection sort from a nondeterministic specification of sorting.

6.5 Parallel Evaluation

As a final example, we illustrate how our framework can be used to reason about the parallel evaluation of terms. To this end, consider a simple type of arrays indexed by natural numbers, together with a linear fold operator \(\text{lfold}\) that applies an infix operator \(\oplus\) in a right-associated manner along the segment of an array \(a\) from an index \(i\) to an index \(j\), defined as follows:

\[
\text{lfold } (\oplus) i j a = \text{if } i == j \text{ then } a[i] \text{ else } a[i] \oplus \text{lfold } (\oplus) (i+i) j a
\]

For example, assuming that \([a, b, c, d]\) is an array with four elements and that array indexing starts from zero, we have \(\text{lfold } (\oplus) 0 3 [a, b, c, d] = a \oplus (b \oplus (c \oplus d))\).

In a similar manner, we can define a parallelisable fold operator \(\text{pfold}\) that splits the array segment into two parts in a divide-and-conquer style (using an integer division operator \(\div\) that rounds down to the nearest integer), resulting in a binary tree of applications:

\[
\text{pfold } (\oplus) i j a = \text{if } i == j \text{ then } a[i] \text{ else } \text{pfold } (\oplus) ((i+j) \div 2) a \oplus \text{pfold } (\oplus) (((i+j) \div 2) + 1) j a
\]

For example, using this definition we have \(\text{pfold } (\oplus) 0 3 [a, b, c, d] = (a \oplus b) \oplus (c \oplus d)\).

If the operator \(\oplus\) is associative, both fold operators defined above will produce the same final result, but with potentially different operational behaviours. In particular, if the operator \(\oplus\) is equally efficient regardless of how its applications are associated, then the \(\text{pfold}\) function is potentially better than \(\text{lfold}\), because it offers greater opportunities for parallel evaluation.

We can prove this property using improvement induction. In this case, the cost function will count the number of applications of the \(\oplus\) operator along the critical path, i.e. the longest sequence of data dependencies that can arise during evaluation. This models the expected running time for evaluating a term if we could evaluate arbitrarily many processes in parallel with no overheads. For the purpose of this example, we assume a strict evaluation order. We write \(\checkmark\) for an arbitrary operation that increases the cost of the critical path by one unit, so the improvement induction theorem will hold.

To verify that \(\text{lfold } \triangleright \text{pfold}\), we begin by defining a context based upon the body of the function \(\text{pfold}\), but with holes taking the place of the recursive calls:
\[ C = \lambda (\oplus) \ i \ j \ a \rightarrow \begin{cases} a[i] & \text{if } i == j \\ \lfold ((i + j) \div 2) a \oplus \lfold ((i + j) \div 2 + 1) j a & \text{else} \end{cases} \]

We then have \( pfold \not\preceq \sqrt{C}[pfold] \). Hence, by improvement induction it suffices to show that \( Ifold \not\preceq \sqrt{C}[Ifold] \). We begin by applying both sides to all their arguments, to give \( Ifold (\oplus) \ i \ j \ a \not\preceq \sqrt{C}[Ifold]\ i \ j \ a \). If this improvement holds so must the original, as the operational behaviour of a \( \lambda \)-abstraction is entirely determined by its behaviour when applied to arguments.

We now proceed by case analysis on \( m = j - i \), the difference between the two indices, noting that \((i + j) \div 2 = i + (m \div 2)\). For \( m = 0 \) and \( m = 1 \), it is easy to show that both sides of our improvement take the same number of steps. For \( m \geq 2 \), we calculate as follows:

\[
\begin{align*}
  \lfold & (\oplus) \ i \ j \ a \\
  \not\preceq & \quad \{\text{definition of } \lfold\} \\
  & \sqrt{a[i]} \oplus \lfold(\oplus)(i+1) \ j \ a \\
  \not\preceq & \quad \{\text{definition of } \lfold\} \\
  & \sqrt{\sqrt{a[i]} \oplus (a[i+1] \oplus \lfold(\oplus)(i+2) \ j \ a)} \\
  \not\preceq & \quad \{\text{associativity of } \oplus, \text{critical path reduced}\} \\
  & \sqrt{\sqrt{\sqrt{a[i]} \oplus (a[i+1] \oplus \lfold(\oplus)(i+2) \ j \ a)} \\
  \not\preceq & \quad \{\text{associativity of } \oplus\} \\
  & \sqrt{\sqrt{\sqrt{\sqrt{a[i]} \oplus (a[i+1] \oplus \ldots \oplus a[i + (m \div 2)] \ldots) \oplus \lfold(\oplus)(i + (m \div 2) + 1) j a}} \\
  \not\preceq & \quad \{\text{repeatedly applying definition of } \lfold\} \\
  & \sqrt{\lfold(\oplus)i ((i + (m \div 2)) a \oplus \lfold(\oplus)(i + (m \div 2) + 1) j a} \\
  \not\preceq & \quad \{\text{arithmetic}\} \\
  & \sqrt{\lfold((i+j) \div 2) a \oplus \lfold(((i+j) \div 2) + 1) j a} \\
  \not\preceq & \quad \{\text{conditionals are free in our cost model, } i \neq j\} \\
  & \sqrt{\text{if } i == j \text{ then } a[i] \text{ else}} \\
  & \lfold ((i+j) \div 2) a \oplus \lfold(((i+j) \div 2) + 1) j a \\
  \not\preceq & \quad \{\text{definition of context}\} \\
  & \sqrt{C[Ifold]}(\oplus) i j a \\
\end{align*}
\]

We have shown that \( Ifold \not\preceq \sqrt{C}[Ifold] \), so by improvement induction, \( Ifold \not\preceq pfold \). Hence, in settings where parallelism is cheap, we should choose \( pfold\) over \( Ifold \).

7 RELATED WORK

The ideas in this article build on work by researchers from within and outside the field of program efficiency, including some from outside the field of computer science in general. In this section, we give a brief overview, focusing on the work that has directly informed our approach.

The original form of improvement theory addressed call-by-name evaluation time costs [Sands 1991]. Later work considered call-by-need evaluation for both time [Moran and Sands 1999a] and space [Gustavsson and Sands 1999, 2001]. Sands [1997] developed general improvement rules for a certain class of operational semantics, which was one of the inspirations for the present article. However, our work requires no assumptions about the underlying operational semantics.

Improvement theory was introduced by Sands [1991] as a purely operational relation defined on the terms of a language. However, predecessors of a more denotational approach to improvement can be found in work on program refinement, such as that of Hoare and He [1990] and Back and
A Generic Foundation for Program Improvement

Wright [1998]. More recently, Hackett and Hutton [2015] have applied categorical techniques to improvement theory based on the notion of preorder-enriched categories.

Metric spaces were first introduced by Fréchet [1906] as a subclass of L-spaces, sets equipped with a notion of the limit of an infinite sequence. The property that contractions have unique fixed points was first observed by Banach [1922], and the name “metric space” is due to Hausdorff [1914]. Lawvere [1973] observed that metric spaces can be viewed as a kind of enriched category, and a number of metric space concepts can be viewed as instances of their categorical counterparts.

The metric space approach to program semantics was first explored by de Bakker and Zucker [1982] in the context of concurrent programs, while it was Escardó [1999] who first applied this approach in functional programming by providing a semantics for the typed functional language PCF. The competing approach of domain theory was first introduced by Scott [1970]. In certain cases, it can be shown that these two approaches coincide [de Bakker and Meyer 1987].

8 CONCLUSION AND FURTHER WORK

We have used metric spaces to develop a new foundation for program improvement that is generic in the source language, operational semantics and underlying cost model. This foundation leads to a number of useful theorems and proof techniques, and we have demonstrated how they can be applied to a range of examples, both replicating existing results and producing novel results. Finally, our approach has given new insight into the problem of space improvement, explaining why this has been difficult in the past and suggesting a simple way to resolve the problem.

Our approach validates many of the proof rules that have been used in earlier work to deal with recursion. However, when applying these proof techniques, there are typically many “administrative” steps where ticks (unit time costs) are moved around or terms are rearranged. Our metric-based theory is in a sense too generic to justify these steps, as they require appealing to a particular language and semantics. It would be useful to have a systematic way to produce these administrative rules from a given operational semantics and cost model, perhaps even automatically. This would have the secondary advantage of treating recursion and simple evaluation as two separate concerns, allowing us to mix and match different techniques for dealing with each.

To provide mechanical assistance with proofs of call-by-need time improvement, the Unie system was recently developed by Handley and Hutton [2018]. A similar tool could be developed based on our metric space framework, offering the power of metric-based improvement to programmers without requiring them to understand all of the theoretical underpinnings. Alternatively, a library for metric space-based improvement could be developed for a general-purpose proof assistant such as Coq or Agda. This would help to promote further developments within the metric space framework, as well as new applications for the results that it provides.

Metric spaces are based on a notion of “distance” between points. It seems likely that this notion of distance could also be used to address questions of quantified improvement, telling us not only whether one program is better than another, but also how much. This may require us to move to quasimetric spaces, where the symmetry requirement is dropped. Quasimetric spaces model situations where it might take a different amount of work to go from A to B than it does to go from B to A, such as when climbing or descending a hill. Existing work has applied metric spaces to complexity functions to create a complexity space [Romaguera and Schellekens 1999; Schellekens 1995], using the Banach fixed point theorem to prove upper bounds on the costs of functions. This suggests that we may be able to use metric spaces to answer questions of how much cost is saved, and whether the asymptotic behaviour is changed.

This article has focused on the problem of improvement in all cases, known as strong improvement. However, in some situations the notion of weak improvement [Moran and Sands 1999a] is more appropriate, in which improvements need only hold up to some constant factor, allowing us to
reason about *asymptotic* rather than absolute cost. Unfortunately, this form of improvement lacks many of the tools that strong improvement has for dealing with recursion, as induction-like proof techniques tend to require a strong improvement in the precondition. It would be interesting to investigate whether our new metric-based approach can provide any insights into how to address this issue, either by developing proof techniques that work for weak improvement, or by developing some kind of half-way point between weak and strong improvement.

**Acknowledgements.** We would like to thank Martin Handley for many useful discussions. 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**

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.*