Metric Spaces for Program Improvement

ANONYMOUS AUTHOR(S)

Improvement theory is an approach to reasoning about program efficiency that allows inequational proofs of improvement to be written in a similar manner to equational proofs of correctness. Existing theories of improvement have suffered from two main difficulties. Firstly, they are fundamentally operational, in contrast to the denotational approaches that tend to be used for questions of correctness. And secondly, they are tied to specific languages and cost models, with each choice requiring a new theory of improvement.

This article addresses these issues using pre-ordered metric spaces, a combination of two usually disparate approaches to program semantics. The resulting theory gives a denotational approach to efficiency, abstracting away from the details of operational semantics to give a theory that is both generic and intuitive, while still remaining faithful to questions of efficiency. 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.

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 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. As observed by Harper [2014], this is a deep and fundamental problem in computing. Ultimately, 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 is an operational theory that is tied to specific languages, semantics and cost models, and so requires a new theory for any new combination of these factors. 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. Furthermore, while these improvement theories do provide powerful techniques for dealing with recursive programs, proving these techniques sound requires a significant amount of work and attention to detail.

The solution we propose is thus: we should use simple, denotational theories that capture operational information. By giving a denotational interpretation of improvement theory, we can develop a generic framework for reasoning about efficiency that abstracts away from unnecessary detail. The setting we use for this is pre-ordered metric spaces, an approach that combines two
separate schools of denotational semantics, in the form of domain theory and metric spaces. We take inspiration from the metric model of PCF developed by Escardó [1999] that captures the number of evaluation steps required to produce a result. The advantage we get from synthesising these two approaches is a powerful theorem for reasoning about fixed points that combines the domain-theoretic Knaster-Tarski theorem with the Banach fixed point theorem for metric spaces, relating the unique fixed point of a contraction to its pre- and post-fixed points.

More concretely, the article makes the following contributions:

- We develop a new, generic framework for denotational theories of program improvement based on the concept of pre-ordered metric spaces;
- We prove a number of useful theorems from the literature in our generic framework;
- We explain why space efficiency has historically been a problem in improvement theory, and show how to avoid these problems in our setting;
- We demonstrate the utility of our framework using a range of examples, including a semantics that mirrors the operational nature of improvement theory in a denotational way.

To make the ideas accessible to a broad spectrum of readers, we assume no specialised knowledge of improvement theory or metric spaces, and provide introductions to these concepts in Sections 2 and 3. Section 4 demonstrates how pre-ordered metric spaces can be used to build denotational semantics that capture efficiency information, and presents novel proofs of existing results in this general framework. Sections 5 to 7 give a variety of examples of how our framework can be instantiated to produce improvement theories for specific semantic models, and proves efficiency properties of programs in these models, starting with a theory that closely follows the operational character of improvement theory. Section 8 gives a survey of the related work, and section 9 summarises and discusses possible avenues for future work.

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:

$$M \triangleright N \iff \forall C. \ C[M] \Downarrow^n \Rightarrow C[N] \Downarrow^\leq n$$

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 $M \equiv N$, 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].

In an improvement theory, it is usually necessary to have an operation that represent one unit
of cost, together with rules for manipulating these costs. For example, in the call-by-need time
improvement due to Moran and Sands [1999a] there is the tick operation that adds one unit of time
cost to a term, written ✓. The rules for manipulating ticks are then called tick algebra. For example,
the following rules are valid in Moran and Sands’ tick algebra:

\[
\begin{align*}
\text{let } x &= V \text{ in } C[x] & \Leftrightarrow & \text{ let } x &= V \text{ in } C[\sqrt{V}] & \text{(value-\(\beta\))} \\
\sqrt{\text{case } M \text{ of } \{\text{pat}_i \rightarrow N_i\}} & \Leftrightarrow \text{ case } M \text{ of } \{\text{pat}_i \rightarrow \sqrt{N_i}\} & \text{(case-\(\sqrt{}\))} \\
\sqrt{\text{case } \text{ case } M \text{ of } \{\text{pat}_i \rightarrow N_i\} \text{ of } \{\text{pat}_j \rightarrow L_j\}} & \Leftrightarrow \text{ case } M \text{ of } \{\text{pat}_i \rightarrow \text{ case } N_i \text{ of } \{\text{pat}_j \rightarrow L_j\}\} & \text{(case-case)} \\
\sqrt{M \succeq M} & \Leftrightarrow M & \text{(\(\sqrt{}\)-elim)} \\
\sqrt{\text{case } \text{ c}_i x_1 x_2 \ldots x_{a(i)} \text{ of } \{\ldots ; \text{ c}_i y_1 y_2 \ldots y_{a(i)} \rightarrow M ; \ldots \}} & \Leftrightarrow \text{ M}[x_1/y_1, x_2/y_2, \ldots , x_{a(i)}/y_{a(i)}] & \text{(case-cons)} \\
\sqrt{\text{case } \text{ \sqrt{M} \text{ of } \{\text{pat}_i \rightarrow \text{ case } N_i\} \text{ of } \{\text{pat}_i \rightarrow \text{ case } N_i\}} & \Leftrightarrow \sqrt{\text{case } M \text{ of } \{\text{pat}_i \rightarrow \text{ case } N_i\}} & \text{(\(\sqrt{}\)-float)}
\end{align*}
\]

(Note that in the above rules V is intended to denote a term that is in value form, c_i is a constructor
of arity a(i), and all variables are assumed to be distinct.)

One particularly useful proof rule that appears in a number of improvement theories is improve-
ment induction. This 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.
Improvement induction is formally captured by the following inference rule:

\[
\frac{M \succeq \sqrt{C[M]} \quad \sqrt{C[N]} \succeq N}{M \succeq N}
\]

To give an example of an improvement that can be proved with improvement induction, consider
the following recursive definition for the append operator on lists:

\[
(+ +) = \lambda xs \ ys \rightarrow \text{ case } xs \text{ of } \[
\emptyset \rightarrow \emptyset \\
(x : xs') \rightarrow x : (xs' + + ys)
\]
\]

Note that this definition takes the form of a lambda abstraction, and so is in value form.

Using the above definition, a left-associated append (xs + + ys) + + zs will intuitively take more
steps to evaluate than a right-associated append xs + + (ys + + zs). This behaviour arises from the
fact that + + 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 in combination with call-by-need
tick algebra. To verify the improvement (xs + + ys) + + zs \succeq xs + + (ys + + zs), we first define the
following context, in which [−] denotes the hole where a term will be substituted:

\[
C = \text{ case } xs \text{ of } \[
\emptyset \rightarrow \emptyset + + zs \\
x : xs \rightarrow x : [−]
\]
\]
By value-β, \(xs ++ (ys ++ zs)\) is cost equivalent to \(\sqrt{C}[xs ++ (ys ++ zs)]\). Hence, by improvement induction, it suffices to show that \((xs ++ ys) ++ zs \subseteq \sqrt{C}(xs ++ ys) ++ zs\), which can itself be verified by the following calculation in tick algebra:

\[
\begin{align*}
(xs ++ ys) ++ zs \\
\implies \text{\{value-β for append\}} \\
\check\text{case } xs ++ ys \text{ of} \\
[[]] \rightarrow zs \\
\implies \text{\{value-β for append\}} \\
\check\text{case } \check\text{case } xs \text{ of} \\
[[]] \rightarrow ys \\
\hspace{1cm} x : xs \rightarrow x : (xs ++ ys) \\
\text{of} \\
[[]] \rightarrow zs \\
\hspace{1cm} r : rs \rightarrow r : (rs ++ zs) \\
\implies \text{\{case-✓\}} \\
\check\text{case } \check\text{case } xs \text{ of} \\
[[]] \rightarrow \check ys \\
\hspace{1cm} x : xs \rightarrow \check x : (xs ++ ys) \\
\text{of} \\
[[]] \rightarrow zs \\
\hspace{1cm} r : rs \rightarrow r : (rs ++ zs) \\
\implies \text{\{case-case\}} \\
\check\text{case } xs \text{ of} \\
[[]] \rightarrow \check\text{case } \check ys \text{ of} \\
[[]] \rightarrow zs \\
\hspace{1cm} r : rs \rightarrow r : (rs ++ zs) \\
\hspace{1cm} x : xs \rightarrow \check\text{case } x : (xs ++ ys) \text{ of} \\
[[]] \rightarrow zs \\
\hspace{1cm} r : rs \rightarrow r : (rs ++ zs) \\
\check\text{\{✓-elim\}} \\
\check\text{case } xs \text{ of} \\
[[]] \rightarrow \check\text{case } \check ys \text{ of} \\
[[]] \rightarrow zs \\
\hspace{1cm} r : rs \rightarrow r : (rs ++ zs) \\
\hspace{1cm} x : xs \rightarrow \check\text{case } x : (xs ++ ys) \text{ of} \\
[[]] \rightarrow zs \\
\hspace{1cm} r : rs \rightarrow r : (rs ++ zs) \\
\implies \text{\{case-cons\}} \\
\check\text{case } xs \text{ of} \\
[[]] \rightarrow \check\text{case } \check ys \text{ of} \\
[[]] \rightarrow zs \\
\hspace{1cm} r : rs \rightarrow r : (rs ++ zs) \\
\hspace{1cm} x : xs \rightarrow x : ((xs ++ ys) ++ zs)
\end{align*}
\]
In conclusion, we have shown that in the setting of call-by-need time improvements, associativity of the append operator in lists is indeed a time improvement.

Improvement induction can be thought of as a unique fixed-point principle for improvement theory. The first premise gives us the chain $M \triangleright \triangleright \triangleright \triangleright \triangleright \triangleright C[M] \triangleright \triangleright \triangleright C[\triangleright \triangleright C[M]] \triangleright \ldots$ by repeated application of $C$, which will tend toward some notional fixed point. The second premise states that $N$ is such a fixed point. Therefore, if fixed points are unique, $M$ will be improved by $N$. The main motivation behind the present article is to give a denotational account of this idea, making this sketch proof rigorous. In order to accomplish this, we must move to a denotational semantics in which fixed points are unique. The natural setting for this is that of metric spaces.

### 3 Metric Spaces and Domains

#### 3.1 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]:

\[
\begin{align*}
    d(x, y) &= 0 \iff x = y \quad \text{(identity of indiscernibles)} \\
    d(x, y) &= d(y, x) \quad \text{(symmetry)} \\
    d(x, z) &\leq d(x, y) + d(y, z) \quad \text{(triangle inequality)}
\end{align*}
\]

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 clear.

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.

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.

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 $f : X \to Y$ 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 $f : X \to Y$ is a contraction if there is a constant $0 < c < 1$ such that for any $x_1, x_2 \in X$, we have $d(f(x_1), f(x_2)) \leq c \cdot d(x_1, x_2)$. Any contraction is also non-expansive, but there are non-expansive maps that are not contractions, such as 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 [Banach 1922]. We denote the unique fixed point of a contraction $f$ as $\text{fix } f$.

**Theorem 3.1 (Banach fixed point theorem).** Given a complete, non-empty metric space $(X, d)$ and a contraction $f : X \to X$, $f$ has a unique fixed point $\text{fix } f$.

**Proof.** Let $x_0$ be an arbitrary element of $X$ and let $x_{n+1} = f(x_n)$. This defines a sequence in $X$. Let $c$ be the contraction constant of $f$. A simple induction will show that $d(x_{n+1}, x_n) \leq c^n d(x_1, x_0)$. Let $m > n$. Then:

\[
\begin{align*}
d(x_m, x_n) & \leq \text{triangle inequality} \\
d(x_m, x_{m-1}) + d(x_{m-1}, x_{m-2}) + \cdots + d(x_{n+1}, x_n) & \leq \text{above} \\
c^{m-1} \cdot d(x_1, x_0) + c^{m-2} \cdot d(x_1, x_0) + \cdots + c^n \cdot d(x_1, x_0) & \leq \text{arithmetic} \\
c^n \cdot d(x_1, x_0) \sum_{k=0}^{\infty} q^k
\end{align*}
\]

Clearly this will become arbitrarily small as $n$ grows, so the sequence is Cauchy. Now, define $\text{fix } f = \lim_{n \to \infty} x_n$. We prove that this is a fixed point of $f$.

\[
\begin{align*}
f(\text{fix } f) & = \text{definition of fix } f \\
f(\lim_{n \to \infty} x_n) & = \text{f contraction, so must be continuous} \\
\lim_{n \to \infty} f(x_n) & = \text{definition of } x_{n+1} \\
\lim_{n \to \infty} x_{n+1} & = \text{definition of } x_{n+1} \\
\text{fix } f & = \text{definition of fix } f, \text{ limits}
\end{align*}
\]

Finally, we prove that this fixed point is unique. Suppose $f(y) = y$. Since both $f$ and $y$ are fixed points of $f$, we know that $d(\text{fix } f, y) = d(f(\text{fix } f), f(y))$. Also, because $f$ is a contraction with
constant \(c\), we know that \(d(f(\text{fix } f), f(y)) \leq c \cdot d(\text{fix } f, y)\). Therefore, \(d(\text{fix } f, y) \leq c \cdot d(\text{fix } f, y)\). But the only value that satisfies this relationship is 0, so by identity of indiscernibles, \(\text{fix } f = y\). \(\square\)

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. 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.

### 3.2 Domain Theory

The metric space approach to denotational semantics contrasts with the competing approach of domain theory based upon lattices or complete partial orders. The domain-theoretic ordering on programs often corresponds to some natural ordering by definedness. 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. In this context, the meaning of a program is the least solution to its defining equations, generally the least fixed point of a monotone function. Domain theory was first introduced by Scott [1970]. In certain cases, it can be shown that the domain-theoretic and metric-based approaches to semantics coincide [de Bakker and Meyer 1987].

In lattice-based denotational semantics, there is a powerful tool for reasoning about recursive programs in the form of the Knaster-Tarski theorem:

**Theorem 3.2 (Knaster-Tarski).** For any complete lattice \((L, \leq)\) and monotone function \(f : L \to L\), the least fixed and least pre-fixed points of \(f\) exist and are equal. In a similar manner, the greatest fixed and greatest post-fixed points exist and are also equal.

**Proof.** Let \(p\) be the greatest lower bound of the set of pre-fixed points of \(f\); this must exist because of the completeness of \(L\). Then, for any pre-fixed point \(x\), we have

\[
\begin{align*}
p \leq x & \quad \{x \text{ pre-fixed point}\} \\
\implies & \quad \{f \text{ monotone}\} \\
f(p) & \leq f(x) \\
\implies & \quad \{x \text{ pre-fixed point, so } f(x) \leq x\} \\
f(p) & \leq x
\end{align*}
\]

Therefore, \(f(p)\) is a lower bound of the set of pre-fixed points of \(f\), so \(f(p) \leq p\) by the definition of \(p\). Thus, \(p\) is the least pre-fixed point of \(f\). However, by monotonicity we also have \(f(f(p)) \leq f(p)\), so \(f(p)\) is a pre-fixed point, and since \(p\) is a lower bound of the set of pre-fixed points, \(p \leq f(p)\). Therefore, \(p\) is a fixed point as well as being the least pre-fixed point. Since all fixed points are also pre-fixed points, \(p\) is the least fixed point of the function \(f\). \(\square\)

Using the Knaster-Tarski theorem, one can prove that the least fixed point of a monotone \(f\) is less than some other element \(x\) of the lattice simply by proving that \(x\) is a pre-fixed point of \(f\).

### 3.3 Pre-Ordered Metric Spaces

We can combine the two approaches of metric spaces and domain theory using the idea of a pre-ordered metric space. Put simply, a pre-ordered metric space is a metric space \((X, d)\) equipped with a reflexive and transitive relation (a pre-order) \(<\) that respects limits: if two sequences \(x_1, x_2, x_3, \ldots\) and \(y_1, y_2, y_3, \ldots\) both have limits, and we know that \(x_i < y_i\) for all \(i\), we require that the limits share the same relationship, i.e. \(\lim x_i < \lim y_i\). In this manner, we can have it both ways, gaining
the unique fixed point principle of complete metric spaces but still having a Knaster-Tarski-like theorem for dealing with pre- and post-fixed points:

**Theorem 3.3.** Given a complete, pre-ordered metric space \((X, d, \prec)\) and monotone contraction \(f : X \to X\), if \(f(x) \prec x\) (i.e. \(x\) is a pre-fixed point) then \(\text{fix } f \prec x\). In other words, \(\text{fix } f\) is the least pre-fixed point of \(f\). Dually, \(\text{fix } f\) is the greatest post-fixed point of \(f\).

**Proof.** Consider following two infinite sequences:

\[
\begin{align*}
x & \quad x \quad x \quad x \quad \ldots \\
x & \quad f(x) \quad f(f(x)) \quad f(f(f(x))) \quad \ldots 
\end{align*}
\]

The first sequence is trivially Cauchy, while the second is Cauchy by the same argument as for the Banach fixed point theorem. A simple induction will then show that at every point, the second sequence will be below the first in the order \(\prec\). Therefore, the limit of the first sequence \(x\) must be below the limit of the second sequence \(\text{fix } f\) by the definition of a pre-ordered metric space. The dual case follows from the above by reversing the pre-order \(\prec\). \(\Box\)

4 IMPROVING WITH METRICS

4.1 Denotational Improvement Semantics

Recall the improvement induction rule:

\[
\begin{align*}
M \triangleright \sqrt{C}[M] & \quad \sqrt{C}[N] \not\triangleright N \\
\hline
M \triangleright N
\end{align*}
\]

A combination of domain-theoretic and metric space-based semantics gives us exactly the right ingredients to prove the improvement induction rule sound. The first premise, \(M \triangleright \sqrt{C}[M]\), suggests that \(M\) will be a pre-fixed point of some function corresponding to the context \(\sqrt{C}\), while the second premise, \(\sqrt{C}[N] \not\triangleright N\), suggests that \(N\) will be the unique fixed point of the same function. Therefore, the above theorem will give us that \(M \triangleright N\).

To formalise this idea, given a language with a set of terms \(Trm\), a set of contexts \(Ctx\), a special context \(\sqrt{\cdot}\) and an improvement relation \(\triangleright\), a denotational improvement semantics consists of:

- A pre-ordered metric space \((X, d, \prec)\)
- A term interpretation function \(\llbracket \cdot \rrbracket : Trm \to X\)
- A context interpretation function \(\llbracket \cdot \rrbracket : Ctx \to (X \to X)\)

such that

- For all terms \(M, N\), we have \(M \triangleright N\) iff \(\llbracket M \rrbracket < \llbracket N \rrbracket\)
- For all contexts \(C\) and terms \(M\), \(\llbracket C \rrbracket\) is monotone/non-expansive and \(\llbracket C \rrbracket(\llbracket M \rrbracket) = \llbracket C[M] \rrbracket\)
- \((\sqrt{\cdot})\) is a contraction

Since the requirement that \(M \triangleright N\) iff \(\llbracket M \rrbracket < \llbracket N \rrbracket\) completely constrains what \(\triangleright\) can be, we will sometimes leave the definition of \(\triangleright\) implicit and only define \(<\). We will write \(\delta\) for \((\sqrt{\cdot})\). Given such a denotational improvement semantics, the above sketch proof can now be formalised:

**Theorem 4.1 (Improvement induction, denotationally).** Given a language with a denotational improvement semantics, the improvement induction rule will hold for that language.

**Proof.** Given the first premise, \(M \triangleright \sqrt{C}[M]\), we know that \(\llbracket M \rrbracket < (\delta \cdot \llbracket C \rrbracket)(\llbracket M \rrbracket)\), which expresses that \(\llbracket M \rrbracket\) is a post-fixed point of \(\delta \cdot \llbracket C \rrbracket\). The second premise states that \(\sqrt{C}[N] \not\triangleright N\),
and hence \([N]\) is the unique fixed point of the same function. Since this function is a monotone contraction, Theorem 3.3 applies and so \([M]\prec[N]\). Therefore, we have \(M \succ N\). 

The space improvement theory of Gustavsson and Sands [1999, 2001] lacked this kind of induction rule. Our proof provides an intuition as to why: while the theory has a number of “space gadgets” to play the role of \(\checkmark\), none of these correspond to a contraction. This is because the increase in space usage they cause is only momentary, allowing the space to be re-used. To see why this is a problem, suppose we have two programs \(M\) and \(N\) that each use the same amount of memory, but produce different results. For any gadget \(\checkmark\), there will be some context \(\mathcal{C}\) that re-uses the space it uses, implying that \(\checkmark\mathcal{C}[M] \equiv \mathcal{C}[M]\) for any \(M\). This means that \(\delta \cdot [\mathcal{C}]\) will be the identity, i.e. not a contraction, but this contradicts the requirement that \(\delta\) be a contraction. Therefore, if we wish to use improvement induction in the context of space efficiency, we need to add some operation that persistently takes up space, increasing space usage by one in any context.

4.2 Continuity

In an operational improvement theory, it is useful to have theorems that characterise the behaviour of a recursive term by means 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 let \(x = M\) 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, usually a bottom element for the underlying improvement order.) Similarly, in our denotational approach, we must have a way to characterise programs in terms of finite approximations. In the context of a denotational improvement semantics for a language with recursive bindings, we will say that the semantics is continuous for a class of bindings if for all bindings in the class \(x = \mathbb{D}[x]\) and all contexts \(\mathcal{C}\) that contain no occurrences of \(x\), we have \([\text{let } x = \mathbb{D}[x] \text{ in } \mathcal{C}[x]] = [\mathcal{C}](\text{fix}(\delta \cdot [\mathbb{D}]))\). The use of the contraction \(\delta\) ensures that the fixed point \(\text{fix}(\delta \cdot [\mathbb{D}])\) exists and is unique.

Furthermore, if there is some sequence of terms \(M_0, M_1, M_2, \ldots\) where \([M_n] = [\mathcal{C}](((\delta \cdot [\mathbb{D}])^n(\bot)))\), we say that the term let \(x = \mathbb{D}[x] \text{ in } \mathcal{C}[x]\) has unwindings \(M_0, M_1, M_2, \ldots\). In this case, we say that the semantics has syntactic continuity. Note that the interpretations of these unwindings will form a Cauchy sequence whose limit is given by \([\mathcal{C}](\text{fix}(\delta \cdot [\mathbb{D}]))\).

4.3 Unwindings and Improvement

If syntactic continuity holds for our chosen denotational improvement semantics, we can use the unwindings of two terms to prove an improvement between them. Suppose that we have two terms \(M\) and \(N\) with recursive bindings, 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:

\[
\exists k. \forall n \geq k. M_n \succ N_n \quad \frac{M \succ N}{M \succ N}
\]

This result follows easily from the requirement that the pre-order in a pre-ordered metric space respects the order. A corollary is an analogue to the well-known squeeze theorem for limits of
sequences. Given three terms $M$, $N$ and $P$ with unwindings $M_n$, $N_n$ and $P_n$, we have:

$$\exists k. \forall n \geq k. M_n \models N_n \models P_n \quad M \models P$$

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 $P$.

### 4.4 The Improvement Theorem

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

$$\text{let } f = V \text{ in } V_n \quad \text{let } f = V \text{ in } W_n \quad \text{let } f = W \text{ in } N_n$$

If the semantics is continuous in the sense described above for some class of bindings that includes $f = V$ and $f = W$, this amounts to requiring that for any non-expansive and monotone $f, g$,

$$f(\text{fix } (\delta \cdot f)) < g(\text{fix } (\delta \cdot f)) \Rightarrow \text{fix } (\delta \cdot f) < \text{fix } (\delta \cdot g)$$

Given the precondition $f(\text{fix } f) < g(\text{fix } f)$, we can show by induction that $(\delta \cdot f)^n(\text{fix } (\delta \cdot f)) < (\delta \cdot g)^n(\text{fix } (\delta \cdot f))$ for all $n$. The result follows from the properties that $\lim_{n \to \infty}(\delta \cdot f)^n(x) = \text{fix } (\delta \cdot f)$, $\lim_{n \to \infty}(\delta \cdot g)^n(x) = \text{fix } (\delta \cdot g)$ and the requirement that limits respect the pre-order.

### 5 EXAMPLE: COST-IN-CONTEXT SEMANTICS

Escardó [1999] demonstrated how metric spaces could be used to give a denotational semantics for typed functional programs, with distance at base types being defined based on the difference in cost to produce a result. Inspired by this idea, we produce an untyped version of this model, based on the idea of cost-in-context that forms the basis of operational improvement theory. 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, let $\mathsf{Trm}$ be the set of valid terms in a programming language, $\mathsf{Ctx}$ 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 $\text{cost} : \mathsf{Trm} \to \mathbb{N}_\omega$ (where $\mathbb{N}_\omega = \mathbb{N} \cup \{\infty\}$) that assigns to every terminating term the cost of evaluating it, and to every non-terminating term the symbol $\infty$ that represents a value greater than any finite cost. That is, we have:

$$\text{cost}(M) = \begin{cases} n & \text{if } M \Downarrow^n \\ \infty & \text{if } M \text{ diverges} \end{cases}$$

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

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

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 \( M \triangleright N \iff [N] \leq^* [M] \):

\[
[N] \leq^* [M] \\
\iff \{\text{definition of } \leq^*\} \\
\forall C. [N](C) \leq [M](C) \\
\iff \{\text{definition of } [\cdot]\} \\
\forall C. \text{cost}(C[N]) \leq \text{cost}(C[M]) \\
\iff \{\text{definition of cost, properties of } \leq\} \\
\forall C. [M] \Downarrow^n \Rightarrow C[N] \Downarrow^\leq^n \\
\iff \{\text{definition of } \triangleright\} \\
M \triangleright N
\]

Similarly, we can also show that \( M \ntriangleright N \iff [M] = [N] \).

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 \( \text{Ctx} \to N_{\omega} \), as follows:

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

(Note that we define \( \sup \emptyset = 0 \).) This definition states 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 Escardó’s idea 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. Because the ordering \( \leq^* \) is also pointwise, limits respect the ordering, and so \( (\text{Ctx} \to N_{\omega}, d, \leq^*) \) is a pre-ordered metric space.

Each context \( C \) induces a map \([C]\) on the set \( \text{Ctx} \to N_{\omega} \), defined by \([C](p) = \lambda D. \ p(D[C[-]])\). This map is clearly monotone, and a simple calculation shows that it is also non-expansive:

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

The map \([C]\) is the denotational analogue of the context \( C \), in that \([C]([M]) = [C[M]]\).
Finally, there is a function \( \delta \) on \( \text{Ctx} \rightarrow \mathbb{N}_\omega \) that adds one unit of cost in all contexts, defined by 
\[
\delta(p) = \lambda C. p(C) + 1.
\]
It is clear that \([\sqrt{\_}] = \delta\). We can show that this function is a contraction:
\[
d(\delta(p), \delta(q)) = \{\text{definitions of } d, \delta\}
\]
\[
\sup \{r \min(p(C)+1, q(C)+1) \mid C \in \text{Ctx}, \; p(C) + 1 \neq q(C) + 1\}
\]
\[
= \{\text{arithmetic}\}
\]
\[
\sup \{r \cdot r \min(p(C), q(C)) \mid C \in \text{Ctx}, \; p(C) \neq q(C)\}
\]
\[
= \{\text{definition of } d\}
\]
\[
r \cdot d(p, q)
\]

To summarise, we have a pre-ordered metric space \( (\text{Ctx} \rightarrow \mathbb{N}_\omega, d, \leq^*) \) and interpretation functions 
\[
[\_] : \text{Trm} \rightarrow (\text{Ctx} \rightarrow \mathbb{N}_\omega) \text{ and } [\_] : \text{Ctx} \rightarrow (\text{Ctx} \rightarrow \mathbb{N}_\omega) \rightarrow (\text{Ctx} \rightarrow \mathbb{N}_\omega),
\]
and we have verified that these satisfy the requirements to be a denotational improvement semantics.

Finally, we address the question of continuity. For the case of call-by-name time costs, continuity will hold for all bindings. First, we show that 
\[
[\text{let } x = D[x] \text{ in } x] = \text{fix}(\delta \cdot [D]),
\]
which we do by showing that the left hand side is a fixed point of \( \delta \cdot [D] \), which is a contraction:
\[
[\text{let } x = D[x] \text{ in } x]
\]
\[
= \{\text{lookup costs one tick}\}
\]
\[
[\text{let } x = D[x] \text{ in } \sqrt{D}[x]]
\]
\[
= \{x \text{ free in } D\}
\]
\[
[\sqrt{D}[\text{let } x = D[x] \text{ in } x]]
\]
\[
= \{\text{interpretation of contexts}\}
\]
\[
[\sqrt{D}][[\text{let } x = D[x] \text{ in } x]]
\]

Now we can verify the continuity condition:
\[
[\text{let } x = D[x] \text{ in } C[x]]
\]
\[
= \{x \text{ free in } C\}
\]
\[
[C[\text{let } x = D[x] \text{ in } x]]
\]
\[
= \{\text{interpretation of contexts}\}
\]
\[
[C][[\text{let } x = D[x] \text{ in } x]]
\]
\[
= \{\text{see above}\}
\]
\[
[C][\text{fix}(\delta \cdot [D])]
\]

This reasoning is invalid for call-by-need semantics, as the lookup loses sharing between the different levels of recursion. However, we can recover continuity in the call-by-need case by restricting ourselves to bindings where the right hand side is already in value form.

The space theory of Gustavsson and Sands [1999, 2001] has a syntactic continuity theorem, but this is not continuity in our sense. In particular, while it characterises terms of the form 
\[
\text{let } f = V \text{ in } C[f]
\]
as limits of sequences of finite unwinding, the interpretations of this sequence will not in general be Cauchy when our cost function measures space rather than time. The reason
is that while it takes more time to distinguish successive unw windings 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. Since the sequence is not guaranteed to be Cauchy, in general there is no way to ensure continuity.

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 $f : \mathbb{N}_\omega \times \mathbb{N}_\omega \to \mathbb{N}_\omega$ 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 bindings as the theory that considers time alone (as the interpretation of the time tick $✓$ will remain contractive 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.

### 5.1 Compact Reverse

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

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

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

Intuitively, $\text{reverse}$ will run in quadratic time, as each use of the append operator is linear and there will be a linear number of these uses. On the other hand, $\text{reverse}'$ will be linear, as it uses only constant time operations at each level of recursion. Prior work has used improvement theory to verify that this is in fact an improvement [Hackett and Hutton 2014].

The fast version $\text{reverse}'$ also has another advantage: it uses less space. However, we cannot formally verify this in the same way as 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 improvement induction, which as mentioned before 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:

\[
\begin{align*}
M_0 &= \text{reverse}' \; xs \\
M_1 &= \text{case} \; xs \; \text{of} \\
& \quad [] \rightarrow [] \\
& \quad x : xs' \rightarrow \text{reverse}' \; xs' \leftrightarrow [x] \\
M_2 &= \text{case} \; xs \; \text{of} \\
& \quad [] \rightarrow [] \\
& \quad x : xs' \rightarrow (\text{case} \; xs' \; \text{of} \\
& \quad \quad [] \rightarrow [] \\
& \quad \quad x' : xs'' \rightarrow \text{reverse}' \; xs'' \leftrightarrow [x']) \leftrightarrow [x]
\end{align*}
\]
The interpretations of these terms form a Cauchy sequence with limit \[ reverse\; xs \], because each subsequent term will mimic more of the space behaviour of \( reverse \); note that each level of recursion strictly increases the space usage. Therefore, it suffices to show that the sequence forms a chain of reversed improvements, i.e. \( M_{n+1} \trianglerighteq M_n \) for all \( n \geq 0 \). The ordering is reversed because the improved version \( reverse' \) 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, \( M_1 \trianglerighteq M_0 \). Our proof of the base case makes use of a lemma, namely that \( revcat\; xs\; [ ] \trianglerighteq [ x ] \trianglerighteq revcat\; xs\; [ x ] \). We use the following auxiliary function definition:

\[
revcat'\; x\; xs\; ys = \text{case } xs\; \text{ of}
\begin{align*}
[ ] & \rightarrow [ ] \trianglerighteq [ x ] \\
x' : xs' & \rightarrow revcat'\; x\; xs'\; (x' : ys)
\end{align*}
\]

Note that \( revcat\; xs\; [ ] \trianglerighteq [ x ] \) is cost-equivalent to \( revcat'\; x\; xs\; [ ] \), as unrolling \( revcat\; xs\; [ ] \) and distributing \( \trianglerighteq [ x ] \) over the cases will result in the unrollings of the latter.

Next, we consider the following chain of unfoldings:

\[
N_0 = revcat\; xs\; [ x ]
\]
\[
N_1 = \text{case } xs\; \text{ of}
\begin{align*}
[ ] & \rightarrow [ ] \trianglerighteq [ x ] \\
x' : xs' & \rightarrow revcat\; xs'\; ([ x' ] \trianglerighteq [ x ])
\end{align*}
\]
\[
N_2 = \text{case } xs\; \text{ of}
\begin{align*}
[ ] & \rightarrow [ ] \trianglerighteq [ x ] \\
x' : xs' & \rightarrow \text{case } xs'\; \text{ of}
\begin{align*}
[ ] & \rightarrow [ x ] \trianglerighteq [ x ] \\
x'' : xs'' & \rightarrow revcat\; xs''\; ([ x'', x' ] \trianglerighteq [ x ])
\end{align*}
\end{align*}
\]

This forms a reversed chain of improvements, starting at \( revcat\; xs\; [ x ] \). 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 \( revcat'\; x\; xs\; [ ] \), as each successive term approximates it more closely, and this is cost-equivalent to \( revcat\; xs\; [ ] \trianglerighteq [ x ] \). Hence, we can conclude that \( revcat\; xs\; [ ] \trianglerighteq [ x ] \) is improved by \( revcat\; xs\; [ x ] \), as required.

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

\[
M_1
\equiv
\{\text{definition of } M_1\}
\begin{align*}
\text{case } xs\; \text{ of} & \\
[ ] & \rightarrow [ ] \\
x : xs' & \rightarrow reverse'\; xs' \trianglerighteq [ x ]
\end{align*}
\]

\[
\equiv
\{\text{definition of } reverse'\}
\begin{align*}
\text{case } xs\; \text{ of} & \\
[ ] & \rightarrow [ ]
\end{align*}
\]
\[ x : xs' \rightarrow \text{revercat} \; xs' \; [] \; \ddagger \; [x] \]

\[ \begin{cases} \text{(above lemma)} \\
\text{case} \; xs \; \text{of} \\
\; [] \rightarrow [] \\
\; x : xs' \rightarrow \text{revercat} \; xs' \; [x] \\
\end{cases} \]

\[ \begin{aligned} \\
\text{(definition of revercat)} \\
\text{revercat} \; xs \; [] \\
\end{aligned} \]

\[ \begin{aligned} \\
\text{(definition of reverse') } \\
\text{reverse'} \; xs \\
\end{aligned} \]

\[ \equiv \begin{aligned} \\
\text{(definition of } M_0) \\
M_0 \\
\end{aligned} \]

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 EXAMPLE: CLOCKED BÖHM TREES

Böhm trees are a model of the lambda calculus where terms without a normal form are represented by a notional “infinite” normal form. The Böhm tree \( BT(M) \) of a term \( M \) is defined as follows:

- If \( M \) has no head normal form, then \( BT(M) = \bot \);
- If \( M \) has head normal form \( \lambda x_1 x_2 x_3 \ldots x_n \cdot y M_1 \ldots M_m \), where the \( x_i \) are distinct but \( y \) may be one of the \( x_i \), then \( BT(M) = \lambda x_1 x_2 x_3 \ldots x_n \cdot y(BT(M_1) \ldots BT(M_m)) \).

Note that this definition may not terminate, in which case we want the “limit” of the finite approximations of the tree. For example, consider the following trees:

\[
\begin{array}{ccc}
\lambda x y z . x & \bot & \lambda f . f \\
\overleftarrow{z} & y & \\
& \overleftarrow{z} & f \\
end{array}
\]

The tree on the left is for the \( S \) combinator, the tree in the middle is for the divergent term \( (\lambda x . x)(\lambda x . x) \), and the tree on the right is the tree for any fixed-point combinator.

Böhm trees are a useful model, as any two terms that are not \( \beta \)-equal will necessarily have different trees. However, Böhm trees in some sense identify too many things — for example, there are many fixed-point combinators that are not \( \beta \)-equal, yet they all have the same Böhm tree — and there have been many extensions to try to deal with this. One such approach is given by the notion of clocked Böhm trees [Endrullis et al. 2014], where each non-\( \bot \) node is annotated with the number of head reductions required to reach the head normal form.

There is a natural ordering on clocked Böhm trees, where \( T_1 < T_2 \) either if \( T_1 \) is \( \bot \) or if \( T_1 = r[c_1](T_{1,1} \ldots T_{1,n}) \) and \( T_2 = r[c_2](T_{2,1} \ldots T_{2,n}) \) with \( c_1 \leq c_2 \) and \( T_{1,i} < T_{2,i} \) for all \( i \). For example, the following two trees are both fixed-point combinators,
but the one on the right has lower cost than the one on the left.

We can define a metric on these trees. Given two clocked Böhm trees, \( T_1 \) and \( T_2 \), let \( \text{diff}(T_1, T_2) \) be the minimum total cost along any path in \( T_1 \) or \( T_2 \) that leads to a position in which they differ; if the trees are identical, let this be \( \infty \). We treat \( \bot \) as a node of infinite cost for this purpose. For example, the \( \text{diff} \) of the two trees above would be 1, as there is a 1-cost path in the second tree to a node that differs from the same position in the first tree, i.e. the root.

Given this function \( \text{diff} \), define \( d(T_1, T_2) = 2 - \text{diff}(T_1, T_2) \), with the convention that \( 2^{-\infty} = 0 \). This will turn the set of clocked Böhm trees into an ultrametric space. This space is complete, as any Cauchy sequence of trees will approximate more and more of the cost of a potentially infinite tree, each subtree either going to \( \bot \) or eventually stabilising. Furthermore, limits will respect the order \( < \) by a similar argument. Therefore, the set of clocked Böhm trees forms a pre-ordered metric space. We write \( \delta \) for the contractive function that adds one to the cost at the root of a tree.

We now have all the ingredients of a denotational improvement semantics besides the interpretation of contexts. Given a context \( C \), we can define its interpretation \( \left[ C \right] \) inductively by

\[
\left[ x \right](T) = [x] \\
\left[ CD \right](T) = \text{app}([C](T), [D](T)) \\
\text{app}(\lambda x. T_1, T_2) = \delta(T_1[x := T_2]) \\
\text{app}(x(T_{1,1} \ldots T_{1,m}), T_2) = \delta(x(T_{1,1} \ldots T_{1,m}T_2)) \\
\left[ \lambda x. C \right](T) = \lambda x. [C](T)
\]

where \( \lambda x.T \) prepends \( x \) to the list of bound variables at the root of \( T \), and \( T_1[x := T_2] \) is capture-avoiding substitution. Note that application adds one to the root of the resulting tree, to account for the cost of the extra head reduction. It is easy to see that the resulting function will be non-expansive and monotone. Finally, we take \( \sqrt{\cdot} = (\lambda x.x)[\cdot] \) and note that \( \left[ \sqrt{\cdot} \right] = \delta \). Thus we see that clocked Böhm trees form a denotational improvement semantics for the lambda calculus.

6.1 Comparing Fixed-Point Combinators

Now that we have a denotational improvement semantics for the lambda calculus, we can use improvement theory to compare the efficiency of Curry’s fixed-point combinator

\[
Y = \lambda f. (\lambda x.f(xx))(\lambda x.f(xx))
\]

with that of Turing’s fixed-point combinator

\[
\theta = (\lambda xy.y(xy))(\lambda xy.y(xy))
\]

First, we note that Turing’s combinator reduces to \( \lambda y.y(\theta y) \) in one step, so \( \theta \preceq \sqrt{\lambda y.y(\theta y)} \). By improvement induction, if we can show an improvement between \( Y \) and \( \lambda y.y(\tilde{Y}y) \) then we will
have an improvement between \( Y \) and \( \theta \). We calculate as follows:

\[
Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))
\]

\( \triangleright \) \{\( \beta \)-reduction\}

\[
\lambda f.\sqrt{f((\lambda x.f(xx))(\lambda x.f(xx)))}
\]

\( \triangleright \) \{pushing \( \sqrt \) inwards is an improvement\}

\[
\lambda f.f(\sqrt{f((\lambda x.f(xx))(\lambda x.f(xx)))})
\]

\( \triangleright \) \{\( \beta \)-reduction\}

\[
\lambda f.f((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))f) \equiv \lambda f.f(Y f)
\]

The preconditions of improvement induction are satisfied, so we can conclude that \( \theta \nRightarrow Y \). Imperative programmers may be surprised to learn that Curry improves Turing!

7 EXAMPLE: NONDETERMINISM WITH FAILURE

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 semantics. 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.

For our semantics, we will use non-empty bags. A bag over a set \( X \) is a function \( f : X \rightarrow \mathbb{N} \) that counts how many times each element occurs. We write bags using the same notation as sets, but allowing for repeated elements. If there is some \( x \in X \) such that \( f(x) \) is non-zero, we say that the bag is is non-empty. We write the set of non-empty finite bags over \( X \) as \( \mathcal{B}^+(X) \).

We interpret programs with input type \( A \) and output type \( B \) using the domain \( A \rightarrow \mathcal{B}^+(B \cup \{\bot\}) \). In this case, we define \([P](i)(o)\) to be the number of possible execution paths that take input \( i \) and return output \( o \), with \([P](i)(\bot)\) as the number of execution paths that take input \( i \) and result in a failure. The operator choice then takes two functions and adds them by adding their bags pointwise, while fail will be the constant function to the bag \( \{\bot\} \).

There is a natural ordering on \( \mathcal{B}^+(B \cup \{\bot\}) \), defined as follows:

\[
b_1 < b_2 \iff \forall o \in B. b_1(o) \geq b_2(o) \land b_1(\bot) \geq b_2(\bot)
\]

That is, \( b_1 \) is less than \( b_2 \) if it contains more of every element. This ordering can be lifted pointwise to functions. Furthermore, we can define a metric on bags as follows,

\[
d_B(b_1, b_2) = \sup\{2^{-\min\{b_1(o), b_2(o)\}} \mid o \in B \cup \{\bot\}, b_1(o) \neq b_2(o)\}
\]

and this metric can also be lifted to functions:

\[
d(f, g) = \sup\{d(f(i), g(i)) \mid i \in A\}
\]

Finally, we note that limits exist in this metric space, and can be computed pointwise in terms of numerical limits. Furthermore, limits of bags respect the ordering on bags, and thus limits of functions respect the ordering on functions. Hence, our domain is a pre-ordered metric space.
It is straightforward to define the interpretation functions for terms and contexts, and the interpretation of contexts will be monotone and non-expansive provided that all the operations are interpreted as monotone, non-expansive functions. We can define our context \( \text{choice} \) to be \( \text{choice} [-] \text{fail} \), which means that its interpretation will be a function that adds a single failure, which is a contraction. Therefore, we have a denotational improvement semantics.

### 7.1 Sorting

Now we can apply our theory to an example. Consider the following nondeterministic sorting program which randomly permutes the input list and checks 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 \text{choice} operator:

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

\[
\text{selectRand}\,[x] = (x,[])
\]

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

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

\[
\text{checkOrdered}\,[] = []
\]

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

While the above definition for \text{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 \text{checkOrdered} and \text{permute} together, calculating a version of \text{sort} that recurses directly:

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

\[
\Rightarrow \quad \{ \text{definition of sort} \}
\]

\[
\text{checkOrdered}\,(\text{permute}\, xs)
\]

\[
\Rightarrow \quad \{ \text{definition of permute} \}
\]

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

\[
\Rightarrow \quad \{ \text{definition of checkOrdered} \}
\]

\[
\text{if} \, \text{null}\, xs \, \text{then} \, [] \, \text{else} \\
\text{let} & (r, rs) = \text{selectRand}\, xs; \, rs' = \text{permute}\, rs \, \text{in} \\
\text{if} & \text{all} \, (\geq r) \, rs' \, \text{then} \, r : \text{checkOrdered}\, rs' \, \text{else} \, \text{fail}
\]

\[
\Rightarrow \quad \{ \text{all is order-independent} \}
\]

\[
\text{if} \, \text{null}\, xs \, \text{then} \, [] \, \text{else} \\
\text{let} & (r, rs) = \text{selectRand}\, xs; \, rs' = \text{permute}\, rs \, \text{in} \\
\text{if} & \text{all} \, (\geq r) \, rs \, \text{then} \, r : \text{checkOrdered}\, rs' \, \text{else} \, \text{fail}
\]

\[
\Rightarrow \quad \{ \text{substituting rs'} \}
\]

\[
\text{if} \, \text{null}\, xs \, \text{then} \, [] \, \text{else}
\]
let \((r, rs) = \text{selectRand} \; xs\) in
\[
\text{if all } (\geq r) \; rs \text{ then } r : \check{\text{checkOrdered}}(\text{permute} \; rs) \text{ else fail}
\]

≡ \{definition of \text{sort}\}
\[
\text{if null} \; xs \text{ then } [] \text{ else let } (r, rs) = \text{selectRand} \; xs \text{ in}
\]
\[
\text{if all } (\geq r) \; rs \text{ then } r : \text{sort} \; rs \text{ else fail}
\]

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

\[
C = \lambda xs \rightarrow \text{if null} \; xs \text{ then } [] \text{ else let } (r, rs) = \text{selectRand} \; xs \text{ in}
\]
\[
\text{if all } (\geq r) \; rs \text{ then } r : [-] \; rs \text{ else fail}
\]

That is, \text{sort} \triangleright 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 \(\text{sort} \triangleright f\). However, since \(C\) corresponds to a contraction, 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 \(C[f] \triangleright f\).

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:

\[
\text{let } (r, rs) = \text{selectRand} \; xs \text{ in}
\]
\[
\text{if all } (\geq r) \; rs \text{ then } (r, rs) \text{ else fail}
\]

\[\triangleright\]
\[
\text{selectMin} \; xs
\]

This specification states that \text{selectMin} behaves in a similar manner to \text{selectRand}, with the key difference that it will only return values \(r\) 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 \(C[f] \triangleright f\):

\[
C[f]
\equiv \{\text{definition of } C\}
\]
\[
\lambda xs \rightarrow \text{if null} \; xs \text{ then } [] \text{ else let } (r, rs) = \text{selectRand} \; xs \text{ in}
\]
\[
\text{if all } (\geq r) \; rs \text{ then } r : f \; rs \text{ else fail}
\]

\[\triangleright\]
\[
\{\text{factoring out } r : f \; rs\}
\]
\[
\lambda xs \rightarrow \text{if null} \; xs \text{ then } [] \text{ else let } (r, rs) = \text{selectRand} \; xs \text{ in}
\]
\[
\text{let } (r', rs') = \text{if all } (\geq r) \; rs \text{ then } (r, rs) \text{ else fail}
\]
\[
in \; r' : f \; rs'
\]

\[\triangleright\]
\[
\{\text{rearranging lets}\}
\]
\[
\lambda xs \rightarrow \text{if null} \; xs \text{ then } [] \text{ else let } (r', rs') = \text{let } (r, rs) = \text{selectRand} \; xs \text{ in}
\]
\[
\text{if all } (\geq r) \; rs \text{ then } (r, rs) \text{ else fail}
\]
\[
in \; r' : f \; rs'
\]

\[\triangleright\]
\[
\{\text{specification of } \text{selectMin}\}
λxs → if null xs then [] else
    let (r', rs') = selectMin xs in r' : f rs'
  \{defining f to be this term\}
f
Finally, some simple renaming results in the following definition of selection sort:

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

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.

8 RELATED WORK

In this section we review a range of related work. We partition our review into two sub-sections: the first covers work on denotational semantics that capture cost information, while the second covers work on using metric spaces for program semantics and cost analysis.

8.1 Cost Semantics

Van Stone [2003] uses category theory to build denotational semantics that capture cost information. Her work focuses on functional programs, addressing both call-by-value and call-by-name, but not call-by-need, and primarily for the question of time costs. This work differs from the approach taken by improvement theory in that it is mainly concerned with individual programs, generating recurrence relations for their costs from the categorical semantics. The problem of comparing different programs is considered, and a denotational improvement relation is defined that is sound (but not complete) with respect to context-based operational improvement. The question of developing general proof techniques for working with this improvement relation is not addressed.

Danner, Licata, and Ramyaa [2015] give a denotational semantics for functional programs with inductive types where complexity is captured as a function of input size. This is based on the idea of an inductively-defined size function. In this case, the actual results of programs are not captured by the semantics, only the size of the output, which they call its potential. This results in an abstract, compositional semantics that includes enough information to allow for complexity analysis.

Ghica [2005] introduces slot games, a denotational cost semantics based on Hyland-Ong game semantics. This approach augments the usual game semantics by adding a new kind of action that corresponds to paying cost: the name “slot games” comes by analogy with slot machines. This semantics is demonstrated for a call-by-name language, Idealised Concurrent Algol, and is shown to be fully abstract with respect to a Sands-style contextual improvement relation. A number of simple program optimisations are shown to be improvements.

Endrullis, Hendriks, Klop, and Polonsky [2014] use clocked Böhm trees to prove that lambda calculus terms are inconvertible. This approach takes the usual Böhm trees and augments them with cost information, with the cost of each node recording the number of head reduction steps required to produce it. The same process is performed for Lévy-Longo and Berarducci trees. They define two notions of improvement for these trees, global and eventual improvement, and it is shown that if two terms M and N are β-converible, then there must be some reduct of M that globally improves N. However, this work does not directly address questions of efficiency.

8.2 Metric Spaces

Schellekens [1995] introduces a quasi-metric (a metric without the requirement of symmetry) on complexity functions for programs, showing that divide-and-conquer recurrences correspond to
fixed points of contractive functions on this space. This allows metric space techniques to be used when solving recurrences, and points toward a possible unified approach to denotational semantics and complexity theory. Romaguera and Schellekens [1999] extend this work, introducing the dual complexity space and proving a number of useful properties of both spaces.

Romaguera and Valero [2009] give a quantitative model of computation based on partial metric spaces, a generalisation of the usual notion of metric spaces where self-distance is not required to be zero. Given a partial metric space, there is a corresponding poset of formal balls; the authors relate these two semantic perspectives, showing that the poset of formal balls is a Scott domain if and only if the original partial metric space was complete.

De Bakker and Zucker [1982] use metric spaces to give denotational semantics for several concurrent process algebras including Hoare’s CSP and Milner’s CCS, defining semantic domains as the solutions to fixed point equations on metric spaces. De Bakker and Meyer [1987] build on this work, defining both metric space and domain theoretic semantics for concurrency based on streams, and showing that the two approaches coincide.

Krishnaswami and Benton [2011b] use ultrametric spaces to give a denotational semantics for reactive programs. In this context, non-expansiveness of a function corresponds to causality, the requirement that the current value of the output should not depend on a future value of the input. A domain specific language for reactive programs is specified and given an implementation in ML, and this implementation is proved correct with respect to the semantics. The same authors [2011a] later extend this work to deal with graphical user interfaces.

9 CONCLUSION AND FURTHER WORK

We used the notion of pre-ordered metric spaces to develop a new approach to tackling questions of program improvement, producing a generic theory of denotational improvement semantics that abstracts away from specific cost models while remaining a faithful model of program efficiency. We have proved a number of important results from the literature in our new framework, and given a range of example instantiations, including a cost-in-context semantics that mirrors the operational nature of improvement theory in a denotational way. Finally, our approach gives new insight into the problem of space improvement, explaining why this has been problematic 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 it is better. This may require us
to move to \textit{quasi}metric 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 \textit{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 \textit{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 \textit{strong} improvement. However, in some situations the notion of \textit{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 \textit{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.

Many optimisations involve some initial set-up costs to allow for more efficient operations later on. This includes the transformation from slow to fast reverse, as well as many applications of program transformations such the worker/wrapper transformation [Gill and Hutton 2009]. These are in some sense “moral” improvements, as for \textit{almost} all contexts the total cost will be less, but this is not covered by standard improvement theory. We would like to be able to integrate this into our theory without having to limit ourselves to weak improvement. The “eventual improvement” relation of Endrullis et al. [2014] suggests a possible line of attack here.

\section*{REFERENCES}


Martin Hötzel Escardó. 1999. A Metric Model of PCF. In \textit{Workshop on Realizability Semantics and Applications}.


Jennifer Hackett and Graham Hutton. 2015. Programs for Cheap!. In \textit{Logic in Computer Science}.


, Vol. N/A, No. POPL, Article . Publication date: July 2018.